UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Auto-generación de núcleos computacionales para redes neuronales sobre GPUs

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Del Campo Calvo, Francisco Javier

Tutor/a: Alonso Jordá, Pedro

Cotutor/a: Castelló Gimeno, Adrián

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Auto-generación de núcleos computacionales para redes neuronales sobre GPUs

TRABAJO FIN DE MÁSTER

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Autor: Francisco Javier Del Campo Calvo

Tutor: Adrián Castelló Gimeno
Pedro Alonso Jordá

Curso 2022-2023

Resumen

La adopción de las redes neuronales en prácticamente todos los ámbitos científicos está propiciando su uso en una amplia variedad de dispositivos. Estos dispositivos pueden ser de muy diversa naturaleza: desde grandes y complejos servidores de cómputo, hasta procesadores de bajo consumo como los integrados en los teléfonos móviles. Sin embargo, ya sean de un tipo u otro, casi todos incluyen un acelerador gráfico o GPU que puede ser utilizado para cómputo de propósito general.

Generar núcleos computacionales optimizados para cada acelerador de forma manual requiere de un enorme esfuerzo por parte de los desarrolladores, de hecho, en la práctica, esto se lleva a cabo solamente para algunos modelos. El objetivo del presente trabajo es automatizar la generación de código optimizado para cualquier tipo de GPU mediante una herramienta conocida como Apache TVM. Esta herramienta permite especificar la operación a optimizar a un alto nivel y, a través de una serie de modificaciones intermedias por parte del framework, generar el código optimizado.

En este trabajo se desarrolla código para optimizar la operación de multiplicación de matrices, la cual es ampliamente utilizada en la mayoría de los campos científicos, incluyendo las redes neuronales.

Una vez generado el código con TVM, se lleva a cabo un estudio de prestaciones, comparando su rendimiento con el de la librería de altas prestaciones NVIDIA cuBLAS y el de otra utilidad de TVM llamada Auto-Tuning (AutoScheduler), que permite automatizar el proceso de desarrollo a partir de una descripción del cómputo.

Los resultados muestran que el código desarrollado obtiene un rendimiento similar al obtenido por cuBLAS, y supera el generado por AutoScheduler en multiplicaciones de matrices de dimensiones grandes.

Al aplicar esta comparativa a un caso de estudio real, la red neuronal ResNet50-v1.5, donde el escenario de aplicación cambia a matrices muy rectangulares, ha sido posible vencer a cuBLAS en algunas de las capas que corresponden a los productos de matrices más rectangulares y al AutoScheduler en la mayoría de las capas.

Palabras clave: Multiplicación de matrices, Apache TVM, GPU, generación automática, redes neuronales, convolución

---

Resum

L'adopció de les xarxes neuronals en pràcticament tots els àmbits científics està propiciant el seu ús en una àmplia varietat de dispositius. Aquests dispositius poden ser de molt diversa naturalesa: des de grans i complexos servidors de còmput, fins a processadors de baix consum com els integrats en els telèfons mòbils. No obstant això, ja siguen d'un tipus o un altre, quasi tots inclouen un accelerador gràfic o GPU que pot ser utilitzat per a còmput de propòsit general.

Generar nuclis computacionals optimitzats per a cada accelerador de manera manual requereix d'un enorme esforç per part dels desenvolupadors, de fet, en la pràctica, això es duu a terme solament per a alguns models. L'objectiu del present treball és automatitzar la generació de codi optimitzat per a qualsevol tipus de GPU mitjançant una eina coneguda com a Apache TVM. Aquesta eina permet especificar l'operació a optimitzar a un alt nivell i, a través d'una sèrie de modificacions intermèdies per part del framework, generar el codi optimitzat.

En aquest treball es desenvolupa codi per a optimitzar l'operació de multiplicació de matrius, la qual és àmpliament utilitzada en la majoria dels camps científics, incloent-hi les xarxes neuronals.

Una vegada generat el codi amb TVM, es duu a terme un estudi de prestacions, comparant el seu rendiment amb el de la llibreria d'altes prestacions NVIDIA cuBLAS i el d'una altra utilitat de TVM anomenada Auto-Tuning (AutoScheduler), que permet automatitzar el procés de desenvolupament a partir d'una descripció del còmput.

Els resultats mostren que el codi desenvolupat obté un rendiment similar a l'obtingut per cuBLAS, i supera el generat per AutoScheduler en multiplicacions de matrius de dimensions grans.

En aplicar aquesta comparativa a un cas d'estudi real, la xarxa neuronal ResNet50-v1.5, on l'escenari d'aplicació canvia a matrius molt rectangulars, ha sigut possible véncer a cuBLAS en algunes de les capes que corresponen als productes de matrius més rectangulars i al AutoScheduler en la majoria de les capes.

Paraules clau: Multiplicació de matrius, Apache TVM, GPU, generació automàtica, xarxes neuronals, convolució

Abstract

The adoption of neural networks in virtually all scientific fields is leading to their use in a wide variety of devices. These devices can range from large, complex computational servers to low-power processors such as those embedded in mobile phones. However, whether they are of one type or another, almost all include a graphics accelerator or GPU that can be used for general-purpose computing.

Generating optimized computational cores for each accelerator manually requires enormous effort on the part of developers; in fact, in practice, this is only done for some models. The aim of the present project is to automate the generation of optimized code for any type of GPU using a tool known as Apache TVM. This tool allows to specify the operation to be optimized at a high level and, through a series of intermediate modifications by the framework, to generate the optimized code.

In this project, code is developed to optimize the matrix multiplication operation, which is widely used in most scientific fields, including neural networks.

Once the TVM code is generated, a performance study is carried out, comparing its performance with that of the NVIDIA cuBLAS high-performance library and that of another TVM utility called Auto-Tuning (AutoScheduler), which allows automating the development process from a description of the computation.

The results show that the developed code obtains a performance similar to that obtained by cuBLAS, and outperforms that generated by AutoScheduler in multiplications of high-dimensional matrices.

By applying this comparison to a real case study, the ResNet50-v1.5 neural network, where the application scenario changes to very rectangular matrices, it has been possible to beat cuBLAS in some of the layers corresponding to the products of more rectangular matrices and AutoScheduler in most of the layers.

Key words: Matrix multiplication, Apache TVM, GPU, automatic generation, neural networks, convolution

Índice general

Índice general v
Índice de figuras vii
Índice de tablas xi

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 1
1.3 Estructura de la memoria . . . 2
2 Entorno de trabajo 5
2.1 Introducción . . . 5
2.1.1 Redes Neuronales . . . 5
2.1.2 Computación de Altas Prestaciones en GPU . . . 11
2.2 Hardware y software utilizado . . . 22
2.2.1 Descripción del hardware utilizado . . . 22
2.2.2 Software utilizado: Apache TVM . . . 29
3 Desarrollo e implementación 35
3.1 Introducción . . . 35
3.2 Progreso del algoritmo . . . 37
3.3 Resumen de las versiones . . . 60
4 Resultados numéricos 63
4.1 Comparativa con otras librerías . . . 63
4.2 Caso de estudio: ResNet50-v1.5 . . . 66
4.2.1 Descripción de la red ResNet-50v1.5 . . . 66
4.2.2 Resultados obtenidos . . . 72
5 Conclusiones 77
Bibliografía 79

Apéndices
A Comparativa de GPU 85
B Estudio sobre el rendimiento 87

Índice de figuras

2.1 Componentes de un Perceptrón. 8
2.2 Gráfico comparativo de las arquitecturas de CPU y GPU. 17
2.3 Aplicaciones de la computación en GPU. 18
2.4 Ejemplo de la escalabilidad automática que el modelo de CUDA ofrece. . 19
2.5 Jerarquía de memoria en CUDA. 21
2.6 Arquitectura de Volta GV100. 25
2.7 Arquitectura de un SM Volta. 26
2.8 Arquitectura de Turing TU104. 28
2.9 Arquitectura de un SM Turing. 29
2.10 Características de Apache TVM. 30
2.11 Proceso de optimización de modelos de TVM. 30

3.1 Operación GEMM utilizando Tensor Expressions de TVM. 36
3.2 Compilado y ejecución de la función en el device. 36
3.3 Schedule de la Versión 1. 37
3.4 Código generado por el schedule de la Versión 1. 38
3.5 Gráfico del rendimiento de la versión V1 en función de num threads en la máquina Quadro. 38
3.6 Gráfico del rendimiento de la versión V1 en función de num threads en la máquina Volta. 38
3.7 Schedule de la Versión 2. 39
3.8 Gráfico del rendimiento de la versión V2 en función de num threads en la máquina Quadro. 39
3.9 Gráfico del rendimiento de la versión V2 en función de num threads en la máquina Volta. 39
3.10 Schedule de la Versión 3. 40
3.11 Gráfico del rendimiento de la versión V3 en función de num threads en la máquina Quadro. 40
3.12 Gráfico del rendimiento de la versión V3 en función de num threads en la máquina Volta. 40
3.13 Schedule de la Versión 4. 41
3.14 Gráfico del rendimiento de la versión V4 en función de num threads en la máquina Quadro. 41
3.15 Gráfico del rendimiento de la versión V4 en función de num threads en la máquina Volta. 41
3.16 Schedule de la Versión 5. 42
3.17 Gráfico del rendimiento de la versión V5 en función de elements per thread en la máquina Quadro. 43
3.18 Gráfico del rendimiento de la versión V5 en función de elements per thread en la máquina Volta. 43
3.19 Gráfico del rendimiento de la versión V5 en función de num threads en la máquina Quadro. 43

3.20 Gráfico del rendimiento de la versión V5 en función de num threads en la máquina Volta. . . . 43
3.21 Schedule de la Versión 6. . . . 44
3.22 Gráfico del rendimiento de la versión V6 en función de elements per thread en la máquina Quadro. . . . 44
3.23 Gráfico del rendimiento de la versión V6 en función de elements per thread en la máquina Volta. . . . 44
3.24 Gráfico del rendimiento de la versión V6 en función de num threads en la máquina Quadro. . . . 45
3.25 Gráfico del rendimiento de la versión V6 en función de num threads en la máquina Volta. . . . 45
3.26 Schedule de la Versión 7. . . . 46
3.27 Gráfico del rendimiento de la versión V7 en función de elements per thread en la máquina Quadro. . . . 47
3.28 Gráfico del rendimiento de la versión V7 en función de elements per thread en la máquina Volta. . . . 47
3.29 Gráfico del rendimiento de la versión V7 en función de num threads en la máquina Quadro. . . . 47
3.30 Gráfico del rendimiento de la versión V7 en función de num threads en la máquina Volta. . . . 47
3.31 Gráfico del rendimiento de la versión V7 en función de step en la máquina Quadro. . . . 48
3.32 Gráfico del rendimiento de la versión V7 en función de step en la máquina Volta. . . . 48
3.33 Schedule de la Versión 8. . . . 49
3.34 Gráfico del rendimiento de la versión V8 en función de elements per thread en la máquina Quadro. . . . 50
3.35 Gráfico del rendimiento de la versión V8 en función de elements per thread en la máquina Volta. . . . 50
3.36 Gráfico del rendimiento de la versión V8 en función de num threads en la máquina Quadro. . . . 50
3.37 Gráfico del rendimiento de la versión V8 en función de num threads en la máquina Volta. . . . 50
3.38 Gráfico del rendimiento de la versión V8 en función de step en la máquina Quadro. . . . 51
3.39 Gráfico del rendimiento de la versión V8 en función de step en la máquina Volta. . . . 51
3.40 Schedule de la Versión 9. . . . 52
3.41 Gráfico del rendimiento de la versión V9 en función de elements per thread en la máquina Quadro. . . . 52
3.42 Gráfico del rendimiento de la versión V9 en función de elements per thread en la máquina Volta. . . . 52
3.43 Gráfico del rendimiento de la versión V9 en función de num threads en la máquina Quadro. . . . 53
3.44 Gráfico del rendimiento de la versión V9 en función de num threads en la máquina Volta. . . . 53
3.45 Gráfico del rendimiento de la versión V9 en función de step en la máquina Quadro. . . . 53
3.46 Gráfico del rendimiento de la versión V9 en función de step en la máquina Volta. . . . 53
3.47 Schedule de la Versión 10. . . . 54

3.48 Gráfico del rendimiento de la versión V10 en función de elements per thread en la máquina Quadro. . . . 55
3.49 Gráfico del rendimiento de la versión V10 en función de elements per thread en la máquina Volta. . . . 55
3.50 Gráfico del rendimiento de la versión V10 en función de num threads en la máquina Quadro. . . . 55
3.51 Gráfico del rendimiento de la versión V10 en función de num threads en la máquina Volta. . . . 55
3.52 Gráfico del rendimiento de la versión V10 en función de step en la máquina Quadro. . . . 56
3.53 Gráfico del rendimiento de la versión V10 en función de step en la máquina Volta. . . . 56
3.54 Accediendo a la memoria compartida, ejemplo de un bank conflict. . . . 57
3.55 Evitando el bank conflict mediante el uso de virtual threads de TVM. . . . 58
3.56 Schedule de la Versión final. . . . 58
3.57 Gráfico del rendimiento de la versión VFINAL en función de elements per thread en la máquina Quadro. . . . 59
3.58 Gráfico del rendimiento de la versión VFINAL en función de elements per thread en la máquina Volta. . . . 59
3.59 Gráfico del rendimiento de la versión VFINAL en función de num threads en la máquina Quadro. . . . 59
3.60 Gráfico del rendimiento de la versión VFINAL en función de num threads en la máquina Volta. . . . 59
3.61 Gráfico del rendimiento de la versión VFINAL en función de step en la máquina Quadro. . . . 60
3.62 Gráfico del rendimiento de la versión VFINAL en función de step en la máquina Volta. . . . 60
3.63 Comparativa del rendimiento a lo largo de las diferentes versiones implementadas en la máquina Quadro. . . . 60
3.64 Comparativa del rendimiento a lo largo de las diferentes versiones implementadas en la máquina Volta. . . . 60

4.1 Empleo del AutoScheduler de TVM para optimizar un modelo. . . . 64
4.2 Gráfico comparativo de rendimiento de GEMM usando la librería cuBLAS, la función AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro. . . . 65
4.3 Gráfico comparativo de rendimiento de GEMM usando la librería cuBLAS, la función AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta. . . . 65
4.4 Distribución del tiempo para el modelo AlexNet con ImageNet. . . . 69
4.5 Capa fully-connected expresada como una operación GEMM. . . . 69
4.6 Capa convolucional expresada de forma matricial. . . . 70
4.7 Imagen de entrada expresada como una matriz bidimensional. . . . 70
4.8 Capa convolucional expresada como una operación GEMM. . . . 71
4.9 Gráfico comparativo de rendimiento para las capas de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro. . . . 73
4.10 Gráfico comparativo de rendimiento para las capas de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta. . . . 73

4.11 Gráfico del tiempo agregado total de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro. 74
4.12 Gráfico del tiempo agregado total de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta. 75

A.1 Comparativa de las dos GPU empleadas. 86

B.1 Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 4096 en la máquina Quadro. 87
B.2 Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 4096 en la máquina Volta. 87
B.3 Gráfico del rendimiento en función de num threads para un tamaño de matriz de 4096 en la máquina Quadro. 88
B.4 Gráfico del rendimiento en función de num threads para un tamaño de matriz de 4096 en la máquina Volta. 88
B.5 Gráfico del rendimiento en función de step para un tamaño de matriz de 4096 en la máquina Quadro. 88
B.6 Gráfico del rendimiento en función de step para un tamaño de matriz de 4096 en la máquina Volta. 88
B.7 Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 8192 en la máquina Quadro. 89
B.8 Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 8192 en la máquina Volta. 89
B.9 Gráfico del rendimiento en función de num threads para un tamaño de matriz de 8192 en la máquina Quadro. 89
B.10 Gráfico del rendimiento en función de num threads para un tamaño de matriz de 8192 en la máquina Volta. 89
B.11 Gráfico del rendimiento en función de step para un tamaño de matriz de 8192 en la máquina Quadro. 90
B.12 Gráfico del rendimiento en función de step para un tamaño de matriz de 8192 en la máquina Volta. 90
B.13 Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro. 91
B.14 Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta. 91
B.15 Gráfico del rendimiento en función de num threads para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro. 91
B.16 Gráfico del rendimiento en función de num threads para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta. 91
B.17 Gráfico del rendimiento en función de step para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro. 92
B.18 Gráfico del rendimiento en función de step para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta. 92
B.19 Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro. 93
B.20 Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=2048, N=2048, K=8192, K=2048 en la máquina Volta. 93
B.21 Gráfico del rendimiento en función de num threads para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro. 93
B.22 Gráfico del rendimiento en función de num threads para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Volta. 93

B.23 Gráfico del rendimiento en función de step para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro. 94
B.24 Gráfico del rendimiento en función de step para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Volta. 94

Índice de tablas

4.1 Arquitectura de las redes de la familia ResNet. 68
4.2 Operaciones GEMM de la red ResNet50-v1.5. 71
4.3 Combinaciones de parámetros óptimas para las operaciones GEMM correspondientes a las capas de la red ResNet50-v1.5. 72
4.4 Resultados de tiempo agregado total de la ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en ambas máquinas. 74

CAPÍTULO 1
Introducción

En este capítulo se expondrán las motivaciones y objetivos que tiene este Trabajo de Fin de Máster y se describirá su estructura, explicando y resumiendo el contenido de cada capítulo.

1.1 Motivación

La Inteligencia Artificial cada vez está presente en más aspectos de nuestra vida, siendo sus casos de uso numerosos tanto en las más ambiciosas investigaciones científicas, médicas o financieras como para entornos más mundanos como puede ser un chatbot de atención al cliente.

Uno de los campos con mayor proyección en la Inteligencia Artificial es el llamado Machine Learning, especialmente empleando las Redes Neuronales Profundas o Deep Learning, las cuales se inspiran en el modelo de procesamiento de información del cerebro para permitir que las máquinas puedan "aprender".

Debido a la gran utilidad y versatilidad de estos modelos de Machine Learning, contar con una implementación que haga uso de la llamada Computación de Altas Prestaciones para poder realizar los cuantiosos cálculos que éstos requieren a las velocidades de vértigo que permiten los más modernos sistemas HPC posibilitará un sinnúmero de beneficios en todo tipo de campos.

Uno de los enfoques más importantes de la Computación de Altas Prestaciones es la llamada aceleración por hardware, la cual hace uso de elementos de procesamiento que permiten realizar ciertas tareas más eficientemente de lo que es posible empleando una CPU de propósito general. El uso de GPU es posiblemente el ejemplo más utilizado de aceleración por hardware, y es el que se tratará en este trabajo.

La plataforma NVIDIA CUDA permite la implementación de código que aproveche la potencia de cómputo paralelo de las GPU de NVIDIA para poder obtener un rendimiento mucho más elevado que el que permite una CPU. Existen además herramientas que permiten automatizar el desarrollo de este código, permitiendo la generación de código optimizado y consciente de la arquitectura de la GPU, incorporando además una capa de abstracción que facilita el proceso de desarrollo.

1.2 Objetivos

El principal objetivo de este Trabajo de Fin de Máster es el desarrollo de un código que permita optimizar la operación de producto matricial. Esta operación está en la base

de muchos de los modelos de redes neuronales, por lo que optimizar esta operación permitiría reducir de forma considerable el tiempo de ejecución de estos modelos en GPU de la marca NVIDIA.

Para ello, se va a utilizar el framework Apache TVM, que permite generar código CUDA que se ejecute en la GPU y saque partido a su elevada potencia de cómputo. Las GPU son especialmente eficientes en la realización de operaciones de coma flotante, siendo estas operaciones las que aglutinan la práctica totalidad del tiempo de cómputo de un producto matricial.

Se va a implementar un código en Python que haga uso del paquete TVM para llevar a cabo la generación de código CUDA optimizado para la arquitectura de la GPU utilizada que realice de forma eficiente este producto matricial. Lo que se busca es poder realizar con el mayor rendimiento posible los productos matriciales empleados en las redes neuronales. Como se verá, las multiplicaciones matriciales que acostumbran a aparecer en este contexto son muy grandes y muy rectangulares, mientras que las implementaciones de altas prestaciones más comunes suelen estar optimizadas para las multiplicaciones de matrices cuadradas.

Una vez se haya dado por concluido el proceso de desarrollo, tratando de incluir todas las optimizaciones que permita Apache TVM, se realizarán mediciones de rendimiento en GFLOPS y se llevará a cabo una comparativa con los resultados obtenidos por la implementación de altas prestaciones cuBLAS. También se comparará con los resultados conseguidos empleando otra funcionalidad de Apache TVM llamada *Auto-Tuning*, que permite automatizar el proceso de optimización a partir de una descripción del cómputo a realizar, prescindiendo del proceso de desarrollo.

Se realizarán en primer lugar medidas de rendimiento con productos de matrices de distintos tamaños y dimensiones, tanto cuadradas como rectangulares. A continuación se pondrá en práctica un caso real, tomando las operaciones usadas en la red neuronal ResNet50-v1.5, viendo así cuál sería el desempeño real de nuestro código en la ejecución de un modelo de Deep Learning.

Todas estas mediciones se llevarán a cabo en dos máquinas distintas, las cuales cuentan con dos GPU diferentes, una perteneciente a la línea de productos Quadro, enfocada a estaciones de trabajo de diseño profesional, y la otra perteneciente a la línea Tesla, enfocada al *GPGPU* (computación de propósito general en GPU).

Por tanto, podemos concretar los objetivos de este Trabajo de Fin de Máster en los siguientes:

1. El desarrollo de un código que permita optimizar la operación de producto matricial en GPU de NVIDIA empleando la herramienta de generación de código Apache TVM.

2. La realización de un estudio de rendimiento, comparando los resultados del algoritmo desarrollado con otras implementaciones de altas prestaciones existentes.

3. La aplicación de este código a un caso de estudio real de los productos matriciales empleados en la red neuronal ResNet50-v1.5, realizando la misma comparativa de rendimiento.

1.3 Estructura de la memoria

A continuación, se explica cómo está estructurado este trabajo, explicando a grandes rasgos los objetivos de cada capítulo:

1. *Capítulo 2: Entorno de trabajo*: Se presentan y exponen todos los antecedentes teóricos necesarios para la comprensión del Trabajo, además de llevar a cabo una descripción del hardware empleado (las dos GPU) y sus arquitecturas, así como de la herramienta software utilizada (Apache TVM).

2. *Capítulo 3: Desarrollo e implementación*: Se explica detalladamente el proceso de desarrollo del código, versión por versión, ilustrando los cambios y mejoras realizados entre éstas, sus motivaciones y cómo afectan al rendimiento.

3. *Capítulo 4: Resultados numéricos*: Se exponen los resultados obtenidos, que se corresponden con las mediciones y comparaciones de rendimiento en GFLOPS del código implementado, la librería cuBLAS y la función Auto-Tuning de TVM. En primer lugar, con matrices cuadradas y rectangulares de prueba y, posteriormente, a partir de un caso de estudio real de una Red Neuronal.

4. *Capítulo 5: Conclusiones*: Se enuncian las principales conclusiones obtenidas en la realización de este Trabajo, así como los posibles estudios futuros tras este.

CAPÍTULO 2
Entorno de trabajo

Este capítulo consta de dos partes, y en él se presenta el marco de trabajo de este Proyecto. En la primera parte se exponen todos los preliminares necesarios para poder comprender los objetivos del Trabajo, mientras que en la segunda se trata el hardware y software utilizado en el desarrollo de éste.

2.1 Introducción

Esta parte se divide en dos secciones, estando la primera enfocada al campo de la Inteligencia Artificial y las Redes Neuronales y la segunda a la Computación de Altas Prestaciones en GPU haciendo un especial énfasis en la plataforma CUDA.

2.1.1. Redes Neuronales

Inteligencia Artificial

En 1950, el matemático Alan Turing se hizo una pregunta: "¿Pueden pensar las máquinas?". Esta simple pregunta transformaría el mundo [1]. El artículo de Alan Turing "Computing Machinery and Intelligence" [2] y el consiguiente "Test de Turing" sentaron las bases de la Inteligencia Artificial, su visión y sus objetivos.

Una posible definición de Inteligencia Artificial (IA) [3] sería *la simulación de procesos de inteligencia humana por parte de máquinas, especialmente sistemas informáticos*. Estos procesos incluyen el aprendizaje (la adquisición de información y reglas para el uso de la información), el razonamiento (usando las reglas para llegar a conclusiones aproximadas o definitivas) y la autocorrección, entre otros.

Por otra parte, la Comisión Europea la define como *aquellos sistemas que manifiestan un comportamiento inteligente, al ser capaces de analizar el entorno y realizar acciones, con cierto grado de autonomía, con el fin de alcanzar objetivos específicos*.

La IA es un amplio campo que abarca muchas disciplinas diferentes, incluidas la informática, el análisis de datos y las estadísticas, la ingeniería de hardware y software, la lingüística, la neurociencia y hasta la filosofía y la psicología.

Una posible forma de clasificar distintos tipos de IA se puede encontrar en la categorización que hizo Arend Hintze, profesor de Biología Integradora e Ingeniería y Ciencias de la Computación en la Universidad Estatal de Michigan. Hintze distingue entre 4 tipos de Inteligencia Artificial, tanto existentes hoy en día como todavía por desarrollarse [3, 4, 5]:

1. *Máquinas reactivas*: Son el tipo más básico de IA. Se basan en decisiones sobre el presente, es decir, no tienen memoria y, por lo tanto, no pueden mirar al pasado para aprender de experiencias pasadas y son incapaces de evolucionar. Un ejemplo de este tipo de IA se puede encontrar en Deep Blue, el ordenador que derrotó al Gran Maestro de ajedrez Garri Kaspárov en 1997.

2. *Memoria limitada*: Son capaces de mirar al pasado, pero de una forma limitada y temporal. De esta manera, pueden almacenar la información que recogen durante cierto tiempo y añadirla a su programación para crear nuevos patrones de comportamiento y respuesta para un futuro no lejano. El Deep Learning, o Aprendizaje Profundo, un subconjunto del Machine Learning, se considera inteligencia artificial con memoria limitada.

3. *Teoría de la mente*: Este término hace referencia a IA capaz de emular la mente humana y que tiene capacidades de toma de decisiones similares a las de un ser humano, lo cual incluye reconocer y recordar emociones, y reaccionar en situaciones sociales como lo haría un ser humano. Actualmente no existe IA con teoría de la mente, pero se están investigando distintas posibilidades.

4. *Autoconciencia*: Se trataría de una Inteligencia Artificial que ha desarrollado conciencia de sí misma y es capaz de reconocerse como una entidad independiente, que puede tomar sus propias decisiones, diferenciando entre ella y los objetos, personas y sistemas que la rodean. Este tipo de IA solo existe de forma hipotética, y se cree que diseñar este tipo de IA está muy lejos de materializarse.

Algunas de las múltiples aplicaciones y casos de uso de la Inteligencia Artificial en la actualidad son el reconocimiento de voz, los chatbots de servicio al cliente, la visión computacional, la detección de fraudes bancarios, los motores de recomendación en compras online y otros múltiples campos donde se encuentra en desarrollo, como la asistencia sanitaria o legal [6].

Debido a que la Inteligencia Artificial es capaz de automatizar procesos que anteriormente requerían intervención humana, permitiendo además reducir el factor error humano consiguiendo una mayor precisión y rapidez, es una de las tecnologías más influyentes en la actualidad.

Machine Learning

El Machine Learning o Aprendizaje Automático es una rama de la Inteligencia Artificial que trata de lograr que las máquinas puedan aprender sin una programación previa. Se dice que un agente aprende cuando su desempeño mejora con la experiencia y mediante el uso de datos [7].

La principal diferencia entre IA y Machine Learning (ML) es que mientras que la IA es concepto más amplio de habilitar una máquina o un sistema para detectar, razonar, actuar o adaptarse como una persona, el ML es una aplicación de la IA que permite que las máquinas extraigan conocimiento de los datos y aprendan de ellos de forma autónoma [8].

Los algoritmos de aprendizaje automático mejoran el rendimiento con el paso del tiempo, ya que se *entrenan* (es decir, se exponen a más datos). Los modelos de aprendizaje automático son el resultado o lo que el programa aprende cuando se ejecuta un algoritmo en los datos de entrenamiento. Cuantos más datos se usen, mejor será el modelo y las acciones resultantes serán mejores y más precisas.

La adaptabilidad del Machine Learning lo convierte en una excelente opción en escenarios en los que los datos siempre cambian, la naturaleza de la solicitud o la tarea siempre se transforma o la codificación de una solución sería realmente imposible [9].

Existen cuatro tipos de algoritmos de Machine Learning [10, 11]:

- *Aprendizaje supervisado*: Estos algoritmos cuentan con un aprendizaje previo basado en un sistema de etiquetas asociadas a unos datos (datos de entrenamiento) que les permiten tomar decisiones o hacer predicciones. Los conjuntos de datos están etiquetados para que los patrones puedan ser detectados y utilizados para así poder etiquetar nuevos conjuntos de datos.
- *Aprendizaje no supervisado*: Estos algoritmos no cuentan con un conocimiento previo. En su entrenamiento se enfrentan a conjuntos de datos no etiquetados con el objetivo de encontrar patrones y relaciones que permitan organizarlos de alguna manera.
- *Aprendizaje semisupervisado*: En este tipo de aprendizaje, se toman en cuenta los datos supervisados y no supervisados, combinando los dos anteriores para que pueda clasificar de manera adecuada. En estos algoritmos la máquina cuenta con una serie de datos parcialmente etiquetados y realiza su tarea usando los datos etiquetados para entender los parámetros e interpretar los datos no etiquetados.
- *Aprendizaje por refuerzo*: Su objetivo es que un algoritmo aprenda a partir de la propia experiencia. Esto es, que sea capaz de tomar la mejor decisión ante diferentes situaciones de acuerdo a un proceso de prueba y error en el que se recompensan las decisiones correctas.

Los lenguajes de programación más populares para la implementación de algoritmos de Machine Learning son Python y R, y entre los paquetes de software más empleados se encuentran *TensorFlow* [12], *Dlib* [13], *MLPACK* [14], *Apache Spark MLlib* [15] y *Weka* [16]. Entre las técnicas de Machine Learning más empleadas se encuentran los árboles de decisión, las reglas de asociación, los algoritmos genéticos, las redes neuronales artificiales, las máquinas de vectores de soporte, los algoritmos de agrupamiento o *clustering* y las redes bayesianas [11].

Redes Neuronales y Deep Learning

Una red neuronal es un método de la Inteligencia Artificial que enseña a los ordenadores a procesar datos de una manera que está inspirada en la forma en que lo hace el cerebro humano. El objetivo de una red neuronal es aprender modificándose automáticamente a si misma de forma que puede llegar a realizar tareas complejas que no podrían ser realizadas mediante la clásica programación basada en reglas. De esta forma se pueden automatizar funciones que en un principio solo podrían ser realizadas por personas [17, 18].

Esta formada por un conjunto de nodos conocidos como neuronas artificiales que están conectadas y transmiten señales entre sí. Estos nodos reciben información del exterior o de otras neuronas, de manera similar a los impulsos nerviosos que reciben las neuronas del cerebro humano, las procesan y generan un valor de salida que alimenta a otras neuronas de la red o son la salida hacia el exterior de la red [19].

Estas unidades neuronales se conectan con fuerzas de conexión variables o *ponderaciones*. Al principio, todas las ponderaciones de la red son aleatorias, por lo que las respuestas que resultan de ésta serán incorrectas. La red aprende a través del *entrenamiento*.

Continuamente se presentan a la red ejemplos para los que se conoce el resultado, y las respuestas que proporciona se comparan con los resultados conocidos. La información procedente de esta comparación se pasa hacia atrás a través de la red, realizando ajustes en las ponderaciones gradualmente cuando se realiza una predicción incorrecta.

A medida que progresa el entrenamiento, la red se va haciendo cada vez más precisa en la replicación de resultados conocidos. Este proceso se repite muchas veces y la red sigue mejorando sus predicciones hasta haber alcanzado uno o varios criterios de parada. Una vez entrenada, la red se puede aplicar a casos futuros en los que se desconoce el resultado [20].

Una red neuronal básica tiene neuronas artificiales interconectadas en tres capas [17]:

- *Capa de entrada:* La información del mundo exterior entra en la red neuronal artificial desde la capa de entrada. Los nodos de entrada procesan los datos, los analizan o los clasifican y los pasan a la siguiente capa.

- *Capa oculta:* La/s capa/s oculta/s toma/n su entrada de la capa de entrada o de otras capas ocultas. Las redes neuronales artificiales pueden tener una gran cantidad de capas ocultas. Cada capa oculta analiza la salida de la capa anterior, la procesa aún más y la pasa a la siguiente capa.

- *Capa de salida:* La capa de salida proporciona el resultado final de todo el procesamiento de datos que realiza la red neuronal artificial. Puede tener uno o varios nodos. Por ejemplo, para un problema de clasificación binaria, la capa de salida tendrá un nodo de salida que dará como resultado 1 o 0. Sin embargo, en un problema de clasificación multiclase, la capa de salida puede estar formada por más de un nodo de salida.

Las redes neuronales se basan en una estructura básica, en el *Perceptrón* y en el mecanismo de *Backpropagation*, que permite a la neurona "aprender por sí misma" y descubrir la información oculta en los datos de entrada con los que es "entrenada" [21].

El perceptrón es el tipo más sencillo y básico de red neuronal [22, 23]. La Figura 2.1 muestra los componentes de un perceptrón [24]:

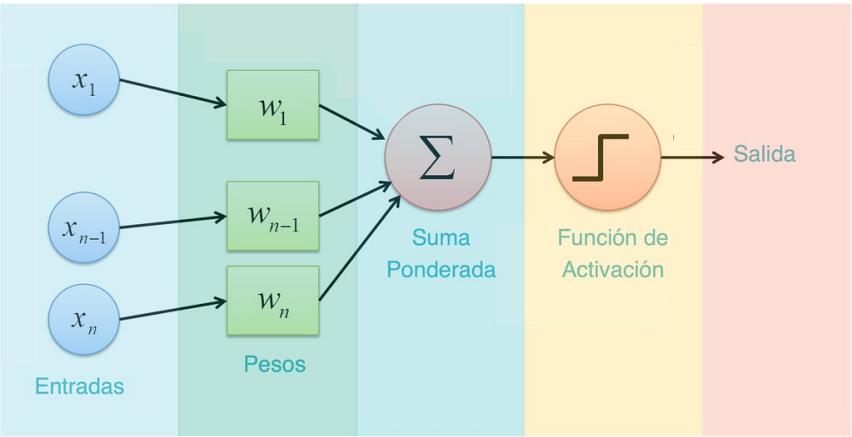

Figura 2.1: Componentes de un Perceptrón.

- *Entrada*. Las entradas en el algoritmo del perceptrón se entienden como $x1$, $x2$, $x3$, $x4$ y así sucesivamente. Todas estas entradas denotan los valores del perceptrón de características y la ocurrencia total de las características.

- *Pesos*. Se observan como valores que se planifican al largo de la sesión de preparación del perceptrón. Los pesos ofrecen un valor preliminar en el inicio del aprendizaje del algoritmo. Con la ocurrencia de cada inexactitud de entrenamiento, los valores de los pesos se actualizan. Estos se representan principalmente como $w1$, $w2$, $w3$, $w4$ y así sucesivamente.

- *Suma ponderada*. Es la proliferación de cada valor de entrada o característica asociada con el valor de paso correspondiente.

- *Función de activación*. Cada función de activación, o no lineal, toma un único número y realiza una determinada operación matemática fija sobre él. Hay varias funciones de activación que se pueden encontrar en la práctica, las más comunes son la Sigmoide o la ReLU (unidad lineal rectificada).

- *Salida*. La suma ponderada se pasa a la función de activación y cualquier valor que obtengamos después del cálculo es nuestra salida predicha.

Un perceptrón puede ser *simple*, si está compuesto de una única capa, o *multicapa*, si está formado por más de una. El perceptrón simple incluye una red *feedforward* (es decir, una en la cual las conexiones entre las neuronas no forman un ciclo) que depende de una función de transferencia de umbral en su modelo. Es el tipo más sencillo de red neuronal artificial, y puede analizar solo objetos linealmente separables con resultados binarios [24].

Los valores de entrada alimentan el modelo del perceptrón, el modelo se ejecuta con los valores de entrada, y si el valor estimado es el mismo que la salida requerida, entonces el rendimiento del modelo se encuentra satisfecho, por lo que los pesos no exigen cambios. Por otra parte, si el modelo no cumple con el resultado requerido, entonces se realizan algunos cambios en los pesos para minimizar los errores [24].

En cambio, un modelo de perceptrón multicapa tiene una estructura similar a la de un modelo de perceptrón de una sola capa con más número de capas ocultas. También se denomina algoritmo de retropropagación o *backpropagation*. Se ejecuta en dos etapas: la etapa hacia adelante y la etapa hacia atrás.

En la etapa hacia adelante, las funciones de activación se originan desde la capa de entrada a la capa de salida, y en la etapa hacia atrás, el error entre el valor real observado y el valor demandado se origina hacia atrás en la capa de salida para modificar los pesos. En este caso, la función de activación ya no es lineal. En su lugar se despliegan funciones de activación no lineales como las funciones sigmoides, TanH, funciones de activación ReLU, entre otras.

La inclusión de más capas permite añadir información que no estaba antes. Se toman los datos de entrada, se exploran y se sacan las características que mejor puedan ayudar a entender qué está pasando. Durante el proceso de aprendizaje, cada capa "aprende" a encontrar y detectar las características que mejor ayudan a clasificar los datos.

Los modelos de redes neuronales que constan de más de tres capas se denominan redes neuronales "profundas" o *Deep Learning* (Aprendizaje Profundo). Aunque en realidad no es el número de capas lo que define el Deep Learning. El concepto subyacente en el Deep Learning es el procesamiento de los datos de forma jerárquica. Es decir, el uso de redes neuronales para obtener representaciones cada vez más significativas de los datos mediante el aprendizaje por capas.

Cada capa extrae características de un nivel cada vez más alto hasta llegar a su respuesta final. Al ir profundizando en la red, estas funciones más simples se van combinado para buscar relaciones más complejas como por ejemplo, en un modelo de reconocimien-

to facial, partes concretas de la cara (ojos, boca, nariz). En un siguiente paso se identificaría la cara completa, y por último se identificaría a la persona a la que corresponde esa cara [21].

El Deep Learning utiliza una combinación de redes neuronales de múltiples capas, formación con uso intensivo de procesamiento y datos, inspirada en la comprensión del comportamiento del cerebro humano. Este enfoque se ha vuelto tan eficaz que incluso ha comenzado a superar las capacidades humanas en muchas áreas, como el reconocimiento de imágenes y del habla y el procesamiento del lenguaje natural [25].

Los métodos tradicionales de Machine Learning requieren cierto nivel de interacción humana para preprocesar los datos antes de aplicar los algoritmos, mientras que los algoritmos de Deep Learning van un paso más allá, al crear modelos jerárquicos que deben reflejar los procesos de pensamiento de nuestro propio cerebro. Usan una red neuronal multicapa que no requiere preprocesamiento de los datos de entrada para producir un resultado. Los científicos de datos brindan los datos sin procesar al algoritmo, el sistema analiza los datos según lo que ya sabe, y lo que puede inferir de los nuevos datos, y hace la predicción [26].

Los modelos de Deep Learning procesan grandes cantidades de datos y en general son no supervisados o semisupervisados. El Deep Learning puede aprovechar los conjuntos de datos etiquetados (aprendizaje supervisado), para informar a su algoritmo, pero no requiere necesariamente un conjunto de datos etiquetado [6].

Algunos de los tipos de redes neuronales más utilizados son [27]:

- *Redes Neuronales Prealimentadas (FNN):* Son una de las formas más antiguas de redes neuronales, y en éstas los datos fluyen en una dirección a través de capas de neuronas hasta que se obtiene el resultado. En la actualidad, la mayoría de las FNN se consideran "prealimentadas profundas", constando de varias capas (y más de una capa "oculta").

- *Redes Neuronales Recurrentes (RNN):* Difieren de las redes neuronales prealimentadas en que suelen usar datos de series temporales o datos que involucran secuencias. A diferencia de las redes neuronales prealimentadas, que usan ponderaciones en cada nodo de la red, las redes neuronales recurrentes tienen "memoria" de lo que sucedió en la capa anterior como contingente a la salida de la capa actual.

- *Redes de Memoria Prolongada de Corto Plazo (LSTM):* Son una forma avanzada de RNN que puede usar memoria para "recordar" lo que sucedió en capas anteriores. La diferencia entre las RNN y las LTSM es que pueden recordar lo que sucedió hace varias capas mediante el uso de "celdas de memoria".

- *Redes Neuronales Convolucionales (CNN):* Esta clase incluye algunas de las redes neuronales más comunes en la IA moderna. Las CNN suelen usarse en el reconocimiento de imágenes y emplean varias capas distintas (una capa convolucional y, luego, una capa de agrupación o *Pooling*) que filtran diferentes partes de una imagen antes de volver a unirla (en la capa completamente conectada o *Fully Connected*). Es posible que las capas convolucionales anteriores busquen características simples de una imagen, como colores y bordes, antes de buscar características más complejas en capas adicionales.

- *Redes Generativas Adversarias (GAN):* En este modelo se usan dos redes neuronales que compiten entre sí en un juego que, en última instancia, mejora la exactitud del resultado. Una red (el *generador*) crea ejemplos que la otra red (el *discriminante*) juzga como verdaderos o falsos.

2.1.2. Computación de Altas Prestaciones en GPU

Computación de Altas Prestaciones

La Computación de Altas Prestaciones (High Performance Computing o HPC) es la capacidad de procesar conjuntos de datos masivos y realizar cálculos complejos a velocidades extremadamente altas. Para ponerlo en perspectiva, un equipo portátil o de sobremesa con un procesador de 3 GHz puede realizar unos 3000 millones de cálculos por segundo. Aunque esto es mucho más rápido de lo que puede lograr cualquier humano, palidece en comparación con las soluciones HPC que pueden realizar cuatrillones de cálculos por segundo [28].

Uno de los tipos de soluciones HPC más conocidos es el *superordenador*, el cual fue durante décadas el paradigma del sistema HPC. Un superordenador contiene miles de elementos de procesamiento o *nodos de computación* que trabajan juntos para completar una o varias tareas. Esto se conoce como procesamiento paralelo, y es similar a disponer de miles de equipos conectados en red, combinando su potencia computacional para poder completar tareas más rápido.

Mientras que los sistemas de computación estándar resuelven problemas principalmente mediante computación en serie (dividen la carga de trabajo en una secuencia de tareas y luego ejecutan las tareas una tras otra en el mismo procesador), los sistemas HPC emplean [29]:

- *Computación masiva en paralelo:* La computación paralela consiste en el uso de varios procesadores trabajando juntos para resolver una tarea. Así, un problema es dividido en partes, cada una de las cuales es ejecutada en paralelo en diferentes procesadores [30]. En teoría, al tener n procesadores se debería tener n veces más poder de cálculo, con lo que el problema podría resolverse en $1/n$ veces del tiempo requerido por la ejecución con un único procesador. Por supuesto, esto se cumpliría bajo condiciones ideales que, como establece la *Ley de Amdahl* [31], no pueden conseguirse en la práctica. La computación masiva en paralelo es computación paralela que utiliza decenas de miles e incluso millones de procesadores o núcleos de procesador.

- *Clústeres HPC:* En esta arquitectura, los servidores de computación se conectan en un *clúster*, y los programas y algoritmos se ejecutan simultáneamente en los servidores del clúster. El clúster HPC consta de cientos o miles de servidores de computación (denominados *nodos*) conectados en red entre sí, con un planificador centralizado que gestiona la carga de trabajo de computación paralela.

- *Componentes de alto rendimiento:* Todos los demás recursos informáticos (sistemas de archivos, redes, memoria y almacenamiento) deben ser componentes de alta velocidad, alto rendimiento y baja latencia que puedan estar a la altura de los nodos y optimizar el rendimiento del clúster. El componente de almacenamiento debe poder suministrar y recibir datos hacia y desde los nodos tan rápidamente como se procesa. Del mismo modo, los componentes de red deben ser capaces de admitir el transporte de datos de alta velocidad entre los nodos y el almacenamiento de datos. Si un componente no puede mantener el ritmo del resto, el rendimiento de toda la infraestructura HPC se resiente, ocasionándose un cuello de botella.

Uno de los sistemas más empleados en la clasificación de las arquitecturas paralelas es la llamada *Taxonomía de Flynn*. Esta clasificación se basa en el análisis del flujo de instrucciones y de datos, los cuales pueden ser simples o múltiples. Un flujo de instrucción es una secuencia de instrucciones transmitidas desde una unidad de control a uno o más

procesadores. Un flujo de datos es una secuencia de datos que viene desde un área de memoria a un procesador y viceversa. Existen cuatro categorías, las cuales son [32]:

- *SISD (Single Instruction Stream, Single Data Stream):* En esta arquitectura, se ejecuta un único programa usando solamente un conjunto de datos específicos a él. Está compuesta de una memoria central donde se guardan los datos y los programas y de un procesador. En esta plataforma sólo se puede dar un tipo de paralelismo virtual a través del paradigma de multitareas, en el cual el tiempo del procesador es compartido entre diferentes programas. Un ejemplo de esta arquitectura es un computador monoprocesador con arquitectura Von-Neumann.

- *SIMD (Single Instruction Stream, Multiple Data Stream):* Esta arquitectura consiste en un conjunto de elementos de procesamiento que ejecutan la misma instrucción al mismo tiempo. El enfoque de paralelismo aquí usado se denomina paralelismo de datos. En estas arquitecturas se envía la misma secuencia de instrucciones a ejecutar a los distintos procesadores, los cuales procesarán sus datos empleando dichas instrucciones. Un ejemplo de máquina sería el computador CM-2 (Connection Machine – 2), del MIT.

- *MISD (Multiple Instruction Stream, Single Data Stream):* En esta arquitectura, cada elemento de procesamiento ejecuta una tarea diferente, de tal forma que todos los datos a procesar deben ser pasados a través de cada elemento de procesamiento. Se considera que esta arquitectura sólo existe a nivel teórico, aunque algunas arquitecturas específicas como las llamadas pipeline podrían clasificarse como MISD.

- *MIMD (Multiple Instruction Stream, Multiple Data Stream):* Es el modelo más general de paralelismo. Las dos ideas principales son que múltiples tareas heterogéneas pueden ser ejecutadas al mismo tiempo y que cada procesador opera independientemente con ocasionales sincronizaciones con otros. Puede ser explotado tanto el paralelismo funcional (se descompone el problema en funciones que pueden ser ejecutadas por distintos elementos de procesamiento), como el paralelismo de datos (el conjunto de datos se descompone en subconjuntos de datos, procesado cada uno de ellos por un elemento de procesamiento). La forma de programación usualmente utilizada es del tipo concurrente. Muchos multiprocesadores y sistemas de múltiples computadores pertenecen a esta clasificación.

Centrándose en la arquitectura MIMD, una posible clasificación sería teniendo en cuenta la organización de la memoria, existiendo dos categorías: *arquitecturas de memoria distribuida* y *arquitecturas de memoria compartida*.

En las arquitecturas de memoria distribuida, cada nodo de procesamiento está formado por un procesador independiente y su memoria local. Estos nodos están conectados entre sí a través de una red de interconexión. El rendimiento de estas arquitecturas está muy ligado al rendimiento de las comunicaciones entre nodos. Al no existir memoria compartida por los nodos, la comunicación más habitual entre éstos se realizará mediante paso de mensajes (es decir, operaciones explícitas de envío y recepción de mensajes). La librería de paso de mensajes más conocida y empleada es MPI (Message Passing Interface).

Por otra parte, en las arquitecturas de memoria compartida, el conjunto de elementos de procesamiento comparte el mismo espacio de memoria. Debido a esto, todas las comunicaciones entre los nodos se realizan a través de la memoria. La correcta gestión de la memoria es un aspecto clave a la hora de desarrollar un código que funcione correcta y eficientemente.

Un concepto clave en la programación en arquitecturas de memoria compartida es el de *exclusión mutua*. Debido a que todos los nodos tendrán acceso a la misma memoria resultará imprescindible controlar los casos en los que dos o más procesos puedan acceder al mismo tiempo a una misma posición de memoria y se puedan ocasionar situaciones de inconsistencia. La API más conocida y empleada en la implementación de programas que exploten el paralelismo en arquitecturas de memoria compartida es OpenMP, que emplea un modelo *fork-join*.

Recientemente, ha surgido un nuevo tipo de paradigma HPC que supone una solución rentable para empresas que no desean invertir en su propia infraestructura: la *HPC como servicio*. Con la HPC como servicio, las empresas de tecnología hospedan soluciones de HPC en sus infraestructuras y permiten a otras empresas el acceso a ellas a través de la nube. Estas empresas cuentan entonces con los recursos que necesitan y la capacidad de escalar verticalmente con rapidez si lo requieren, mientras solo pagan por la capacidad que utilizan [33].

Las principales aplicaciones de la Computación de Altas Prestaciones son [28, 29]:

- *Inteligencia Artificial y Machine Learning:* Los sistemas HPC pueden utilizarse en aplicaciones de ML altamente avanzadas donde se analizan grandes cantidades de datos.
- *Análisis de grandes conjuntos de datos:* Se recurre a la comparación rápida y a la correlación de grandes conjuntos de datos para complementar investigaciones y resolver problemas académicos, científicos, financieros, comerciales, gubernamentales, de salud y de seguridad cibernética. Este trabajo requiere un rendimiento masivo y capacidades de cómputo de una enorme potencia.
- *Modelado avanzado y simulación:* Al no tener que realizar un montaje físico en las primeras etapas del proceso, el modelado avanzado y la simulación permiten que las empresas ahorren tiempo, materiales y costos de contratación de personal para lanzar sus productos al mercado con mayor rapidez.

Por otra parte, algunos casos de uso son [34]:

- *Laboratorios de investigación:* Se utiliza para ayudar a los científicos a encontrar fuentes de energía renovable, comprender la evolución de nuestro universo, predecir y rastrear tormentas y crear nuevos materiales.
- *Ciencias de la salud y atención sanitaria:* El primer intento de secuenciar un genoma humano duró 13 años; hoy en día, los sistemas HPC pueden lograr esta tarea en menos de un día. Otras aplicaciones de HPC en atención sanitaria y ciencias de la vida incluyen el descubrimiento y el diseño de medicamentos, el diagnóstico rápido de cáncer y la creación de modelos moleculares.
- *Medios de comunicación y entretenimiento:* Empleado para editar largometrajes, producir efectos especiales y transmitir eventos en directo en todo el mundo.
- *Petróleo y gas:* Utilizado para identificar con mayor precisión dónde se perforan nuevos pozos y para ayudar a impulsar la producción de pozos existentes.
- *Servicios financieros:* Se emplea para realizar un seguimiento de las tendencias de las acciones en tiempo real y automatizar las operaciones comerciales, así como para la detección de fraudes, para lo cual se deben analizar millones de transacciones al tiempo en que se van produciendo. También es el motor de ciertas aplicaciones en la simulación de Monte Carlo y otros métodos de análisis de riesgos.

- *Meteorología:* En este sector se pueden encontrar dos casos de uso creciente de HPC: los pronósticos meteorológicos y la creación de modelos climáticos, que implican el proceso de ingentes cantidades de datos meteorológicos históricos y millones de cambios a diario en puntos de datos relacionados con el clima.

- *Energías renovables:* Se pueden emplear en procesamiento de datos sísmicos, creación de modelos y simulación de embalses, analítica geoespacial, simulación de vientos y cartografía.

- *Ventas y marketing:* Analizar cantidades masivas de datos de clientes para proporcionar recomendaciones de productos más específicas y mejor servicio al cliente.

Aceleración por hardware

El modelo de Computación de Altas Prestaciones del que se habló en el apartado anterior va enfocado al uso de CPU, lo que limita su potencial, pues pese a que se cuente con un gran número de elementos de procesamiento, no dejarán de ser CPU, y su rendimiento estará limitado, entre otras cosas, a nivel físico, pues cada vez es más difícil diseñar CPU con mayor rendimiento, debido a que el procesador no deja de ser un circuito integrado compuesto principalmente por transistores, por lo que cada vez es necesario miniaturizar más todavía los transistores que componen sus circuitos, y cada vez ser requieren tecnologías de fabricación más complejas.

Existe, en cambio, otro tipo de elementos de procesamiento que permiten realizar ciertas tareas de computación más eficientemente de lo que es posible usando software ejecutándose en una CPU de propósito general. Esto se denomina *aceleración por hardware*.

La aceleración por hardware combina la flexibilidad de los procesadores de propósito general, como las CPU, con la eficiencia del hardware especializado (conocido como acelerador por hardware), como las GPU (Graphics Processing Unit) y los ASIC (Application-Specific Integrated Circuit), incrementando la eficiencia en varios órdenes de magnitud cuando cualquier aplicación se implementa más arriba en la jerarquía de los sistemas de cómputo.

Por ejemplo, los procesos de visualización pueden ser delegados a una tarjeta gráfica para así conseguir una reproducción más rápida y de mayor calidad de vídeos y juegos, mientras que al mismo tiempo la CPU queda libre para ejecutar otras tareas. Esta aceleración es tan importante porque permite que los procesos complejos se ejecuten en paralelo y a una mayor velocidad, lo que a su vez puede mejorar significativamente el rendimiento [35].

Los tipos de aceleradores hardware más comunes son:

- *Graphics Processing Unit (GPU):* Fueron originalmente diseñadas para acelerar el renderizado de gráficos, pero ahora se emplean en cálculos que involucran enormes cantidades de datos, acelerando una parte de la aplicación mientras el resto continúa ejecutándose en la CPU. La paralelización masiva de las GPU modernas permite el procesado de miles de millones de registros al instante. El siguiente apartado de este trabajo profundizará en este tipo de acelerador hardware.

- *Field Programmable Gate Array (FPGA):* Un circuito integrado especificado por un Lenguaje de Descripción de Hardware (HDL) diseñado para permitir que el usuario configure una gran parte de la funcionalidad electrónica. Un FPGA se puede utilizar para acelerar partes de un algoritmo, compartiendo parte del cómputo entre éste y un procesador de propósito general.

- *Application-Specific Integrated Circuit (ASIC):* Un circuito integrado diseñado y customizado específicamente para un propósito o aplicación concreta, mejorando la velocidad al estar especializado únicamente en la ejecución de su única función. Los ASIC modernos tienen una complejidad superior a los 100 millones de puertas lógicas.

Algunas de las principales aplicaciones de la aceleración por hardware son:

- Renderizado de gráficos mediante una GPU.
- Procesamiento de señales digitales mediante un procesador de señales digitales.
- Procesamiento de señal analógicas mediante un FPGA.
- Procesamiento de sonido mediante una tarjeta de sonido.
- Criptografía mediante un acelerador criptográfico.
- Inteligencia Artificial mediante un acelerador de IA.
- Todo tipo de tareas realizadas por FPGA, ASIC, Complex Programmable Logic Devices (CPLD), and Systems-on-Chip (SoC).

GPU

Una Unidad de Procesamiento Gráfico (más conocida como GPU, siglas de *Graphics Processing Unit*) es un coprocesador (microprocesador utilizado como suplemento de las funciones del procesador principal, es decir, la CPU) dedicado al procesamiento de gráficos u operaciones de coma flotante. Las GPU se utilizan para procesar y renderizar la información visual que se muestra en la pantalla de un dispositivo. Estas tareas incluyen el dibujo de texturas, la iluminación, la sombra y otros efectos gráficos en juegos y aplicaciones gráficas intensivas. Además, las GPU también se utilizan en aplicaciones de realidad virtual y aumentada, sistemas de Inteligencia Artificial y Deep Learning, y en algunos sistemas de supercomputación [36].

Los primeros computadores se basaban en interfaces de texto, por lo que la CPU era la encargada de gestionar toda la información. Sin embargo, con la introducción de elementos gráficos, la exigencia del Sistema Operativo comenzó a crecer, ya que el procesador central se sobrecargaba, reduciéndose la eficiencia del computador. En ese momento comenzaron a emplearse los coprocesadores matemáticos (*Floating Point Unit* o FPU) con el objetivo de tratar la información gráfica, ya que éstos eran capaces, a diferencia de la unidad central, de procesar datos en coma flotante, modo en que las operaciones gráficas podían ser planteadas.

Posteriormente, la FPU fue evolucionando hasta que en los años 80 aparecieron pequeños chips de control, considerados los verdaderos precursores de las tarjetas gráficas, y encargados de funciones básicas y sencillas para el procesamiento de imágenes. No fue hasta 1999 que la marca NVIDIA decidió acuñar el concepto Graphics Processing Unit (GPU) para referirse a este componente hardware que, con el paso de los años, ha ido perfeccionándose y evolucionando hasta alcanzar la complicada estructura que es hoy en día y llegar a poder tratar operaciones tridimensionales [37].

Una GPU puede ser *integrada*, si se encuentra en el procesador y comparte la memoria del sistema con la CPU, o *dedicada*, si tiene su propia memoria y es independiente de la CPU (encontrándose en una tarjeta gráfica). Las GPU integradas son adecuadas para tareas gráficas simples y ahorran energía, mientras que las dedicadas son más potentes

y más indicadas para aplicaciones gráfica intensivas. También existen las GPU híbridas, que comparten la memoria del sistema, aunque también tienen una limitada memoria propia para desarrollar operaciones inmediatas y acelerar el rendimiento del PC.

Una GPU no es una tarjeta gráfica en sí misma, sino su núcleo. Una tarjeta gráfica además de integrar una GPU en su haber, necesita un PCB, resistencias, SMD, capacitores, controladores de voltaje, reguladores de fase, VRAM, salidas de pantalla y más. Normalmente tienen un tamaño bastante superior al del *die* de una CPU y en la actualidad es muy complicado que bajen de los 100 mm^2 incluso en los procesos litográficos más avanzados. Incluye un sustrato de un tamaño que normalmente ronda entre el triple y el cuádruple del área del chip, principalmente por la cantidad de elementos a conectar con ella [38].

Una GPU se basa en un chip de silicio que alberga una serie de millones de transistores bajo una arquitectura específica y con unas prestaciones dedicadas para cada modelo en concreto. Independientemente de cual sea la marca y generación de una GPU, estas se componen de los mismos elementos en mayor o menor número, entre los que se encuentran los siguientes [39]:

- *Unidades Streaming Multiprocessor (SM) o Compute Units:* NVIDIA utiliza las primeras, AMD las segundas. Cada unidad integra un conjunto de motores de sombreado, unidades de textura, unidades de rasterizado y, en función de la arquitectura, núcleos especializados para *Ray Tracing* e IA. Son la base de cualquier GPU.
- *Procesador de comandos:* Lee la lista de pantalla o de instrucciones que le envía la CPU, tanto para generar gráficos como para hacer cálculos complejos.
- *Motores de sombreado:* Un núcleo capaz de ejecutar programas pensados para manipular primitivas gráficas a tiempo real. Se ocupa de realizar la carga de trabajo asociada a tareas como la transformación de la geometría, tanto a nivel de color como de sombreado y de otros efectos (iluminación, neblina, reflejos...).
- *Unidad de Rasterizado:* Realiza el procesamiento necesario para que una imagen, que ha sido expresada en formato vectorial, sea convertida en un conjunto de píxeles perfectamente ordenados que se escriben en el *framebuffer*, desde el cual serán transmitidos de forma directa a la pantalla.
- *Raster Output:* Unidad encargada de dibujar los píxeles finales sobre el buffer de imagen. Es junto a la cache de último nivel la única pieza que tiene permisos de escritura sobre la VRAM.
- *Unidad de Texturizado:* Se encarga de aplicar un mapeado de texturas a la geometría, es decir, aplicar una imagen sobre una superficie para simular textura o color en la vida real.
- *Unidad de Teselación:* Unidad que subdivide los vértices de los objetos para darles un aspecto más redondeado y pulido.
- *Unidad de Intersección:* Calcula la intersección de los rayos de la escena con los objetos. Esencial para el *Ray Tracing*.
- *RT Core:* Se dedica a acelerar el procesado de la carga de trabajo relacionada con el *Ray Tracing*. Estos núcleos fueron introducidos en 2018.
- *Tensor cores:* Se utilizan en las tarjetas gráficas NVIDIA basadas en la arquitectura Volta, Turing y Ampere. Están especializados en cargas de trabajo centradas en inferencia, inteligencia artificial y aprendizaje profundo. En gaming, se ocupan de la carga de trabajo que representa aplicar el DLSS 2.0.

- *CODEC de Vídeo:* Procesador independiente que decodifica vídeo en varios formatos multimedia y los reproduce, así como se encarga de generar vídeo e incluso pasar de un formato a otro.

- *Interfaz de memoria:* Permite a la GPU leer de su memoria RAM, conocida como VRAM.

- *DMA (Direct Memory Access):* Unidad que le permite a la GPU leer de la RAM principal del sistema.

La GPU proporciona un rendimiento de instrucciones y un ancho de banda de memoria muy superiores respecto de una CPU con un precio y consumo energético similares. Otros dispositivos como los FPGA, son también muy eficientes energéticamente, pero ofrecen una flexibilidad de programación mucho menor a la de las GPU.

La diferencia de capacidades entre GPU y CPU se debe a que ambos han sido diseñados con distintos objetivos en mente. Mientras que una CPU es diseñada para destacar en la ejecución de una secuencia de operaciones (conocida como *thread*) lo más rápido posible, y es capaz de ejecutar unas pocas decenas de threads en paralelo, la GPU es diseñada para la ejecución de miles de threads en paralelo (amortizando el inferior rendimiento para un thread individual y consiguiendo un rendimiento mucho mayor).

Para poner las cosas en contexto, un procesador actual de alto rendimiento puede contar, por ejemplo, con 8 núcleos (elementos de procesamiento), mientras que una GPU correspondiente al mismo momento actual aproximadamente, como por ejemplo, la RTX 3080 de NVIDIA cuenta con 8704 *CUDA Cores*, es decir, tiene miles de pequeños núcleos, lo que permite paralelizar cargas de trabajo grandes y complejas, y sacarlas adelante de una forma más eficiente [38, 40].

La GPU está especializada en cálculos altamente paralelizados, por lo que se emplean más transistores en el procesamiento de datos, en vez de en la captura de datos y el control de flujo. La Figura 2.2 ilustra un ejemplo de la distribución de recursos en una CPU y en una GPU.

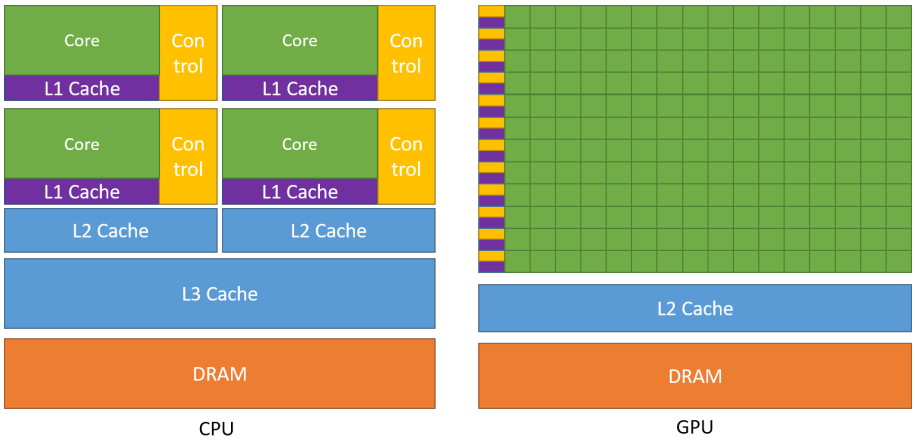

Figura 2.2: Gráfico comparativo de las arquitecturas de CPU y GPU.

Emplear más transistores para el procesamiento de datos (por ejemplo, operaciones de coma flotante), es bueno para los cálculos altamente paralelizados, pues la GPU puede compensar las latencias de acceso a memoria con operaciones de cómputo, en vez de basarse en grandes caches de datos y un complejo control de flujo (siendo ambos muy costosos en cuanto a transistores) para evitar grandes latencias de acceso a memoria.

Esta comparación permite entender por qué una GPU tiene un potencial tan elevado en operaciones de coma flotante, y por qué son capaces de ofrecer un alto nivel de rendimiento en otros sectores que van más allá del diseño gráfico, del renderizado y del gaming, como por ejemplo la investigación científica, la inferencia, el aprendizaje profundo y el análisis de datos. Se trata de sectores que, por la propia naturaleza de las cargas de trabajo que representan, se resuelven mucho mejor al ser paralelizadas en miles de núcleos pequeños.

Por regla general, las aplicaciones tienen una combinación de partes paralelizadas y partes secuenciales, por lo que los sistemas se diseñan con una combinación de GPU y CPU con el fin de maximizar el rendimiento global. Las aplicaciones que cuenten con un alto grado de paralelismo pueden explotar la naturaleza paralela de la GPU para obtener un mayor rendimiento respecto de la CPU. La computación acelerada por GPU permite asignar a la GPU el trabajo de los aspectos de la aplicación donde la computación es más intensiva, mientras que el resto del código se ejecuta en la CPU. Desde la perspectiva del usuario, las aplicaciones se ejecutan de forma mucho más rápida [40].

Programación de GPU: CUDA

CUDA es una plataforma de computación paralela de propósito general y un modelo de programación que permite aprovechar la potencia de cómputo paralelo de las GPU de NVIDIA para resolver problemas complejos más eficientemente que en una CPU [40].

CUDA fue lanzado en noviembre de 2006 por NVIDIA y es un entorno que permite a los desarrolladores trabajar con C++ como lenguaje de alto nivel. También existen otras APIs como pueden ser FORTRAN, DirectCompute y OpenACC.

La Figura 2.3 muestra algunas de las librerías y APIs de computación en GPU más importantes, así como los distintos tipos de arquitecturas de las GPU de NVIDIA.

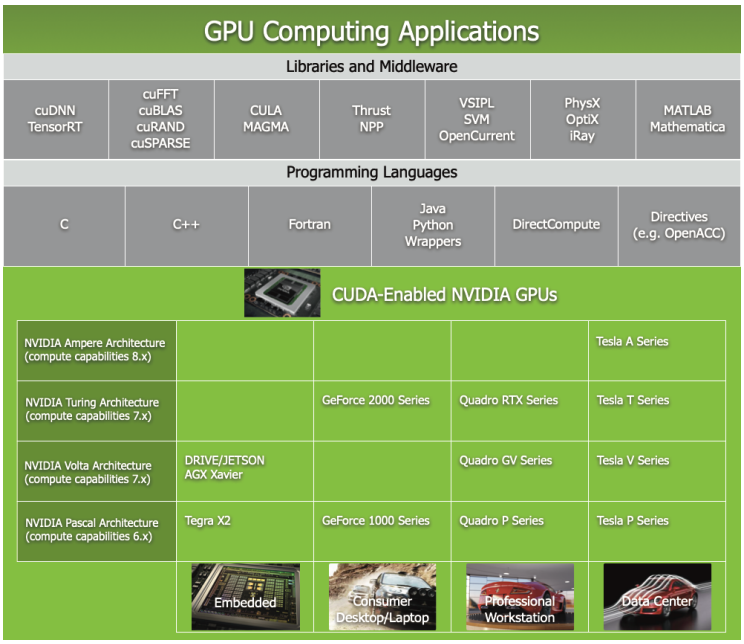

Figura 2.3: Aplicaciones de la computación en GPU.

El modelo de programación paralela de CUDA está diseñado para poder cumplir el objetivo de desarrollar software de aplicación que pueda escalar de forma transparente su paralelismo para poder aprovechar el creciente número de elementos de procesamiento

al mismo tiempo que se mantiene una baja curva de aprendizaje para desarrolladores que estén familiarizados con lenguajes de programación estándar como C.

Presenta tres abstracciones clave: una jerarquía de grupos de threads, memorias compartidas y sincronización mediante barreras, que son proporcionadas al programador como unas simples extensiones del lenguaje.

Estas abstracciones permiten implementar un paralelismo de datos de *grano fino* (aquel en el cual las tareas que se asignan a cada elemento de procesamiento son pequeñas, y las comunicaciones entre éstos son frecuentes) y un paralelismo de threads anidados dentro de un paralelismo de datos de *grano grueso* (aquel en el cual las tareas asignadas son grandes y los elementos de procesamiento apenas se comunican entre ellos) y un paralelismo de tareas.

Dichas abstracciones guían al programador para que divida el problema en subproblemas de grano grueso que puedan ser resueltos de forma independiente en paralelo por bloques de threads, y cada subproblema en partes más "finas" que puedan ser resueltas de forma cooperativa en paralelo por los distintos threads de un mismo bloque.

Esta descomposición permite que los threads cooperen en la resolución de cada subproblema, al mismo tiempo que permite una escalabilidad automática, ya que cada bloque de threads puede ser asignado a cualquiera de los *Streaming Multiprocessors* (SM) de la GPU que estén disponibles, en cualquier orden sea concurrente o secuencial. Gracias a ello, un programa de CUDA puede ser ejecutado en cualquier número de SM y tan sólo el sistema *runtime* necesita saber de cuántos SM se dispone (Figura 2.4).

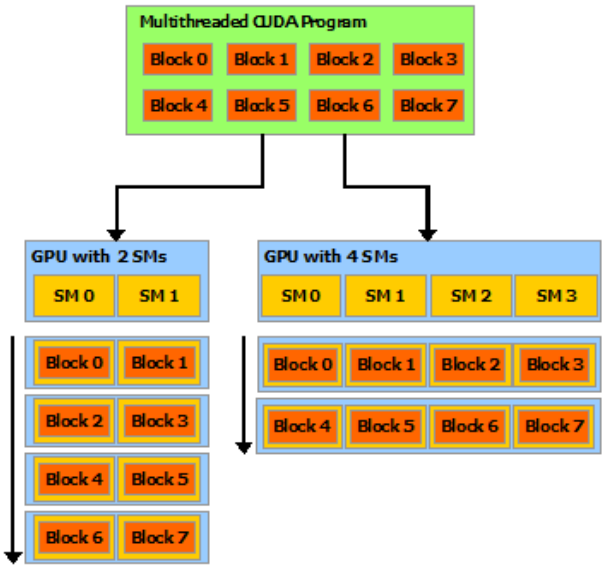

Figura 2.4: Ejemplo de la escalabilidad automática que el modelo de CUDA ofrece.

En lo que respecta al modelo de programación, es crucial destacar el concepto de *kernel*. CUDA C++ extiende C++ al permitir al programador definir funciones denominadas *kernels*, las cuales al ser invocadas serán ejecutadas N veces en paralelo por N diferentes threads, en vez de una única vez como las funciones típicas de C++.

El número de threads que ejecutarán un *kernel* se especifica en la llamada al éste, pudiendo determinar tanto el número de bloques a lanzar como el número de threads a lanzar por cada bloque. Cada thread cuenta con una ID de thread única, accesible mediante la variable *threadIdx*.

Por conveniencia, *threadIdx* es un vector de 3 componentes, por lo que los threads pueden ser identificados mediante un índice de 1, 2 ó 3 dimensiones, formando un bloque de threads que también puede ser unidimensional, bidimensional o tridimensional. Esto permite poder realizar cálculos sobre elementos vectoriales, matriciales, tensoriales... de una forma más natural.

Existe un límite máximo en el número de threads por bloque, debido a que todos los threads de un mismo bloque deben residir en el mismo SM y compartir los mismos recursos de memoria. En las GPU actuales, un bloque puede contener un máximo de 1024 threads.

Pese a esto, un *kernel* puede ser ejecutado por múltiples bloques de threads de idénticas dimensiones, por lo que el número total de threads será igual al número de threads por bloque multiplicado por el número de bloques.

Los bloques se organizan en una malla o *grid* que puede ser unidimensional, bidimensional o tridimensional. El número de bloques en un *grid* suele ir dictado por el tamaño de los datos a procesar, y normalmente excede al número de SM del sistema.

Cada bloque en el *grid* puede ser identificado por un índice de 1, 2 ó 3 dimensiones accesible en el *kernel* mediante la variable *blockIdx*. El tamaño del bloque de threads puede ser consultado en el *kernel* mediante la variable *blockDim*.

Los bloques de threads deben poder ser ejecutados de forma independiente y en cualquier orden. Esta independencia permite que los bloques sean asignados a los procesadores en cualquier orden y a cualquier número de procesadores, disponiendo así de un código escalable.

Los threads pertenecientes a un mismo bloque pueden cooperar compartiendo datos mediante una memoria compartida y sincronizando su ejecución para poder coordinar los acceso a memoria. Para ello, CUDA con la función *__syncthreads()*, la cual, ejecutada dentro de un *kernel*, actúa como una barrera en la que todos los threads de un mismo bloque deberán esperar a que todos los demás threads de dicho bloque la hayan alcanzado, pudiendo entonces continuar su ejecución.

Las GPU compatibles con la especificación *Compute Capability* 9.0 o superiores podrán hacer uso de un nivel extra de jerarquía, conocido como *Thread Block Clusters*, los cuales están formados por bloques de threads. Del mismo modo en el que los threads pertenecientes a un mismo bloque serán asignados a un mismo SM, los bloques de un mismo Cluster serán asignados a un mismo GPU Processing Cluster (GPC) de la GPU.

El modelo de programación CUDA se considera *programación heterogénea* debido a que asume que los threads se ejecutan en un dispositivo físicamente separado (conocido como *device*, que se corresponde con la GPU) que opera como coprocesador respecto de un *host* (como se conoce a la CPU) que ejecuta el programa de C++.

En lo que respecta a la arquitectura de memoria, este modelo también asume que tanto el *host* (CPU) como el *device* (GPU) poseen sus propios espacios separados de memoria. La única zona de memoria accesible desde el *host* es la memoria global. Además, tanto la reserva o liberación de memoria global como la transferencia de datos entre el *host* y el *device* debe hacerse de forma explícita desde el *host* mediante llamadas a funciones específicas de CUDA, ya que un *kernel* sólo puede operar sobre la memoria del *device*, por lo que necesitaremos funciones específicas para reservar y liberar la memoria de éste, así como funciones para la transferencia de datos entre la memoria del *host* y del *device* [41].

Existen diferentes espacios de memoria mediante los cuales los threads de CUDA (que se ejecutan en el *device*) pueden acceder a los datos: cada thread cuenta con su memoria local privada, cada bloque de threads posee un espacio de memoria compartida visible a todos los threads de dicho bloque y que posee el mismo tiempo de vida que

el bloque y finalmente existe un espacio de memoria global al cual tienen acceso todos los threads. Cuando es posible trabajar con Clusters de bloques (*Compute Capability* 9.0 o superiores), los bloques pertenecientes a un Cluster pueden realizar lectura, escritura y operaciones atómicas en la memoria compartida de otros.

También se cuenta con dos espacios adicionales de memoria de sólo lectura que son accesibles por todos los threads: la memoria constante y la memoria de texturas. La memoria global, así como la memoria constante y la memoria de texturas son persistentes entre distintas llamadas a *kernels* de una misma aplicación.

La Figura 2.5 muestra los distintos niveles de jerarquía de memoria, representándose mediante flechas las memorias a las cuales pueden acceder tanto la CPU (*host*) como los threads (ejecutados en el *device*), siendo la flecha unidireccional en los casos en los cuales el acceso de de sólo lectura:

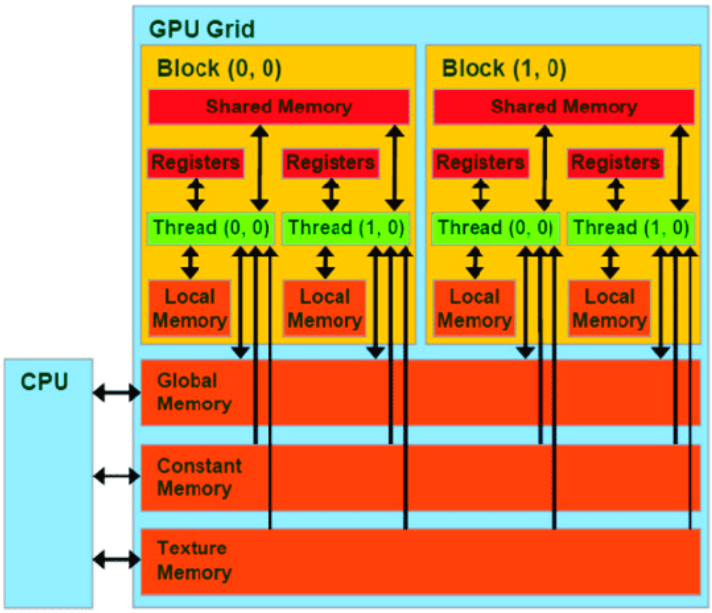

Figura 2.5: Jerarquía de memoria en CUDA.

A continuación se explican las características de cada uno de los tipos de memoria que aparecen en el esquema:

- *Memoria global:* Es la más grande y la más lenta. Puede ser leída y escrita por la CPU (*host*) y por los threads de la GPU (*device*). Permite comunicar datos entre *host* y *device*, así como para comunicarlos entre todos los threads independientemente del bloque al que pertenezcan. El patrón de acceso a memoria por los threads puede afectar el rendimiento. Esta memoria se encuentra ubicada en un chip externo de memoria DRAM. Dado que posee una latencia muy elevada, es una buena práctica copiar los datos que van a ser accedidos frecuentemente a la zona de memoria compartida.

- *Memoria constante:* Es parte de la memoria global. El *host* puede leer y escribir, y es de sólo lectura para el *device*. Ofrece mayor ancho de banda cuando grupos de threads acceden al mismo dato.

- *Memoria de texturas:* Es controlada por el programador y puede beneficiar aplicaciones con localidad espacial donde el acceso a memoria global es un cuello de botella.

- *Memoria compartida:* Es pequeña y muy rápida y es compartida por todos los threads de un bloque. Es de lectura/escritura por los threads. Puede comunicar datos entre threads del mismo bloque. Posee un elevado ancho de banda y baja latencia (similar a una cache de nivel 1, que es el tipo de cache más rápido, y que se utiliza para almacenar y acceder a datos e instrucciones importantes y de uso frecuente). Puede verse afectada por el patrón de acceso de los threads.

- *Registros:* Cada thread utiliza su propio conjunto de registros. El programador no tiene un control explícito de los registros, y son utilizados para la ejecución de programas de la misma forma que los registros de propósito general de una CPU.

- *Memoria local:* Es usada por el compilador automáticamente para alojar variables cuando hace falta.

Tanto los registros, como la memoria compartida y la constante son de acceso rápido y permiten reducir los accesos a la memoria global, sin embargo su tamaño constituye una limitación importante. Cada *device* dispone de una determinada cantidad de memoria, la cual limita el número de threads que pueden ser alojados en los SM. Por lo general, cuanto más espacio de memoria necesite un thread, menos threads podrán ser alojados en un SM. Lo mismo sucede con la memoria compartida, que limita el número de bloques que pueden ser alojados por un SM.

CUDA C++ está formado por una serie de extensiones al lenguaje C++ y una librería *runtime*. Los ficheros fuente que contengan algunas de las extensiones de CUDA C++ deberán ser compilados con el compilador *nvcc*. El *runtime* proporciona las funciones que se ejecutan en el *host* para permitir la reserva y liberación de memoria del *device*, así como la transferencia de datos entre las memorias del *host* y el *device*, o el trabajo con sistemas con múltiples *devices*.

Los *kernels* pueden ser escritos empleando el conjunto de instrucciones de CUDA, conocido como PTX, pero suele ser más recomendable emplear un lenguaje de programación de alto nivel como C++. En ambos casos, los *kernels* deben ser compilados por el compilador *nvcc*.

El compilador *nvcc* compila ficheros que pueden incluir una mezcla de código que debe ser ejecutado en el *host* y código que debe ser ejecutado en el *device*, por lo que el flujo de trabajo de nvcc consiste en separar ambos códigos, compilar el código del *device* a un ensamblador (PTX) y/o una forma binaria (objeto *cubin*) y modificar el código del *host* sustituyendo las expresiones del tipo <<<. . . >>> por llamadas a las funciones necesarias del *runtime* que permitan la carga y ejecución de los *kernels* compilados.

2.2 Hardware y software utilizado

En esta sección se describirán en primer lugar las arquitecturas de las máquinas donde se trabajará, las cuales son dos GPU de NVIDIA pertenecientes a dos líneas de productos diferentes; y a continuación se explicará la herramienta Apache TVM, la cual ha sido empleada para realizar el desarrollo del código.

2.2.1. Descripción del hardware utilizado

En la realización de este Trabajo se han empleado dos máquinas, las cuales cuentan con dos GPU diferentes, una perteneciente a la línea NVIDIA Quadro, especializada en procesamiento gráfico profesional, y otra perteneciente a la línea NVIDIA Tesla, centrada

en la Computación de Altas Prestaciones. Este capítulo se abordará tanto una descripción como una comparación de ambas GPU, así como de sus arquitecturas, Turing y Volta.

Máquina 1: NVIDIA Quadro RTX 5000

La NVIDIA Quadro RTX 5000 es una GPU perteneciente a la línea de productos Quadro, la cual está enfocada a su uso en estaciones de trabajo profesionales de Diseño Asistido por Computadora (CAD), Imagen Generada por Computadora (CGI), Creación de Contenido Digital (DCC), cálculos científicos y Machine Learning.

Las diferencias más destacadas entre las GPU Quadro y las GeForce (que es la versión generalista) son el uso de memoria ECC (que emplea un código de corrección de errores para detectar y corregir la corrupción de datos que puede ocurrir en memoria) y una precisión de coma flotante mejorada. Éstas son propiedades deseables cuando la GPU es empleada para cálculos, los cuales, en contraste con el renderizado de gráficos, requieren precisión y fiabilidad.

La serie Quadro RTX (a la cual pertenece la GPU usada) está basada en la arquitectura Turing, y es la primera en contar con *Ray Tracing* (algoritmo para síntesis de imágenes que calcula el camino de la luz como píxeles en un plano de la imagen y simula sus efectos sobre las superficies virtuales en las que incide, con el fin de producir imágenes con un alto grado de realismo) en tiempo real gracias a sus nuevos cores RT.

La línea Quadro fue descontinuada en 2020 con la llegada de tarjetas que cuentan con la arquitectura Ampere, que pasan a formar parte de la nueva línea de productos NVIDIA RTX.

La NVIDIA Quadro RTX 5000 [42] cuenta con 3072 *CUDA cores* (elementos de procesamiento, o lo que es lo mismo, unidades con la capacidad de operar con números en coma flotante de 32 bits de precisión, y que forman parte de los *Streaming Multiprocessors*), 384 *Tensor cores* (diseñados para ejecutar operaciones con datos agrupados en tensores o matrices, lo cual está especialmente enfocado a la IA del tipo Deep Learning o las Redes Neuronales Convolucionales), 48 *RT Cores* (especializados en el Ray Tracing o trazado de rayos, técnica que permite generar imágenes con un mayor grado de realismo) y 16GB de memoria GDDR6 (Graphics Double Data Rate de sexta generación, que es una memoria del tipo *Double Data Rate*, lo que significa que se puede realizar el doble de transferencias con una misma frecuencia de reloj).

Su interfaz de memoria es de 256 bits y el ancho de banda de ésta es de hasta 448 GB/s. Su arquitectura es NVIDIA Turing, que se destallará más adelante. Posee soporte para *NVIDIA NVLink*, lo cual permite que las aplicaciones escalen memoria y rendimiento empleando configuraciones multi-gpu. Además, su implementación del nuevo *VirtualLink* proporciona conectividad con dispositivos de Realidad Virtual para permitir que los diseñadores puedan visualizar su trabajo en entornos virtuales de última generación. Su rendimiento en simple precisión (32 bits) es de 11.2 TFLOPS, mientras que su rendimiento tensorial es de 89.2 TFLOPS.

Máquina 2: NVIDIA Tesla V100

La NVIDIA Tesla V100 pertenece a la línea de productos NVIDIA Tesla, la cual está enfocada al GPGPU (General-Purpose Computing on Graphics Processing Units, es decir, el uso de GPU, que normalmente se empleaban únicamente en el procesamiento gráfico, para realizar cómputos en aplicaciones tradicionalmente realizadas por el CPU), centrada en los ámbitos de la Computación de Altas Prestaciones (HPC), el Deep Learning y la Inteligencia Artificial.

Este tipo de GPU cuenta con configuraciones que ofrecen una mayor potencia trabajando con cargas que son frecuentes en el ámbito científico, y también inferencia e IA. Entre otras cosas, estas GPU ofrecen una mayor potencia en FP64 (doble precisión), cuentan con mayores cantidades de memoria dedicada e incrorporan (de igual manera que las Quadro) tecnología ECC (corrección de errores). Todas estas características son clave para la realización de grandes simulaciones y tareas comparativas que tengan una validez plena. Ofrecen, además, una elevada escalabilidad, y a diferencia de las Quadro no poseen salida de vídeo. Es, por tanto, una familia especialmente enfocada a entornos científicos.

La marca NVIDIA Tesla fue retirada en 2020, posiblemente debido a posibles conflictos con la marca automovilística, y fue sustituida por NVIDIA Data Center, llegando las primeras GPU de esta línea con arquitectura Ampere.

La NVIDIA Tesla V100 [43] cuenta con 5120 *CUDA cores* y 640 *Tensor cores* y 32GB de memoria HBM2, que es la segunda generación de la *High Bandwith Memory*, la cual se caracteriza por tener una estructura vertical en la que se amontonan o apilan capas en 3D, ahorrando de esta manera espacio en el PCB y dinero en los chips. Esta memoria presenta una mayor capacidad por pila y un ancho de banda muy elevado, lo cual supone un mayor rendimiento que la del tipo GDDR6, siendo esta última, sin embargo, mucho más usada en las GPU debido a la gran diferencia en el precio. El ancho de banda de la memoria es de 900 GB/s, y la eficiencia de utilización de la DRAM es del 95 %.

A diferencia de la NVIDIA Quadro RTX 5000, no posee *RT Cores*, debido a que está enfocada al cómputo científico y no al procesamiento gráfico. Su arquitectura es NVIDIA Volta, que se destallará más adelante. La arquitectura de la Tesla V100 fue diseñada para simplificar la programabilidad, ya que su nuevo scheduling de threads independientes permite una sincronización de grano fino y mejora la utilización de la GPU mediante la compartición de recursos entre pequeñas tareas. Posee soporte para *NVIDIA NVLink* de segunda generación, lo que permite obtener un mayor ancho de banda, más conexiones y una escalabilidad mejorada para configuraciones multi-GPU y multi-GPU/CPU, soportando hasta 6 conexiones *NVLink* con un ancho de banda total de 300 GB/s.

Esta GPU incorpora un modo de *Máximo Rendimiento*, que la permite operar hasta su TDP (Potencia de Diseño Térmico) de 300 W para acelerar aplicaciones que requieran la mayor velocidad de cómputo, y un modo de *Máxima Eficiencia*, que permite ajustar el consumo energético para operar con un rendimiento por vatio optimizado. Puede establecerse un límite de consumo para todos los GPU de un rack, reduciendo dramáticamente el gasto energético obteniendo aún así un excelente rendimiento del rack.

Según datos proporcionados por NVIDIA [43], posee un rendimiento 47 veces superior al de un CPU (en la comparación se utiliza el Xeon E5-2690v4 @ 2.6GHz) en inferencia de Deep Learning, mientras que si se compara con la arquitectura inmediatamente anterior de NVIDIA (NVIDIA Pascal), se obtiene un rendimiento 12 veces superior en entrenamiento y 6 veces superior en inferencia. Su rendimiento en simple precisión (32 bits) es de 14 TFLOPS, su rendimiento en doble precisión (64 bits) es de 7 TFLOPS y su rendimiento tensorial es de 112 TFLOPS.

Descripción de las arquitecturas

La arquitectura NVIDIA Volta fue lanzada en 2017 como sucesora de la anterior arquitectura Pascal, incorporando todos los avances de ésta y mejorando de forma significativa el rendimiento y la escalabilidad, añadiendo además muchas nuevas funcionalidades que mejoran la programabilidad. Estos avances permiten potenciar el rendimiento en sectores como HPC, data centers, superordenadores y sistemas de Deep Learning [44].

La arquitectura de sus *Streaming Multiprocessors* (SM) está optimizada para Deep Learning. El SM de Volta es un 50 % más eficiente energéticamente que el diseño inmediatamente anterior (Pascal), permitiendo grandes incrementos en el rendimiento FP32 (simple precisión) y FP64 (doble precisión) con un mismo consumo energético. La arquitectura Volta fue la primera en implementar los *Tensor Cores*, especialmente diseñados para tener un rendimiento de Deep Learning superior al de los *CUDA cores* tradicionales (con un pico de rendimiento 12 veces mejor en entrenamiento y 6 veces mejor en inferencia).

Sus rutas de datos paralelos enteros y de coma flotante independientes le permiten ser mucho más eficiente en cargas de trabajo que presentan una mezcla de cómputos y cálculos de direcciones. También incorpora entre otras novedades el scheduling de threads independientes (permitiendo una sincronización de grano fino). Además, su cache de datos L1 y unidad de memoria compartida combinadas permiten incrementar el rendimiento al mismo tiempo que facilitan la programabilidad.

Numerosos frameworks de Deep Learning, como *Caffe2*, *MXNet*, *CNTK* o *TensorFlow* han sido optimizados para aprovechar el rendimiento de la arquitectura Volta así obtener tiempos de entrenamiento mucho más rápidos y un rendimiento de entrenamiento multinodo mayor. Librerías como *cuDNN*, *cuBLAS* y *TensorRT* también han sido optimizadas para Volta, obteniendo un mayor rendimiento tanto en inferencia de Deep Learning como en aplicaciones HPC.

La GPU está compuesta por 6 GPC (GPU Processing Clusters), los cuales cuentan con 7 TPC (Texture Processing Clusters) cada uno (incluyendo cada uno dos SM) y 14 *Streaming Multiprocessors* (SM). Esta arquitectura consta de 84 SM Volta en total (de los cuales la GPU usada, Tesla V100, emplea 80), los cuales están formados por 64 cores FP32, 64 cores INT32, 32 cores FP64, 8 *Tensor Cores* y 4 unidades de textura, haciendo un total de 5376, 5376, 2688, 672 y 336 respectivamente (totales ligeramente inferiores para la GPU usada). Cada stack de memoria DRAM HBM2 está controlada por un par de controladores de memoria de 512 bits (haciendo un total de 4096 bits). La GPU incluye un total de 6144 KB de cache L2.

Figura 2.6: Arquitectura de Volta GV100.

La Figura 2.6 muestra la arquitectura completa de NVIDIA Volta GV100, con 84 unidades *Streaming Multiprocessors* (SM). La arquitectura de los SM de Volta [44] incluye, entre otras funcionalidades, los ya mencionados *Tensor Cores* especializados en cómputos de Deep Learning, una eficiencia energética un 50 % superior en cargas de trabajo generales, una cache L1 de rendimiento mejorado y un nuevo modelo de threads SIMT (Single

Instruction Multiple Threads) que elimina las limitaciones en los diseños de procesador SIMT y SIMD de arquitecturas anteriores.

Los SM de Volta emplean un nuevo método de particionamiento que mejora la utilización y el rendimiento, particionando el SM en 4 bloques de procesamiento que constan de 16 cores FP32, 8 cores FP64, 16 cores INT32, 2 *Tensor Cores*, un cache de instrucciones L0, un planificador de *warp*, una unidad *dispatch* y un banco de registros de 64 KB.

La incorporación de un cache de instrucciones L0 permite obtener una mayor eficiencia que con los buffers de instrucciones empleados en anteriores GPU. La combinación de memoria compartida y cache L1 permiten incrementar la capacidad de memoria compartida hasta los 96 KB por cada SM. La Figura 2.7 ilustra la arquitectura de un *Streaming Multiprocessor* Volta.

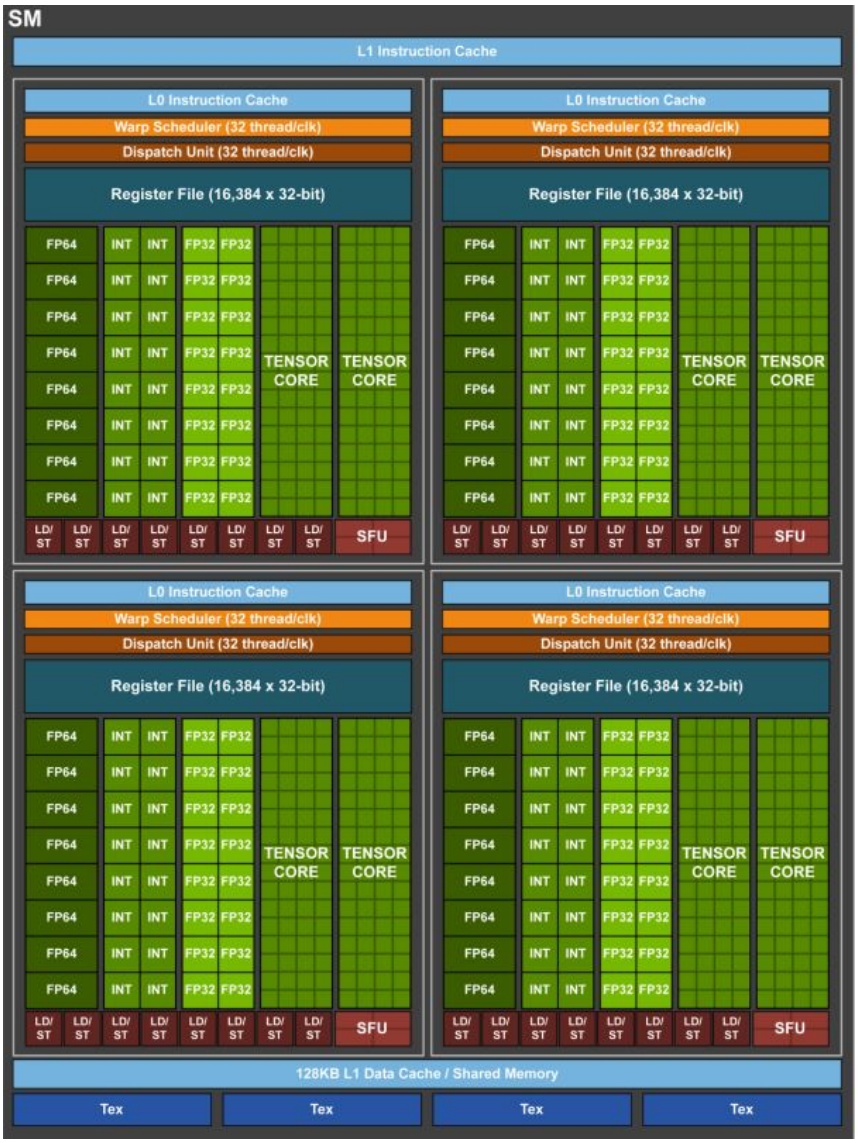

Figura 2.7: Arquitectura de un SM Volta.

La arquitectura NVIDIA Turing fue lanzada en 2018, suponiendo el mayor paso adelante en una década al proporcionar una nueva arquitectura de GPU que permite grandes avances en eficiencia y rendimiento para gaming, aplicaciones gráficas profesionales e inferencia de Deep Learning. Empleando aceleradores hardware y su enfoque de renderizado híbrido, la arquitectura Turing incorpora rasterización, *Ray Tracing* en tiempo real,

IA y simulación con el fin de conseguir un enorme realismo en videojuegos de PC, impresionantes efectos impulsados por redes neuronales, experiencias interactivas fluidas y de calidad cinematográfica al crear o navegar por modelos complejos 3D.

Las características clave que permiten a Turing obtener esas mejoras en el rendimiento son una arquitectura de *Streaming Multiprocessor* (SM) con eficiencia de ejecución de sombreados mejorada y una arquitectura del sistema de memoria que ofrece soporte para la tecnología GDDR6.

Las GPU de Turing [45] introducen los *RT Cores*, unidades aceleradoras dedicadas para la realización de operaciones de *Ray Tracing* con una extraordinaria eficiencia, eliminado los costosos enfoques de *Ray Tracing* basados en emulación por software empleados en el pasado. Esto permite realizar un renderizado en tiempo real de objetos fotorealistas y entornos con sombras, reflexiones y refracciones físicamente coherentes. Además, estos *RT Cores* son aprovechados por numerosas API como *Microsoft DXR*, *NVIDIA OptiX* y *Vulkan*.

Turing incorpora también los llamados *Tensor Cores*, que permiten acelerar enormemente las operaciones de cómputo tensorial y matricial que son la base de las aplicaciones de Deep Learning. Los *Tensor Cores* de Turing presentan un diseño mejorado para la inferencia, incluyendo nuevos modos de precisión INT8 e INT4 para cargas de trabajo que no requieren una precisión FP16.

Las GPU con arquitectura Turing también heredan todas las mejoras de la plataforma NVIDIA CUDA introducidas en la arquitectura Volta, como el scheduling de threads independientes, el Multi Process Service (MPS) acelerado por hardware con espacios de direcciones aislados y los Cooperative Groups. Esto permite incrementar la capacidad, flexibilidad, productividad y portabilidad de las aplicaciones desarrolladas con CUDA.

Los Streaming Multiprocessors (SM) de Turing presentan un incremento dramático en la eficiencia de sombreado (50 % respecto a la generación Pascal). Esto se debe a dos cambios clave en la arquitectura: en primer lugar una ruta de datos de enteros independiente capaz de ejecutar instrucciones concurrentemente a la ruta de datos de operaciones de coma flotante (en anteriores versiones la ejecución de estas instrucciones bloquearía la realización de instrucciones de coma flotante) y en segundo lugar la unificación de memoria compartida, cache de texturas y carga de memoria cache en una sola unidad (que se traduce en un ancho de banda y una capacidad duplicadas para la cache L1 en cargas de trabajo típicas).

La GPU está compuesta por 6 GPC (GPU Processing Clusters), 48 SM y 8 controladores de memoria de 32 bits (256 bits en total). Cada GPC incluye una unidad de rasterizado y 4 TPC (Texture Processing Clusters), conteniendo cada uno de éstos un Polimorph Engine y 2 SM. Cada *Streaming Multiprocessor* incluye 64 *CUDA cores*, un banco de registros de 256 KB, una cache de datos L1 y de memoria compartida de 96 KB y 4 unidades de textura. Se incluye además una conexión *x8 NVLink*, que proporciona 25 GB/s de ancho de banda en cada dirección.

La Figura 2.8 muestra muestra la arquitectura de NVIDIA Turing TU104, usada en la Quadro RTX 5000, con 48 unidades SM.

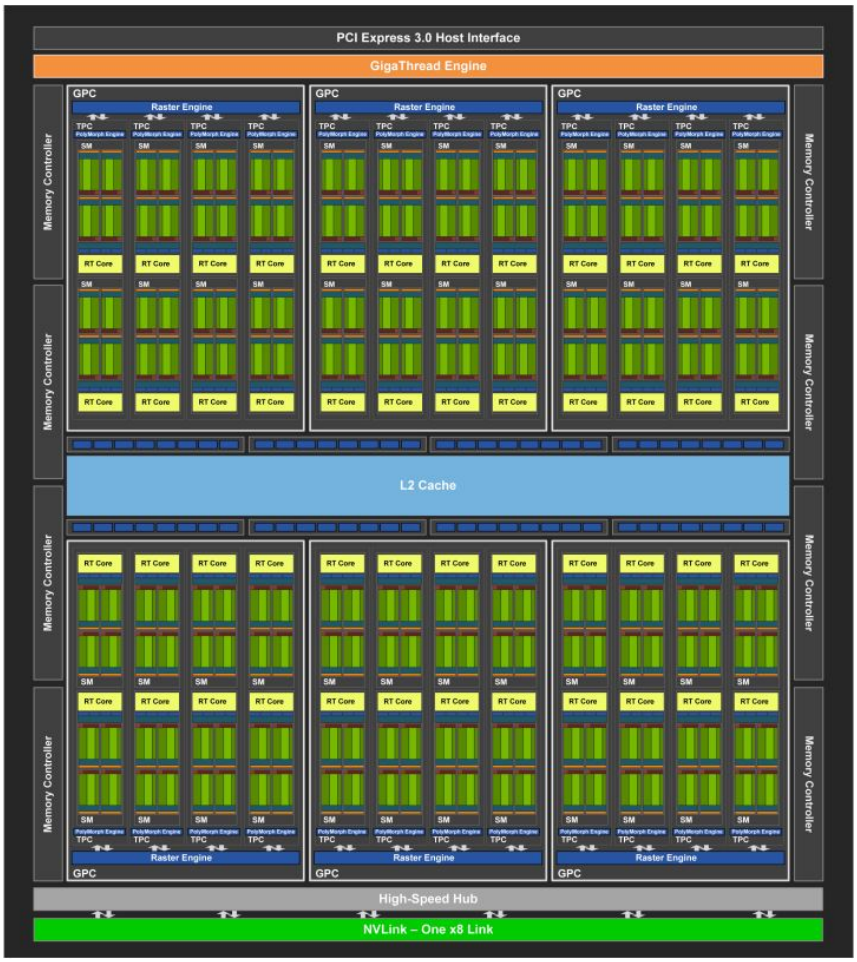

Figura 2.8: Arquitectura de Turing TU104.

La arquitectura de los SM de Turing [45] presenta un nuevo diseño que incorpora muchas de las características introducidas en la arquitectura Volta. Cuenta con 2 SM por cada TPC, teniendo cada SM un total de 64 cores FP32 y 64 cores INT32. Este SM soporta la ejecución concurrente de operaciones FP32 e INT32, como se mencionó anteriormente. Además, cada SM incluye un total de 8 *Tensor Cores* y un *RT Core*.

El SM está particionado en 4 bloques de procesamiento, teniendo cada uno 16 cores FP32, 16 cores INT32, 2 *Tensor Cores*, un planificador de *warp* y una unidad *dispatcher*. Cada bloque incorpora una cache de instrucciones L0 y un banco de registros de 64 KB. Los 4 bloques comparten una cache de datos L1 y de memoria compartida de 96 KB. En los SM de Turing se incluye una segunda unidad de ejecución paralela junto a cada *CUDA core*, la cual ejecuta las operaciones simples como sumas de enteros o comparación de números en coma flotante al tiempo que el *CUDA core* realiza los cómputos de coma flotante.

La Figura 2.9 ilustra la arquitectura de un Streaming Multiprocessor Turing.

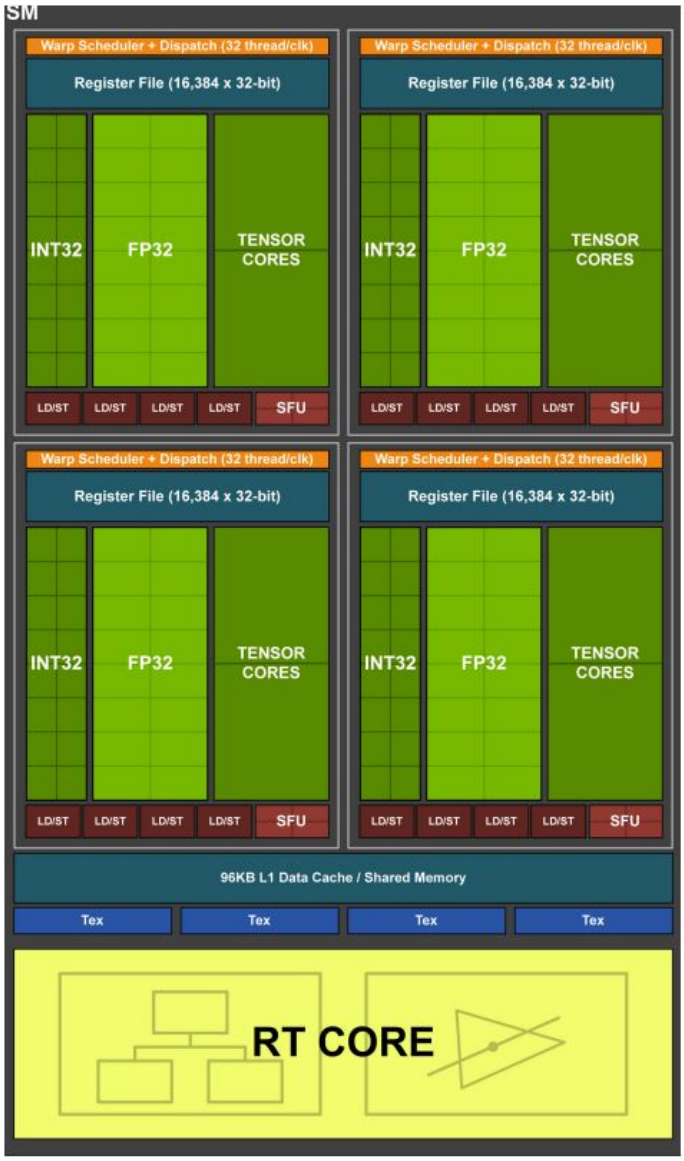

Figura 2.9: Arquitectura de un SM Turing.

En la Figura A.1 del **Apéndice** A se detalla un cuadro comparativo de las características de ambas GPU.

2.2.2. Software utilizado: Apache TVM

Apache TVM es un framework compilador open source de Machine Learning (ML) para CPU, GPU y aceleradores de Machine Learning. Su objetivo es permitir que los ingenieros de ML puedan optimizar y ejecutar cálculos eficientemente en todo tipo de backend hardware.

TVM comenzó en 2017 como un proyecto de investigación del grupo SAMPL de la Paul G. Allen School of Computer Science & Engineering, Universidad de Washington, y fue anunciado como *Top-Level Project* (TLP) de Apache el día 30 de noviembre de 2020. Apache TVM fue lanzado bajo la licencia *Apache License v2.0* [46].

La visión del *Apache TVM Project* es hospedar una diversa comunidad de expertos y practicantes de ML, compiladores y arquitecturas de sistemas para construir un framework open source accesible, extensible y automatizado que sea capaz de optimizar

modelos de ML, tanto actuales como emergentes, en cualquier tipo de plataforma hardware 2.10.

Las funcionalidades principales de TVM son la compilación de modelos Deep Learning a módulos lanzables mínimos y una infraestructura que permite generar y optimizar modelos de forma autómatica con un mejor rendimiento y en más tipos de backend [47].

Algunas de las características clave que hacen que TVM destaque son:

- *Alto rendimiento:* La compilación y los *runtimes* mínimos comúnmente desbloquean cargas de trabajo de ML en el hardware existente.
- *Ejecutable en cualquier tipo de hardware:* TVM genera y optimiza automáticamente operadores tensoriales en CPU, GPU, navegadores, microcontroladores, FPGA, ASIC...
- *Flexibilidad:* TVM incluye parseo de modelos de *TensorFlow*, *TensorFlow Lite* (TFLite), *Keras*, *PyTorch*, *MxNet*, *ONNX* y *Darknet*, entre otros. Soporta bloques dispersos, cuantización, random forests/ML clásico, planificación de memoria, compatibilidad con MISRA-C, prototipado Python...
- *Facilidad de uso:* Permite desarrollar stacks de producción empleando C++, Rust, Java o Python.

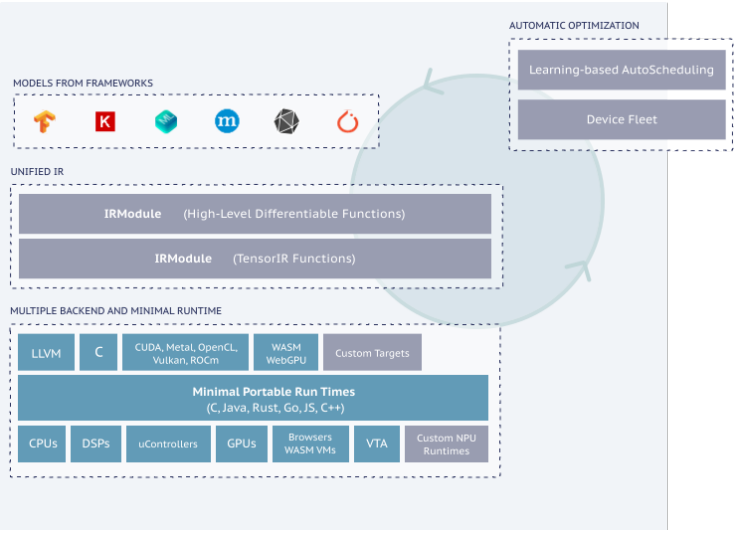

Figura 2.10: Características de Apache TVM.

Apache TVM es utilizado por docenas de corporaciones e instituciones, como Alibaba Cloud, AMD, ARM, AWS, Edge Cortix, Facebook, Huawei, Intel, Microsoft, NVIDIA, Oasis Labs, OctoML, Qualcomm, University of California/Berkeley, University of Washington, Xilinx, entre otros.

El esquema de la Figura 2.11 muestra los pasos por los que un modelo de Machine Learning pasa al ser transformado con el framework compilador-optimizador de TVM [48].

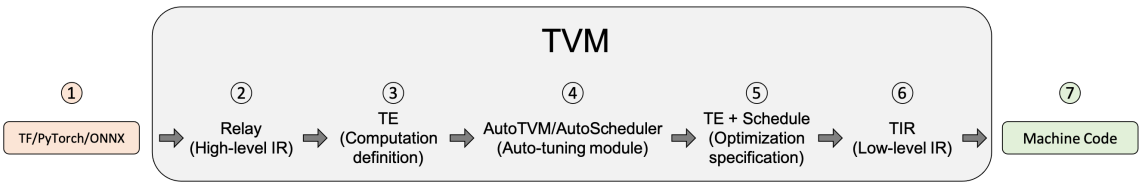

Figura 2.11: Proceso de optimización de modelos de TVM.

1. *Importar modelo:* Se importa un modelo procedente de un framework como *Tensorflow*, *PyTorch*, o *Onnx*. La capa importadora es donde TVM puede tomar modelos de otros frameworks. El nivel de soporte ofrecido por TVM difiere para cada frontend, pero es un proyecto open source en constante mejora. La documentación oficial [48] recomienda convertir a ONNX los modelos que puedan ser problemáticos.

2. *Traducción a Relay:* Se representa el modelo importado en *Relay*, el lenguaje de modelos de alto nivel de TVM. Relay es un lenguaje funcional y una expresión intermedia (IR) para redes neuronales. Relay soporta tanto representaciones de flujo de datos tradicionales como enfoques funcionales. Relay aplica optimización a nivel de grafos para optimizar el modelo.

3. *"Bajar nivel" a Tensor Expression (TE):* La expresión bajar nivel hace referencia a tomar una representación de alto nivel y transformarla a una de bajo nivel. Después de aplicar las optimizaciones de alto nivel, Relay ejecuta *FuseOps* para particionar el modelo en pequeños subgrafos y "baja el nivel" de dichos subgrafos a una representación en *Tensor Expression* (TE). TE es un lenguaje específico de dominio para la descripción de cálculos tensoriales. TE también incluye una serie de primitivas de *schedule* que permiten especificar optimizaciones de bucles de bajo nivel, como *tiling*, vectorización, paralelización, *unrolling* y fusión. Para esta conversión de Relay a TE, TVM incluye el Tensor Operator Inventory (TOPI), con plantillas predefinidas de operadores tensoriales comunes (conv2d, transpose...)

4. *Schedule:* Buscar el mejor *schedule* usando un módulo de *auto-tuning* (AutoTVM o AutoScheduler). Un *schedule* especifica optimizaciones de bucles de bajo nivel para un operador o un subgrafo definidos en TE. Los módulos de *auto-tuning* realizan una búsqueda del mejor *schedule* y lo comparan con modelos de coste y métricas del dispositivo. Los dos módulos de *auto-tuning* presentes en TVM son:

   - *AutoTVM:* Es un módulo de auto-tuning basado en plantillas. Ejecuta algoritmos de búsqueda para hallar los mejores valores para las "perillas giratorias" (parámetros ajustables) de una plantilla definida por el usuario. Para los operadores comunes, las plantillas vienen predefinidas en el TOPI.
   - *AutoScheduler (también conocido como Ansor):* Es un módulo de auto-tuning sin plantillas. No requiere plantillas de *schedule* predefinidas, sino que en su lugar genera el espacio de búsqueda automáticamente analizando la definición del cómputo, y a continuación busca el mejor *schedule* en el espacio de búsqueda generado.

5. *Escoger la configuración óptima:* Se escogen las configuraciones óptimas para la compilación del modelo. Después del proceso de *tuning*, el módulo de *auto-tuning* genera un informe de los resultados del *tuning* en formato JSON. En este paso se escoge el mejor *schedule* para cada subgrafo.

6. *"Bajar nivel" a Tensor Intermediate Representation (TIR):* Después de seleccionar las configuraciones óptimas en base al paso de *tuning*, cada subgrafo de TE se "baja de nivel" a TIR, la representación intermedia de bajo nivel de TVM, y se optimiza mediante optimizaciones de bajo nivel. A continuación, el TIR optimizado es bajado de nivel al compilador objetivo de la plataforma hardware correspondiente. Esta es la fase final de generación de código para producir un modelo optimizado que pueda ser usado en producción. TVM soporta diversos backends de compiladores, como:

- *LLVM*, que puede tener como destino arquitecturas de microprocesadores arbitrarias, como pueden ser procesadores estándar x86 y ARM o generación de código AMDGPU y NVPTX, entre otras plataformas.
- Compiladores especializados, como *NVCC*, el compilador de NVIDIA.
- Sistemas embebidos y objetivos especializados, los cuales son implementados mediante el framework Bring Your Own Codegen (BYOC) de TVM.

7. *Compilación:* Se compila el código a código máquina. Al final de este proceso, el código generado, específico del compilador, puede ser finalmente "bajado de nivel" a código máquina. TVM puede compilar modelos a un módulo objeto enlazable, que puede ser ejecutado por un ligero *runtime* de TVM que contiene las API de C que permiten cargar dinámicamente el modelo, y puntos de entrada para otros lenguajes como Python y Rust. TVM también puede generar un despliegue empaquetado en el que se combina en un solo paquete el runtime y el modelo.

Una de las herramientas más destacadas de TVM es TVMC, la interfaz de línea de comandos que permite utilizar las funcionalidades de TVM como *auto-tuning*, compilado, *profiling* y ejecución de modelos desde línea de comandos. TVMC es una aplicación de Python, parte del paquete TVM de Python [49].

Los comandos principales de TVMC son *compile*, para compilar un modelo para un dispositivo objetivo; *run*, para ejecutar un modelo previamente compilado usando el runtime de TVM y *tune*, para realizar un proceso de *tuning* sobre el modelo. *Tuning* en TVM se refiere al proceso mediante el cual un modelo es optimizado para que se ejecute más eficientemente en un objetivo dado. Difiere del entrenamiento o del *fine-tuning* en que no afecta a la precisión del modelo, sino únicamente a su rendimiento. En lo que respecta a la ejecución del *tuning*, será necesario proporcionar tres elementos: la especificación del dispositivo donde se desea ejecutar el modelo (expresado en un *string tvm.target*), la ruta al archivo de salida donde se guardarán los resultados del *tuning* y la ruta al modelo que se quiere "tunear".

TVM utiliza las llamadas *Tensor Expressions* (TE) para definir los cálculos tensoriales y aplicar las optimizaciones de bucles [50]. TE describe los cálculos tensoriales siguiendo un lenguaje funcional puro (en el cual las expresiones carecen de efectos secundarios). Relay describe un cálculo como una serie de operadores, siendo cada uno de dichos operadores representado por una TE, que toma tensores de entrada y produce un tensor de salida.

Tomando como ejemplo el cálculo de una suma de vectores (que no son sino tensores de orden 1), TVM adopta una semántica tensorial, en la que los resultados intermedios se representan como arrays multidimensionales. El usuario tendrá que encargarse de describir las reglas de cómputo que generarán los tensores. En primer lugar, se definen unos tensores placeholder A y B (los sumandos en este ejemplo), especificando su forma (dimensiones y tamaño), y un tensor resultado C, con una operación *compute*, donde se definirán tanto la forma del tensor como el cómputo a ser realizado en cada posición del tensor, definiéndolo en la función *lambda*.

A continuación, es preciso definir el *schedule* correspondiente al cómputo tensorial definido anteriormente. Un *schedule* es una serie de transformaciones de cómputo (operaciones) que transforma el bucle de cómputos del programa. En el caso de tener un tensor con múltiples ejes (si se tratase por ejemplo de una matriz, tendría 2 ejes), sería posible escoger sobre que eje iterar primero, o repartir el cómputo entre diferentes threads. Las operaciones de *scheduling* de TE permiten, entre otras operaciones, cambiar el orden de los bucles, dividir el cómputo entre diferentes threads y agrupar bloques de datos. Es imprescindible destacar que el *schedule* únicamente describe cómo se ha de realizar el

cómputo, por lo que distintos *schedule* para un mismo TE producirán siempre el mismo resultado.

A partir de la TE y el *schedule*, es posible generar código ejecutable para el lenguaje y arquitectura deseados. Para ello se emplea la función *tvm.build*, a la que se proporciona el *schedule*, los TE y el lenguaje y arquitectura objetivo. La función compilada presenta una concisa API de C que puede ser invocada por cualquier lenguaje de programación.

Algunas de las operaciones de *scheduling* más destacadas son [51]:

- *Parallel*: Permite paralelizar un bucle, repartiendo sus iteraciones entre los distintos threads del procesador. Se le pasa como parámetro el eje del tensor cuyo bucle se desea paralelizar.
- *Vectorize*: Permite vectorizar un bucle, sacando partido a los registros vectoriales con los que cuentan muchos procesadores modernos. Se le pasa como parámetro el eje del tensor cuyo bucle se desea vectorizar.
- *Split*: Divide un eje en dos ejes en base a un factor especificado. Se le pasa como parámetro el eje a dividir y el parámetro que especificará cómo se realizará la partición. Es posible especificar bien el factor de partición con el parámetro factor o bien el número de divisiones externas con nparts.
- *Tile*: Permite realizar la partición a lo largo de dos ejes. Se le pasan como parámetro los dos ejes y los factores de partición de cada eje.
- *Fuse*: Permite fusionar dos ejes de cómputo en uno solo.
- *Reorder*: Permite reordenar los ejes de cómputo en el orden especificado. Se pasan como parámetro los ejes a reordenar, en el nuevo orden deseado.
- *Bind*: Permite asignar un cómputo (especificado por el eje o bucle pasado por parámetro) a un thread específico. Es útil en programación de GPU, al permitir repartir el cómputo entre los distintos bloques de threads, así como entre los distintos threads de un bloque.
- *Compute_at*: Permite establecer que el cómputo de un tensor sea realizado dentro del primer eje de cómputo de otro tensor pasado por parámetro. Por defecto, en un *schedule* que conste de múltiples operadores, TVM computará los tensores en el nivel más externo de la función, o en la raíz.
- *Compute_inline*: Permite establecer que el cómputo de un tensor sea expandido e insertado en la dirección donde este tensor sea requerido.
- *Compute_root*: Permite mover el cómputo de un tensor a la capa más externa o raíz. Esto significa que este cómputo será realizado por completo antes de pasar al siguiente tensor.

CAPÍTULO 3
Desarrollo e implementación

En este capítulo se expone el proceso de desarrollo del código implementado en este trabajo para la realización del producto matricial empleando la herramienta Apache TVM, enumerando y presentando las distintas versiones por las que ha ido pasando hasta su iteración final.

3.1 Introducción

El código que se ha desarrollado en este trabajo, y sobre el cual se han realizado posteriormente los experimentos y mediciones, realiza la operación GEMM (General Matrix Multiply, es decir, un producto matricial general), un algoritmo común en álgebra lineal, Machine Learning, estadística y otros muchos campos. La operación GEMM es, como se mencionó anteriormente, un bloque de construcción fundamental en las redes neuronales.

Para implementarlo, se ha empleado el lenguaje Python y se han utilizado las *Tensor Expressions* (TE) que el paquete TVM incluye. La Figura 3.1 muestra la inicialización del entorno de TVM con la selección del *target* (en este caso CUDA) utilizando la clase *tvm.target* (línea 3). Las líneas 6 y 7 definen dos tensores A y B (correspondientes a las dos matrices a multiplicar), que serán de tamaño $M \times K$ y $K \times N$ respectivamente. A continuación se define el eje k sobre el que se realizará la reducción. El tensor C, que almacenará el resultado del producto, se describe con una operación *compute*. Ésta define un cómputo, con una salida conforme con la forma especificada para el tensor ($M \times N$) y el cómputo que ha de ser realizado en cada posición del tensor, definido por la función *lambda* (líneas 13–16). Es importante destacar que en este paso no se está realizando ningún tipo de cómputo, sino únicamente declarando cómo debe hacerse dicho cómputo.

```
# TVM Matrix Multiplication con TE
# Seleccionar la arquitectura
tgt_gpu = tvm.target.Target(target="cuda", host="llvm")

# Definicion de los operandos A y B
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")

# Definicion eje de reduccion
k = te.reduce_axis((0, K), "k")

# Definicion de la operacion
C = te.compute((M, N),
    lambda x, y:
    te.sum(A[x, k] * B[k, y], axis=k),
    name="C")
```

Figura 3.1: Operación GEMM utilizando Tensor Expressions de TVM.

Las expresiones anteriores únicamente describen las reglas del cómputo de *C*, pero éste puede ser realizado de diversas maneras con el fin de optimizarlo a diferentes tipos de dispositivos. Algunas de estas optimizaciones incluyen el reparto del cómputo entre diferentes threads, o en los casos en los que se dispone de un tensor con múltiples ejes (es decir, más de una dimensión), la elección de qué eje/s debe/n ser iterado/s primero.

Para ello TVM cuenta con las operaciones de planificación (o *scheduling* en inglés), las cuales únicamente describen cómo se realiza un cómputo, por lo que distintas *schedule* para un mismo TE producirán el mismo resultado.

El *schedule* usado se tratará a lo largo de las distintas subsecciones que hacen referencia a las distintas versiones, ya que es la única parte del código que varía entre las mismas, debido a que es la que permitirá aprovechar las funcionalidades de TVM para optimizar el cómputo de la operación GEMM en el dispositivo a utilizar [52, 53].

La Figura 3.2 muestra como se construye la operación una vez planificada. La línea 2 genera el planificador (*scheduler*) para el operando *C*, y la línea 5 construye la función recibiendo como parámetros de entrada el *scheduler*, los tensores y el *target* (línea 5).

Para ejecutar la función, se crea un *device* (que representa el dispositivo donde se ejecutará la función) utilizando la clase *tvm.device* y se inicializan los tensores en éste. Finalmente se puede realizar la llamada a la función, que será ejecutada en el dispositivo especificado (líneas 8–13).

```
# Crear el planificador para la operacion
s = te.create_schedule(C.op)

# Construir la funcion con el planificador, los tensores y el target
gemm = tvm.build(s, [A, B, C], target=tgt_gpu, name="mmult")

# Ejecucion de la funcion
dev = tvm.device(tgt_gpu.kind.name, 0)
d_a = tvm.nd.array(a, dev)
d_b = tvm.nd.array(b, dev)
d_c = tvm.nd.array(numpy.zeros((M, N), dtype=dtype), dev)

gemm(d_a, d_b, d_c)
```

Figura 3.2: Compilado y ejecución de la función en el device.

3.2 Progreso del algoritmo

A lo largo de los siguientes subapartados, se irán explicando detalladamente las distintas versiones que han sido implementadas y se analizará el rendimiento (medido en GFLOPS) de cada una de las versiones.

Para realizar el cálculo de GFLOPS, se tendrá en cuenta que el número de operaciones en coma flotante de un producto matricial es de $2mnk$, debido a que en cada iteración se realiza una operación de multiplicación y una de suma. Cabe destacar que todas las mediciones de rendimiento durante el proceso de desarrollo han sido realizadas empleando matrices cuadradas con un tamaño 2048×2048.

Versión 1

En este primer enfoque (Figura 3.3), el más básico, únicamente se divide el cómputo de uno de los ejes de la matriz C (donde se almacenará el resultado del producto matricial) entre los threads y los bloques de threads, empleando una única coordenada de id de bloque, así como una única coordenada de id de thread. Se emplea la función *split* (línea 13) para dividir el cómputo del primer eje de la matriz C (mi) entre la variable de iteración bx (que se asignará al id de bloque de threads) y la variable de iteración tx (que se asignará al id de thread). Esta división se realiza en función del parámetro $num\_threads$.

Con esto se consigue dividir el cómputo en tantos threads como elementos tiene el eje de la matriz a calcular, estando dichos threads agrupados en bloques de $num\_threads$ hilos. Por tanto, el número de bloques será $MATRIX\_SIZE/num\_threads$. Una vez realizado el *split*, se utiliza la operación *bind* (líneas 16-17) para asignar las variables de iteración a los id de thread de GPU. En la Figura 3.4 se muestra el código CUDA generado por el *schedule* definido en esta primera versión.

```
1  # Creamos el schedule
2  s = te.create_schedule(C.op)
3
4  # Establecemos las opciones de tiling
5  num_thread = 8
6
7  # Obtenemos los indices de thread de GPU
8  block_x = te.thread_axis("blockIdx.x")
9  thread_x = te.thread_axis((0, num_thread), "threadIdx.x")
10
11 # Dividimos las cargas de trabajo
12 mi, ni = s[C].op.axis
13 bx, tx = s[C].split(mi, factor=num_thread)
14
15 # Asignamos las variables de iteracion a los indices de thread de GPU
16 s[C].bind(bx, block_x)
17 s[C].bind(tx, thread_x)
```

Figura 3.3: Schedule de la Versión 1.

```
1 extern "C" __global__ void __launch_bounds__(8) mmult_kernel0(float*
      __restrict__ C, float* __restrict__ A, float* __restrict__ B) {
2   for (int y = 0; y < 2048; ++y) {
3     C[((((((int)blockIdx.x) * 16384) + (((int)threadIdx.x) * 2048)) + y)] =
      0.000000e+00f;
4     for (int k = 0; k < 2048; ++k) {
5       C[((((((int)blockIdx.x) * 16384) + (((int)threadIdx.x) * 2048)) + y)]
      = (C[((((((int)blockIdx.x) * 16384) + (((int)threadIdx.x) * 2048)) + y)]
       + (A[((((((int)blockIdx.x) * 16384) + (((int)threadIdx.x) * 2048)) + k)
      ] * B[((k * 2048) + y)]));
6     }
7   }
8 }
```

Figura 3.4: Código generado por el schedule de la Versión 1.

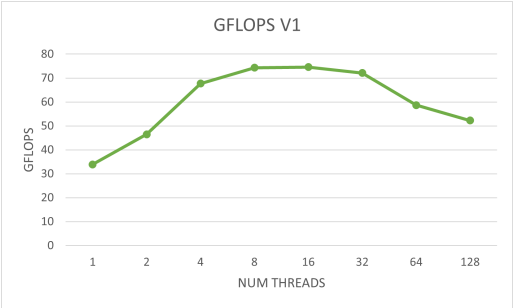

Figura 3.5: Gráfico del rendimiento de la versión V1 en función de num threads en la máquina Quadro.

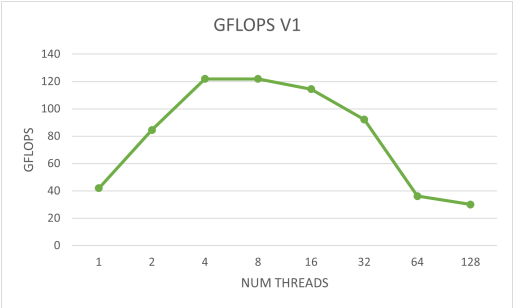

Figura 3.6: Gráfico del rendimiento de la versión V1 en función de num threads en la máquina Volta.

Como se puede ver (Figura 3.5 y Figura 3.6), esta primera versión consigue alcanzar un rendimiento máximo de entre 70 y 80 GFLOPS en la máquina Quadro, y de unos 120 GFLOPS en la máquina Volta. Los valores de *num_threads* que permiten obtener un mayor rendimiento son 8, 16 y 32 en la primera, y 4 y 8 en la segunda. Esto se debe a que con un menor número de threads no se aprovecha del todo la potencia de la GPU, y con uno mayor se lanza un número de bloques de threads demasiado reducido, ya que éste depende directamente del número de threads por bloque. Comparando el rendimiento en ambas máquinas se puede apreciar que en la máquina Volta se experimenta una abrupta caída en prestaciones al incrementar *num_threads* a 64, siendo aún más ineficiente que cuando se emplea un *num_threads* de 1.

Versión 2

El único cambio implementado en esta versión (Figura 3.7) es la realización del *split* (línea 9) sobre el segundo eje de operación de la matriz *C* (*ni*) en vez del primero (*mi*). La razón por la que esto puede permitir obtener un mayor rendimiento es que de esta forma se consigue hacer un uso más eficiente de la memoria, al permitirse que los threads accedan a posiciones contiguas, pues la matriz se almacena en *row-major-order*.

```
1  # Creamos el schedule
2
3  # Establecemos las opciones de tiling
4
5  # Obtenemos los indices de thread de GPU
6
7  # Dividimos las cargas de trabajo
8  mi, ni = s[C].op.axis
9  bx, tx = s[C].split(ni, factor=num_thread)
10
11 # Asignamos las variables de iteracion a los indices de thread de GPU
```

Figura 3.7: Schedule de la Versión 2.

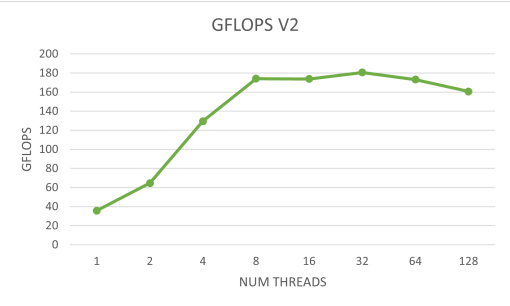

Figura 3.8: Gráfico del rendimiento de la versión V2 en función de num threads en la máquina Quadro.

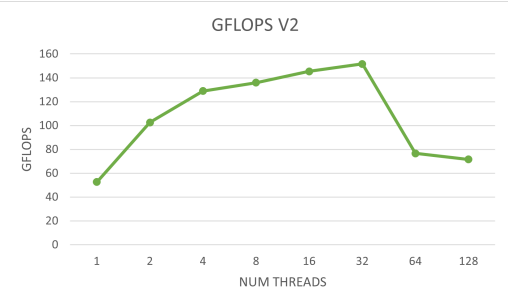

Figura 3.9: Gráfico del rendimiento de la versión V2 en función de num threads en la máquina Volta.

Esta pequeña modificación ha logrado incrementar el rendimiento hasta 180 GFLOPS en la máquina Quadro (Figura 3.8), y 150 GFLOPS en la máquina Volta (Figura 3.9), siendo 32 en ambos casos el valor de *num_threads* que permite alcanzarlos. En la máquina Quadro, el incremento en los GFLOPS respecto de V1 es más notable conforme se incrementa *num_threads*, disminuyendo de forma apreciable la merma en el rendimiento ocasionada al incrementar *num_threads* por encima de 32. Por otra parte, en la máquina Volta el incremento de GFLOPS es más gradual, y se mantiene la notable caída de rendimiento al incrementar *num_threads* a 64.

Versión 3

Para esta versión (Figura 3.10), ya se empiezan a utilizar dos coordenadas, tanto de threads como de bloques de threads. Se realizan, por tanto dos *splits* (líneas 13-14), uno por cada eje de operación de la matriz, dividiendo el cómputo del primer eje (*mi*) entre *bx* y *tx* (primera coordenada de id de bloque y primera coordenada de id de thread) y el segundo eje (*ni*) entre *by* y *ty* (segundas coordenadas). De esta forma, el número total de threads será igual al total de elementos de la matriz resultado, estando estos threads agrupados en bloques de tamaño $num\_threads \times num\_threads$). Por tanto, el número de bloques dependerá tanto del tamaño de la matriz como de *num_threads*.

```
# Creamos el schedule

# Establecemos las opciones de tiling

# Obtenemos los indices de thread de GPU
block_x = te.thread_axis("blockIdx.x")
block_y = te.thread_axis("blockIdx.y")
thread_x = te.thread_axis((0, num_thread), "threadIdx.x")
thread_y = te.thread_axis((0, num_thread), "threadIdx.y")

# Dividimos las cargas de trabajo
mi, ni = s[C].op.axis
bx, tx = s[C].split(mi, factor=num_thread)
by, ty = s[C].split(ni, factor=num_thread)

# Asignamos las variables de iteracion a los indices de thread de GPU
s[C].bind(by, block_y)
s[C].bind(bx, block_x)
s[C].bind(ty, thread_y)
s[C].bind(tx, thread_x)
```

Figura 3.10: Schedule de la Versión 3.

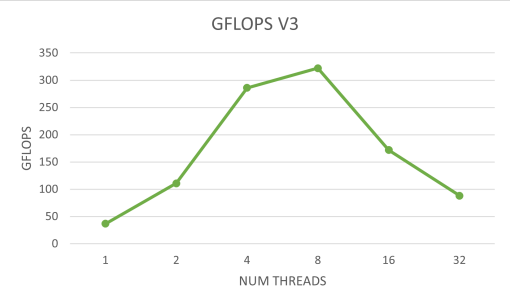

Figura 3.11: Gráfico del rendimiento de la versión V3 en función de num threads en la máquina Quadro.

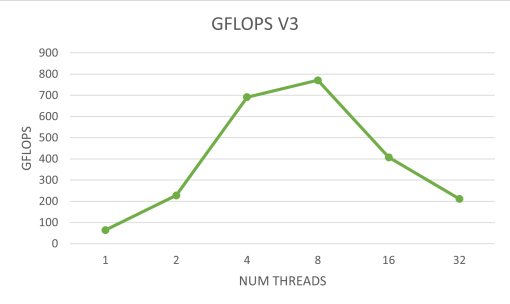

Figura 3.12: Gráfico del rendimiento de la versión V3 en función de num threads en la máquina Volta.

La introducción de 2 coordenadas ha supuesto una gran mejora en las prestaciones, incrementando el rendimiento respecto de la versión anterior en más de un 50 % en la máquina Quadro (Figura 3.11), y quintuplicándolo en la máquina Volta (Figura 3.12). Claramente, el tamaño óptimo de bloque en ambas máquinas es de 8×8 threads (es decir, $num\_threads = 8$), consiguiendo superar los 300 y los 700 GFLOPS respectivamente, siguiéndole de cerca el tamaño de 4. Para valores mayores, el rendimiento disminuye considerablemente.

Esto puede deberse a que en dichos casos se lanza un número de bloques muy reducido (pues éste depende del tamaño de la matriz, que no varía, y de *num_threads*). Esta versión, a diferencia de las dos anteriores, se comporta de forma idéntica en ambas máquinas en cuanto a cómo se ve afectado el rendimiento por la modificación del parámetro *num_threads*.

Versión 4

De igual modo que en la versión V2, se realiza una reasignación de los ejes de operación de la matriz (Figura 3.13), asignándose en este caso el primer eje de operación (*mi*) a las coordenadas Y de las id de bloque y de thread (línea 9), y el segundo eje de operación (*ni*) a las coordenadas X (línea 10). Esto permite obtener un mayor rendimiento debido al ordenamiento de los threads en CUDA, ya que independientemente de si los

bloques son de una, dos o tres dimensiones, al final, los threads se ordenan en un vector "lineal" para ser planificados por el scheduler y entrar en los cores para su ejecución [40], y de esta forma se consigue que threads contiguos (que se encuentran en posiciones adyacentes del eje X y que serán lanzados de forma simultánea para su ejecución) operen con posiciones contiguas de la matriz (ya que ésta se almacena en memoria siguiendo un *row-major-order*).

```
# Creamos el schedule

# Establecemos las opciones de tiling

# Obtenemos los indices de thread de GPU

# Dividimos las cargas de trabajo
mi, ni = s[C].op.axis
by, ty = s[C].split(mi, factor=num_thread)
bx, tx = s[C].split(ni, factor=num_thread)

# Asignamos las variables de iteracion a los indices de thread de GPU
```

Figura 3.13: Schedule de la Versión 4.

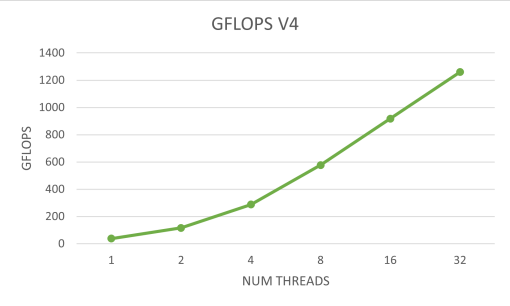

Figura 3.14: Gráfico del rendimiento de la versión V4 en función de num threads en la máquina Quadro.

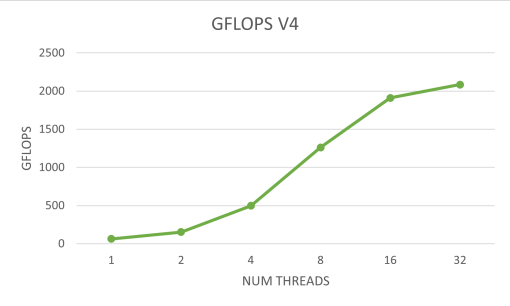

Figura 3.15: Gráfico del rendimiento de la versión V4 en función de num threads en la máquina Volta.

La inversión de las asignaciones de threads a los ejes de la matriz ha conseguido eliminar el descenso en el rendimiento producido en la versión anterior cuando el número de threads por bloque superaba 8×8. En esta versión, el rendimiento continúa aumentando hasta el máximo de threads por bloque permitido ($num\_threads = 32$, con lo que cada bloque tiene 32×32 threads, que hacen un total de 1024, el máximo permitido por CUDA), alcanzado los 1200 GFLOPS en la máquina Quadro (Figura 3.14) y superando los 2000 GFLOPS en la máquina Volta (Figura 3.15) gracias a este uso más eficiente de la memoria.

Esta mejora ha sido hasta el momento, teniendo en cuenta ambas máquinas, la que ha permitido un mayor incremento en las prestaciones respecto de su predecesora, cuadruplicádola en la primera y triplicándola en la segunda. Técnicamente la versión que supuso un mayor cambio en la máquina Volta fue V3, pero la mejora ocasionada por esa versión fue mucho menos notable en la máquina Quadro.

Versión 5

En esta versión (Figura 3.16) se introduce el concepto de *elements_per_thread* (línea 4), variable que controlará el número de elementos de la matriz que serán calculados por un mismo thread, a diferencia de todas las versiones anteriores, donde cada thread calculaba una única posición de la matriz, habiendo siempre tantos threads como elementos a calcular y dependiendo del número de bloques de threads únicamente del tamaño de la matriz y del número de threads. En esta nueva versión, en cambio, el número de bloques irá determinado (línea 6) por $MATRIX\_SIZE/(elements\_per\_thread \times num\_threads)$. En vez de tener un *split* por cada eje de la matriz, se dispondrá de un total de 4 *splits*, 2 por cada eje.

En primer lugar (líneas 12-13) se divide el cómputo total entre los distintos bloques de threads (conservando la inversión de ejes introducida en la versión V4) y posteriormente se divide el cómputo asignado a cada bloque entre cada uno de los distintos threads, en función de la variable *num_threads* (líneas 19-20). Con esto se consigue que cada thread procese *elements_per_thread* $\times$ *elements_per_thread* elementos de la matriz. Finalmente, se realiza un *reorder* para reordenar los bucles (línea 21), siendo los más externos los que se asignan a los id de bloque, a continuación los que se asignan a los id de thread y finalmente los que realiza cada thread.

```
1  # Creamos el schedule
2
3  # Establecemos las opciones de tiling
4  elements_per_thread = 8
5  num_thread = 8
6  block_factor = elements_per_thread * num_thread
7
8  # Obtenemos los indices de thread de GPU
9
10 # Dividimos las cargas de trabajo
11 mi, ni = s[C].op.axis
12 by, mi = s[C].split(mi, factor=block_factor)
13 bx, ni = s[C].split(ni, factor=block_factor)
14
15 # Asignamos las variables de iteracion a los indices de thread de GPU
16 s[C].bind(by, block_y)
17 s[C].bind(bx, block_x)
18
19 ty, mi = s[C].split(mi, nparts=num_thread)
20 tx, ni = s[C].split(ni, nparts=num_thread)
21 s[C].reorder(by, bx, ty, tx, mi, ni)
22
23 s[C].bind(ty, thread_y)
24 s[C].bind(tx, thread_x)
```

Figura 3.16: Schedule de la Versión 5.

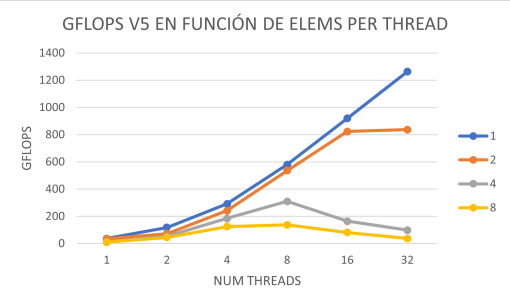

Figura 3.17: Gráfico del rendimiento de la versión V5 en función de elements per thread en la máquina Quadro.

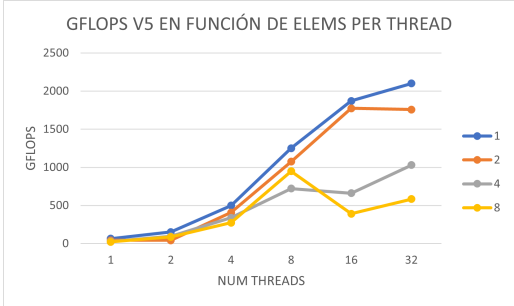

Figura 3.18: Gráfico del rendimiento de la versión V5 en función de elements per thread en la máquina Volta.

Las gráficas (Figura 3.17 y Figura 3.18) muestran que el incremento en el rendimiento producido conforme aumenta *num_threads* se da únicamente cuando *elements_per_thread* tiene un valor de 1 ó de 2 (en este último caso los GFLOPS dejan de aumentar cuando se alcanzan los 16 × 16 threads por bloque), produciéndose una mayor pérdida de prestaciones al incrementar *elements_per_thread* en la máquina Quadro.

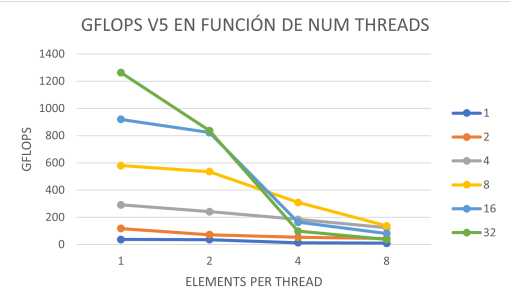

Figura 3.19: Gráfico del rendimiento de la versión V5 en función de num threads en la máquina Quadro.

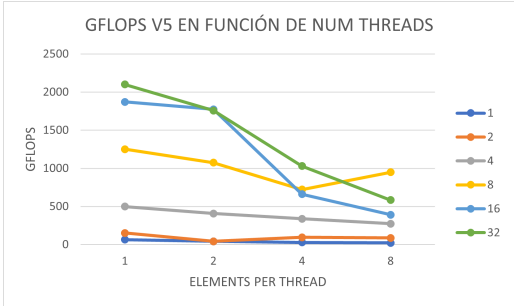

Figura 3.20: Gráfico del rendimiento de la versión V5 en función de num threads en la máquina Volta.

Las gráficas mostradas en la Figura 3.19 y la Figura 3.20 reafirman que a mayor valor de *elements_per_thread*, menor rendimiento, siendo más acusada la caída cuando *num_threads* es mayor. Por tanto, se puede afirmar que la inclusión de este parámetro no ha permitido incrementar las prestaciones.

Versión 6

Una de las posibles causas de que la inclusión de *elements_per_thread* en la versión V5 empeorase el rendimiento podría ser que en dicha versión, a diferencia de las anteriores cada thread debe realizar más de una escritura (concretamente *elements_per_thread* × *elements_per_thread*) en memoria, y los accesos a memoria suelen ser un cuello de botella para el rendimiento.

Para remediarlo, en esta versión (Figura 3.21) se incluye una memoria local de escritura (que recibirá el nombre de *CL*) utilizada como buffer temporal a la hora de que cada thread escriba los resultados que ha calculado. Esto permitirá que cada thread únicamente realice una escritura en la memoria RAM, pues irá almacenando los elementos que vaya calculando (un total de *elements_per_thread* × *elements_per_thread*) en *CL* en vez de en *C*.

Para implementarlo, se define CL como una memoria cache de escritura de ámbito local para el tensor C (línea 4) y se establece el cómputo de dicha memoria CL dentro del bucle tx, que es el bucle más interno de los dos que se asignan a las id de thread (línea 15). Finalmente, tiene lugar un *reorder* (línea 18) para que el bucle más externo del cómputo de CL sea el correspondiente al eje de reducción.

```
1  # Creamos el schedule
2
3  # Desginamos la jerarquia de memoria
4  CL = s.cache_write(C, "local")
5
6  # Establecemos las opciones de tiling
7
8  # Obtenemos los indices de thread de GPU
9
10 # Dividimos las cargas de trabajo
11
12 # Asignamos las variables de iteracion a los indices de thread de GPU
13
14 # Schedule de la memoria de escritura local CL
15 s[CL].compute_at(s[C], tx)
16 mi, ni = s[CL].op.axis
17 rk, = s[CL].op.reduce_axis
18 s[CL].reorder(rk, mi, ni)
```

Figura 3.21: Schedule de la Versión 6.

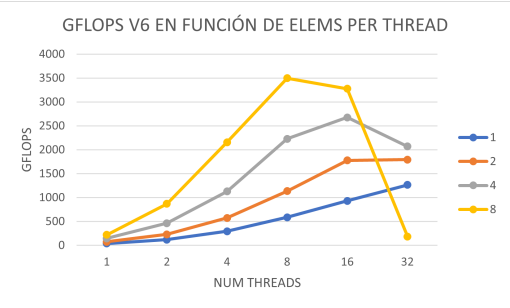

Figura 3.22: Gráfico del rendimiento de la versión V6 en función de elements per thread en la máquina Quadro.

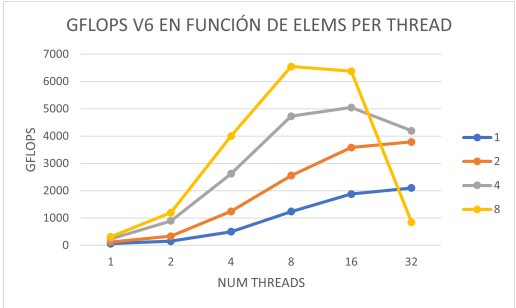

Figura 3.23: Gráfico del rendimiento de la versión V6 en función de elements per thread en la máquina Volta.

Se puede ver (Figura 3.22 y Figura 3.23) que la incorporación de la memoria CL ha permitido sacar partido al parámetro *elements_per_thread*, pues la aplicación conjunta de ambas mejoras ha permitido elevar las prestaciones hasta 3500 GFLOPS en la máquina Quadro y 6500 GFLOPS en la máquina Volta, siendo aproximadamente el triple que sin ellas. Se aprecia que por lo general a mayor valor de *elements_per_thread*, mayor rendimiento, y que el número óptimo de threads varía en función del valor del otro parámetro. Los valores óptimos de los parámetros son $elements\_per\_thread = 8$ y $num\_threads = 8$, aunque subiendo este último parámetro a 16, el rendimiento apenas baja.

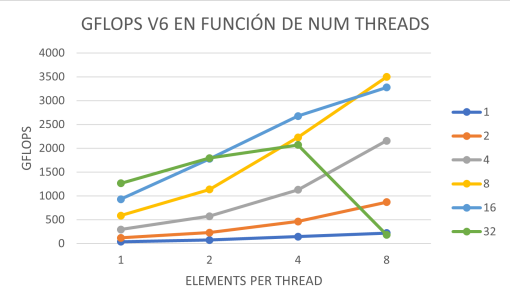

Figura 3.24: Gráfico del rendimiento de la versión V6 en función de num threads en la máquina Quadro.

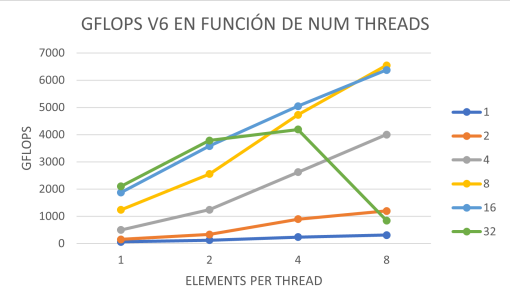

Figura 3.25: Gráfico del rendimiento de la versión V6 en función de num threads en la máquina Volta.

Lo más destacado de estos gráficos (Figura 3.24 y Figura 3.25) radica en el hecho de que incrementar el valor de *elements_per_thread* permite obtener siempre un mayor rendimiento, excepto en el caso de $num\_threads = 32$, pues en éste el máximo se alcanza con un valor de 4, disminuyendo bruscamente al elevarlo a 8. Esto sucede de igual manera en las dos máquina estudiadas.

Versión 7

Aquí (Figura 3.26) se realizan dos mejoras, en primer lugar la implementación de dos memorias globales de lectura, una por cada matriz de entrada (que serán llamadas AA y BB). Estas memorias se utilizan a modo de cache para que los threads no tengan que estar accediendo constantemente a la memoria RAM para leer los datos, pudiendo acceder a un buffer de memoria que es común para los threads de un mismo bloque (sacando además partido de que los threads de un mismo bloque trabajan con posiciones contiguas de la matriz), siendo dicho buffer más lento que una memoria local del thread pero notablemente más rápido que la RAM. Para ello, se definirán AA y BB como memorias cache de lectura de ámbito global (compartidas) para los tensores A y B respectivamente (líneas 4-5).

En segundo lugar, con esta versión también se agrega el parámetro *step* (línea 12), que controla la cantidad de datos que se cargarán de una misma vez en las memorias globales. Por tanto, este parámetro controla directamente el tamaño de dichas memorias (AA y BB), siendo éste de $elements\_per\_thread \times num\_threads \times step$. Para poder implementarlo, en el cómputo del eje de reducción de CL se realiza un *split* de factor *step* (línea 24), teniendo lugar a continuación un *reorder* para que el bucle más externo del cómputo de CL sea el correspondiente al eje de reducción (línea 25).

El cómputo de AA y BB se sitúa en el eje más externo de dicho *split* (líneas 28-29), situando por tanto su cómputo dentro del cómputo de CL. El *schedule* de la carga en memoria de AA (líneas 32-37) y BB (líneas 40-45) se realiza particionando el cómputo únicamente entre las distintas id de thread (ya que al ser una memoria global, es común a todos los threads de un mismo bloque), realizando cada thread *elements_per_thread* cómputos.

```
# Creamos el schedule

# Desginamos la jerarquia de memoria
AA = s.cache_read(A, "shared", [C])
BB = s.cache_read(B, "shared", [C])
CL = s.cache_write(C, "local")

# Establecemos las opciones de tiling
elements_per_thread = 8
num_thread = 8
block_factor = elements_per_thread * num_thread
step = 8

# Obtenemos los indices de thread de GPU

# Dividimos las cargas de trabajo

# Asignamos las variables de iteracion a los indices de thread de GPU

# Schedule de la memoria de escritura local CL
s[CL].compute_at(s[C], tx)
mi, ni = s[CL].op.axis
rk, = s[CL].op.reduce_axis
rko, rki = s[CL].split(rk, factor=step)
s[CL].reorder(rko, rki, mi, ni)

# Unimos el computo a las variables de iteracion
s[AA].compute_at(s[CL], rko)
s[BB].compute_at(s[CL], rko)

# Schedule para la carga en la memoria compartida AA
mi, ni = s[AA].op.axis
ty, mi = s[AA].split(mi, nparts=num_thread)
tx, ni = s[AA].split(ni, nparts=num_thread)
s[AA].reorder(ty, tx, mi, ni)
s[AA].bind(ty, thread_y)
s[AA].bind(tx, thread_x)

# Schedule para la carga en la memoria compartida BB
mi, ni = s[BB].op.axis
ty, mi = s[BB].split(mi, nparts=num_thread)
tx, ni = s[BB].split(ni, nparts=num_thread)
s[BB].reorder(ty, tx, mi, ni)
s[BB].bind(ty, thread_y)
s[BB].bind(tx, thread_x)
```

Figura 3.26: Schedule de la Versión 7.

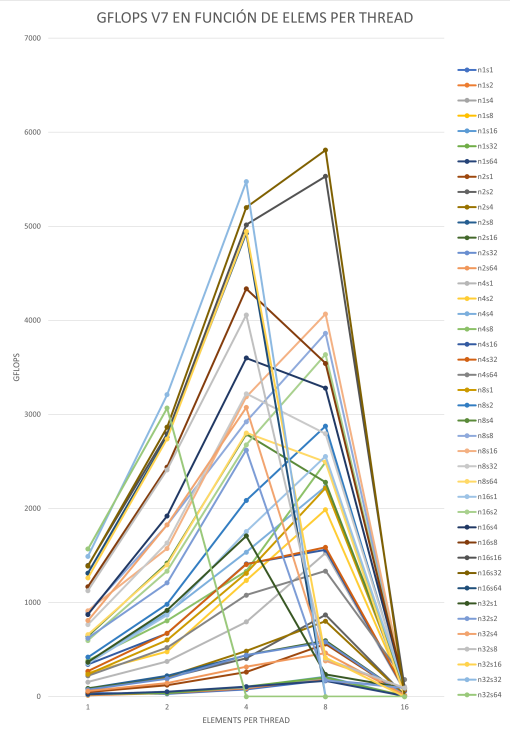

Figura 3.27: Gráfico del rendimiento de la versión V7 en función de elements per thread en la máquina Quadro.

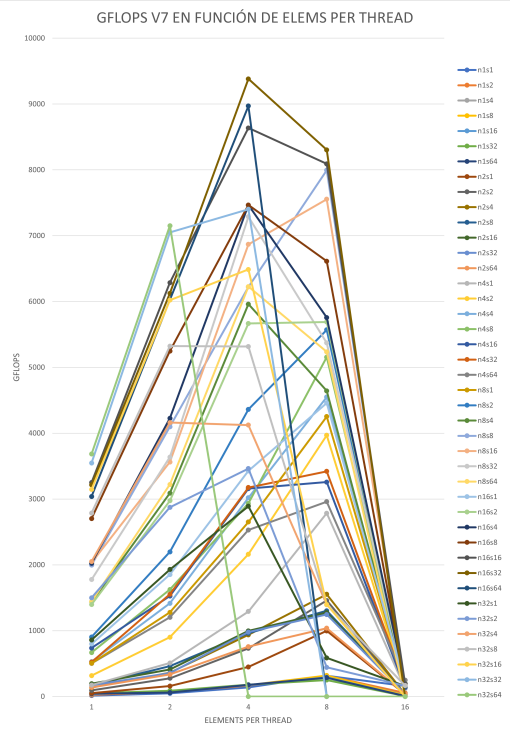

Figura 3.28: Gráfico del rendimiento de la versión V7 en función de elements per thread en la máquina Volta.

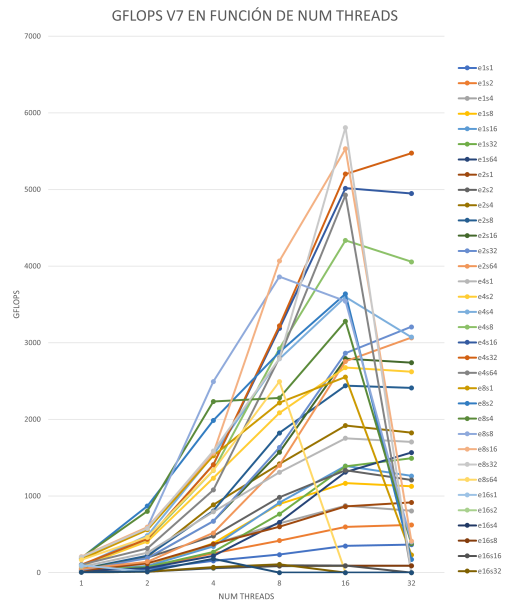

Figura 3.29: Gráfico del rendimiento de la versión V7 en función de num threads en la máquina Quadro.

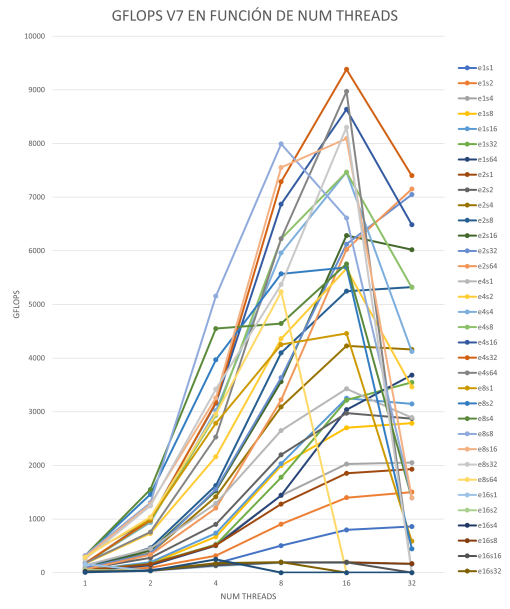

Figura 3.30: Gráfico del rendimiento de la versión V7 en función de num threads en la máquina Volta.

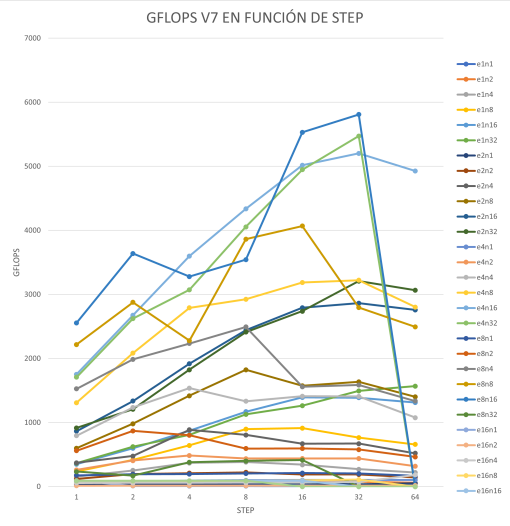

Figura 3.31: Gráfico del rendimiento de la versión V7 en función de step en la máquina Quadro.

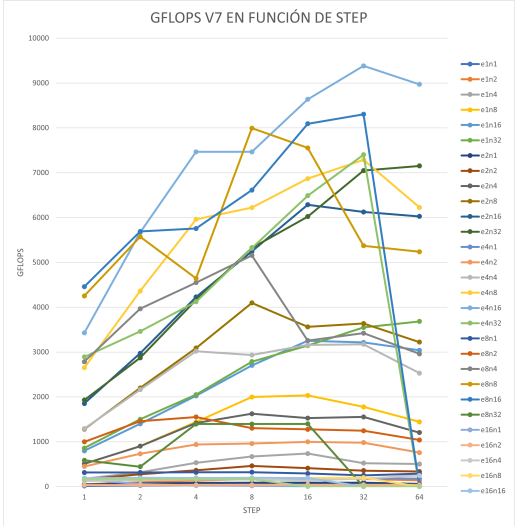

Figura 3.32: Gráfico del rendimiento de la versión V7 en función de step en la máquina Volta.

Gracias a estas nuevas mejoras, hemos conseguido incrementar el rendimiento hasta alcanzar un valor cercano a los 6000 GFLOPS en la máquina Quadro (Figura 3.27) y superando los 9000 GFLOPS en la máquina Volta (Figura 3.28). Lo más destacable de estos primeros gráficos es que el valor óptimo de *elements_per_thread* es de 4 ó de 8 dependiendo de los valores de los otros parámetros, mientras que si se incrementa a 16 se obtiene siempre una bajada desmesurada en el rendimiento. Los casos en los que incrementarlo a 8 ya supone una pronunciada caída son en su mayoría aquellos en los que *num_threads* es 32. En lo que respecta a las combinaciones óptimas de parámetros, el mejor *elements_per_thread* para la máquina Quadro es 8, mientras que en la máquina Volta es 4.

Los gráficos de las Figuras 3.29 y 3.30 muestran que claramente el mejor valor para el parámetro *num_threads* en ambas máquinas es 16. Además, corrobora que los casos en los que incrementarlo a 32 supone una gran bajada en el rendimiento se corresponden con los que tienen un *elements_per_thread* grande.

Las Figuras 3.31 y 3.32 son más difíciles de interpretar que las anteriores, pero se puede ver que los mejores resultados se dan con un *step* de 32 ó 16, empeorando en casi todos los casos al incrementarlo a 64. Hay muchas combinaciones de los otros dos parámetros, especialmente las que no son muy eficientes, en las cuales el valor de *step* apenas afecta a los resultados. Por tanto, este parámetro, a diferencia de los otros dos, no siempre incide de forma notable en el rendimiento, sino únicamente en algunas combinaciones de los otros parámetros. La combinación óptima de parámetros en la máquina Quadro es (8, 16, 32), mientras que en la máquina Volta es (4, 16, 32).

Versión 8

En esta versión (Figura 3.33) se vectoriza la carga de las matrices A y B en las memorias compartidas AA y BB. Esto se realiza aplicando un *vectorize* al eje más interno de dichas matrices (líneas 25 y 35), tras haberlo previamente particionado en factor 4 (líneas 21 y 31). Con esto se puede sacar partido a las capacidades vectoriales de los procesadores modernos, que incorporan registros vectoriales, los cuales permiten realizar varios cómputos de forma simultánea.

```
1  # Creamos el schedule
2
3  # Desginamos la jerarquia de memoria
4
5  # Establecemos las opciones de tiling
6
7  # Obtenemos los indices de thread de GPU
8
9  # Dividimos las cargas de trabajo
10
11 # Asignamos las variables de iteracion a los indices de thread de GPU
12
13 # Schedule de la memoria de escritura local CL
14
15 # Unimos el computo a las variables de iteracion
16
17 # Schedule para la carga en la memoria compartida AA
18 mi, ni = s[AA].op.axis
19 ty, mi = s[AA].split(mi, nparts=num_thread)
20 tx, ni = s[AA].split(ni, nparts=num_thread)
21 _, ni = s[AA].split(ni, factor=4)
22 s[AA].reorder(ty, tx, mi, ni)
23 s[AA].bind(ty, thread_y)
24 s[AA].bind(tx, thread_x)
25 s[AA].vectorize(ni)  # vectorizamos la carga en memoria
26
27 # Schedule para la carga en la memoria compartida BB
28 mi, ni = s[BB].op.axis
29 ty, mi = s[BB].split(mi, nparts=num_thread)
30 tx, ni = s[BB].split(ni, nparts=num_thread)
31 _, ni = s[BB].split(ni, factor=4)
32 s[BB].reorder(ty, tx, mi, ni)
33 s[BB].bind(ty, thread_y)
34 s[BB].bind(tx, thread_x)
35 s[BB].vectorize(ni)  # vectorizamos la carga en memoria
```

Figura 3.33: Schedule de la Versión 8.

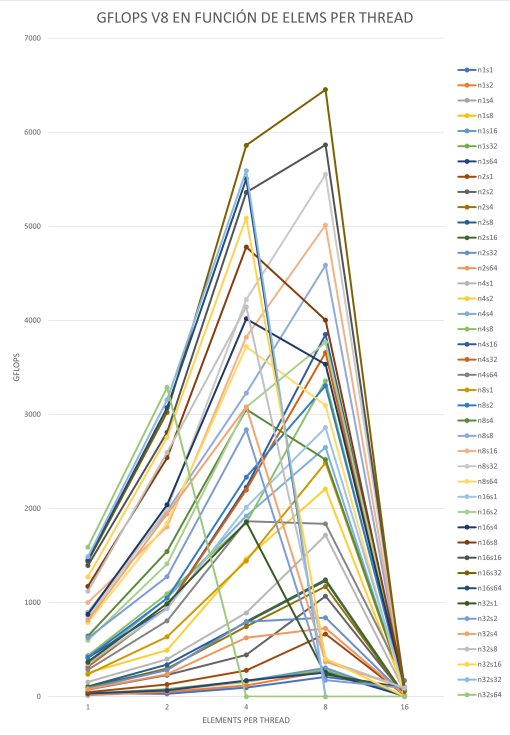

Figura 3.34: Gráfico del rendimiento de la versión V8 en función de elements per thread en la máquina Quadro.

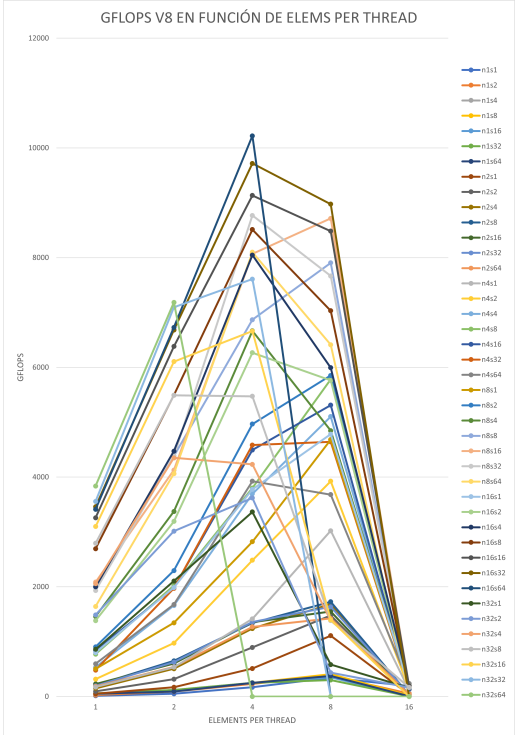

Figura 3.35: Gráfico del rendimiento de la versión V8 en función de elements per thread en la máquina Volta.

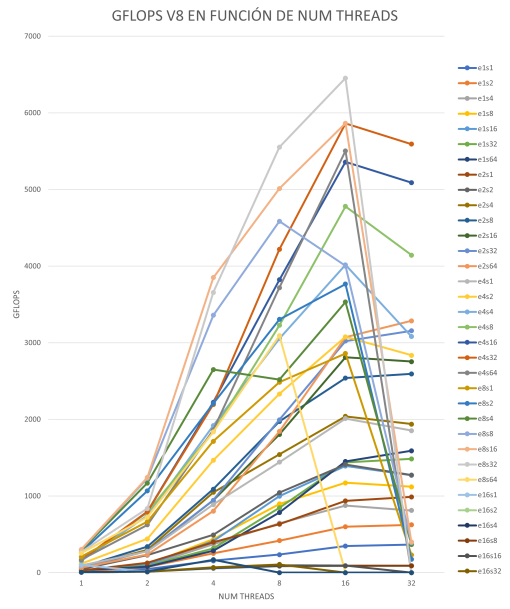

Figura 3.36: Gráfico del rendimiento de la versión V8 en función de num threads en la máquina Quadro.

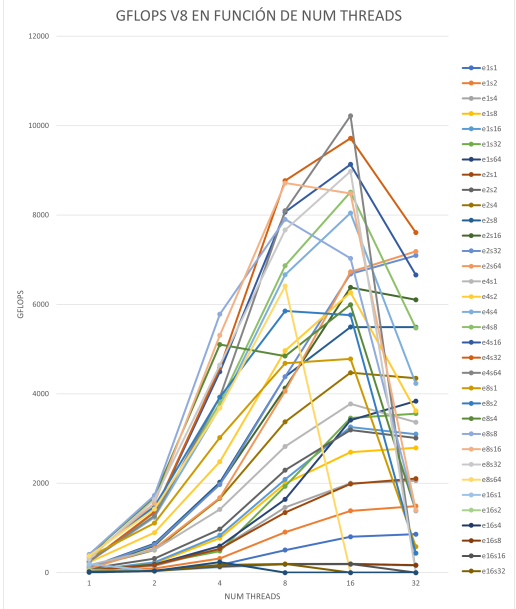

Figura 3.37: Gráfico del rendimiento de la versión V8 en función de num threads en la máquina Volta.

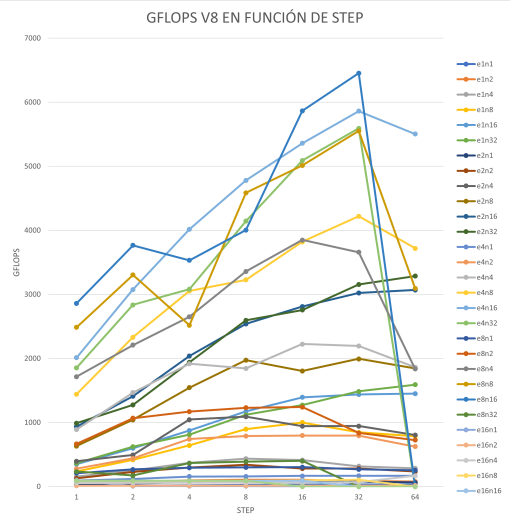

Figura 3.38: Gráfico del rendimiento de la versión V8 en función de step en la máquina Quadro.

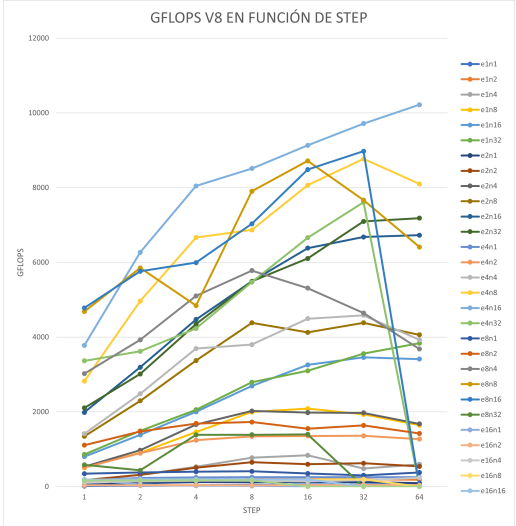

Figura 3.39: Gráfico del rendimiento de la versión V8 en función de step en la máquina Volta.

Gracias a esta vectorización, se ha logrado incrementar el rendimiento hasta alcanzar los 6400 GFLOPS en la máquina Quadro (Figuras 3.34, 3.36 y 3.38) y los 10000 GFLOPS en la máquina Volta (Figuras 3.35, 3.37 y 3.39), siendo por tanto un cambio menos pronunciado que los de las mejoras incluidas anteriormente, pero apreciable y que justifica su implementación.

En la máquina Quadro no hay muchas diferencias en lo que respecta a cómo afectan al rendimiento las distintas combinaciones de los 3 parámetros, conservándose como combinación más eficiente (8, 16, 32). Tan sólo se podría destacar que la diferencia entre emplear un *num_threads* de 16 y de 8 se reduce, y que 32 se impone más claramente como valor óptimo de *step*.

Por otra parte, en la máquina Volta la combinación óptima de parámetros pasa a ser (4, 16, 64), y se puede apreciar que es el único caso en el que se puede incrementar el *step* hasta el valor máximo permitido (pues un valor mayor supondría superar el límite máximo de memoria compartida) sin que se produzca una disminución en el rendimiento. El comportamiento de los otros dos parámetros sí es similar al de la versión anterior.

Versión 9

A continuación (Figura 3.40) se incorporan unas memorias locales de lectura AL y BL, para las matrices A y B. Con esto se establece un segundo nivel de "cache", de tamaño *elements_per_thread*, donde se irán almacenando los datos procedentes de las memorias globales AA y BB, accediendo por tanto los threads a una memoria de lectura aún más rápida que la proporcionada por AA y BB. Para esto se definen AL y BL como memorias cache de lectura de ámbito local para las memorias AA y BB respectivamente (líneas 7-8). El cómputo de AL y BL se ubicará en el eje más interno del eje de reducción de CL (líneas 23-24).

```
# Creamos el schedule

# Desginamos la jerarquia de memoria
AA = s.cache_read(A, "shared", [C])
BB = s.cache_read(B, "shared", [C])
AL = s.cache_read(AA, "local", [C])
BL = s.cache_read(BB, "local", [C])
CL = s.cache_write(C, "local")

# Establecemos las opciones de tiling

# Obtenemos los indices de thread de GPU

# Dividimos las cargas de trabajo

# Asignamos las variables de iteracion a los indices de thread de GPU

# Schedule de la memoria de escritura local CL

# Unimos el computo a las variables de iteracion
s[AA].compute_at(s[CL], rko)
s[BB].compute_at(s[CL], rko)
s[AL].compute_at(s[CL], rki)
s[BL].compute_at(s[CL], rki)

# Schedule para la carga en la memoria compartida AA

# Schedule para la carga en la memoria compartida BB
```

Figura 3.40: Schedule de la Versión 9.

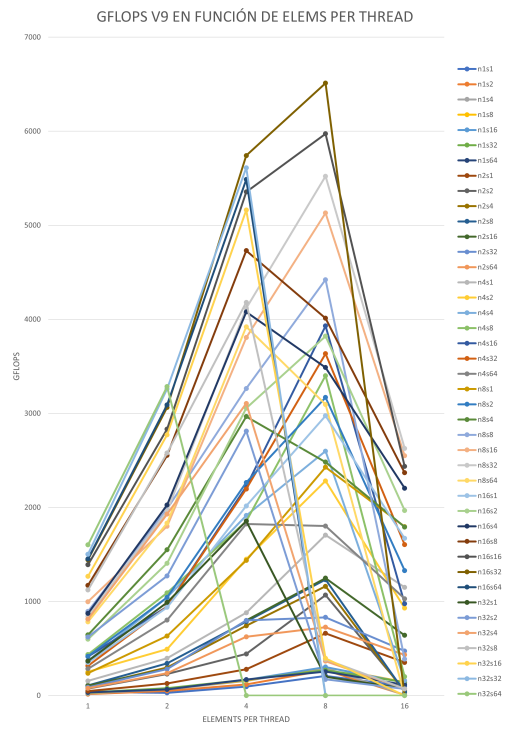

Figura 3.41: Gráfico del rendimiento de la versión V9 en función de elements per thread en la máquina Quadro.

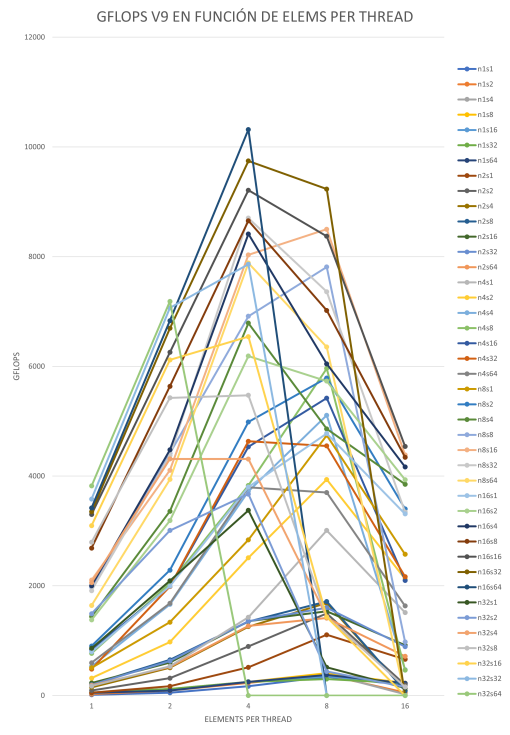

Figura 3.42: Gráfico del rendimiento de la versión V9 en función de elements per thread en la máquina Volta.

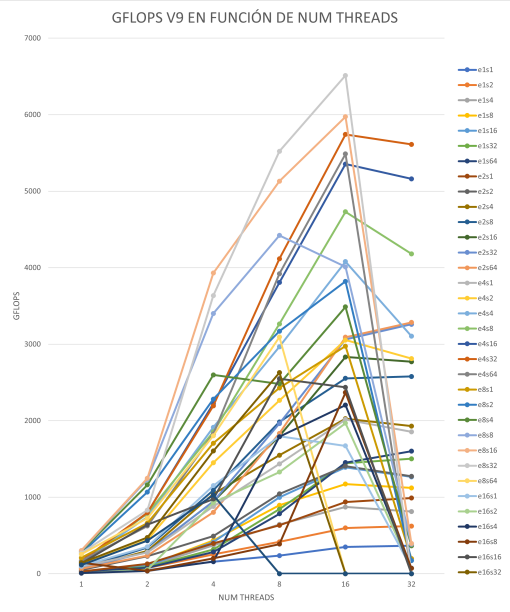

Figura 3.43: Gráfico del rendimiento de la versión V9 en función de num threads en la máquina Quadro.

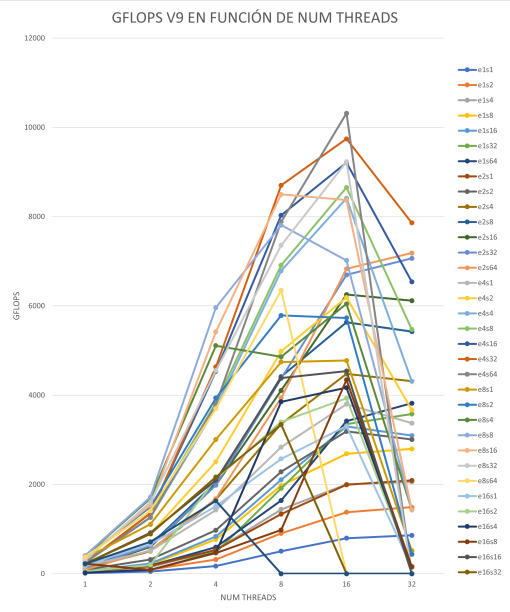

Figura 3.44: Gráfico del rendimiento de la versión V9 en función de num threads en la máquina Volta.

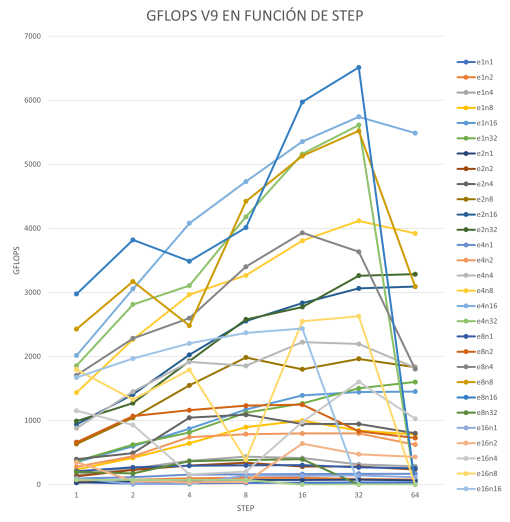

Figura 3.45: Gráfico del rendimiento de la versión V9 en función de step en la máquina Quadro.

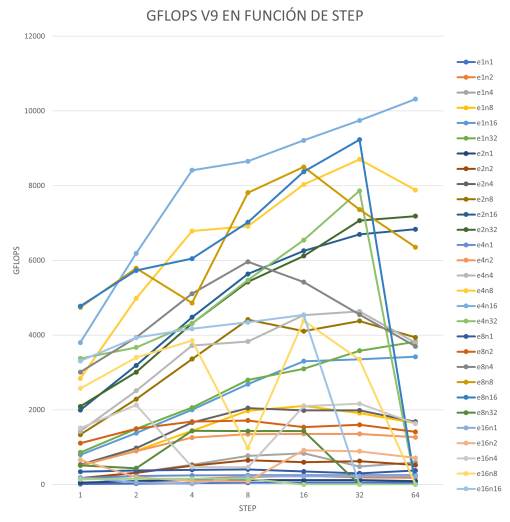

Figura 3.46: Gráfico del rendimiento de la versión V9 en función de step en la máquina Volta.

Los gráficos (Figuras 3.41, 3.42 , 3.43, 3.44, 3.45 y 3.46) muestran que claramente se trata de la mejora menos apreciable, con un incremento de menos de 100 GFLOPS respecto de la versión anterior en ambas máquinas. Lo único que sí cambia enormemente (aunque no forma parte de las combinaciones de parámetros que dan un mejor resultado, por lo que no sería muy relevante) es que con esta versión se consigue evitar que el rendimiento caiga en picado al incrementar *elements_per_thread* a 16, habiendo un descenso mucho menor.

Versión 10

Se ha realizado un intento de vectorizar todos los bucles posibles (Figura 3.47), con el fin de obtener un mayor rendimiento. Para ello, se ha aplicado vectorización al bucle más interno de C (línea 19), al bucle más interno de CL (líneas 31 y 33) y a las memorias AL y BL (líneas 38-43), en las cuales hubo que aplicar dicha vectorización al bucle externo y al interno respectivamente. Finalmente, se aplicaron *unroll* para desenrollar los bucles (líneas 18, 30, 32, 40), lo que puede aumentar el rendimiento en algunos casos [54].

```
1  # Creamos el schedule
2
3  # Desginamos la jerarquia de memoria
4
5  # Establecemos las opciones de tiling
6
7  # Obtenemos los indices de thread de GPU
8
9  # Dividimos las cargas de trabajo
10
11 # Asignamos las variables de iteracion a los indices de thread de GPU
12 s[C].bind(by, block_y)
13 s[C].bind(bx, block_x)
14
15 ty, mi = s[C].split(mi, nparts=num_thread)
16 tx, ni = s[C].split(ni, nparts=num_thread)
17 s[C].reorder(by, bx, ty, tx, mi, ni)
18 s[C].unroll(mi)
19 s[C].vectorize(ni)
20
21 s[C].bind(ty, thread_y)
22 s[C].bind(tx, thread_x)
23
24 # Schedule de la memoria de escritura local CL
25 s[CL].compute_at(s[C], tx)
26 mi, ni = s[CL].op.axis
27 rk, = s[CL].op.reduce_axis
28 rko, rki = s[CL].split(rk, factor=step)
29 s[CL].reorder(rko, rki, mi, ni)
30 s[CL].unroll(mi)
31 xo, ni = s[CL].split(ni, factor=4)
32 s[CL].unroll(xo)
33 s[CL].vectorize(ni)
34
35 # Unimos el computo a las variables de iteracion
36
37 # Optimizamos AL y BL
38 mi, ni = s[AL].op.axis
39 xo, mi = s[AL].split(mi, factor=4)
40 s[AL].unroll(xo)
41 s[AL].vectorize(mi)
42 mi, ni = s[BL].op.axis
43 s[BL].vectorize(ni)
44
45 # Schedule para la carga en la memoria compartida AA
46
47 # Schedule para la carga en la memoria compartida BB
```

Figura 3.47: Schedule de la Versión 10.

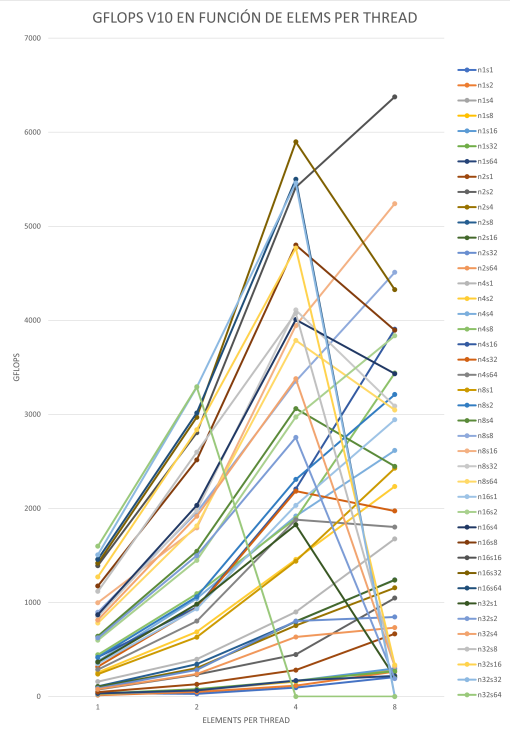

Figura 3.48: Gráfico del rendimiento de la versión V10 en función de elements per thread en la máquina Quadro.

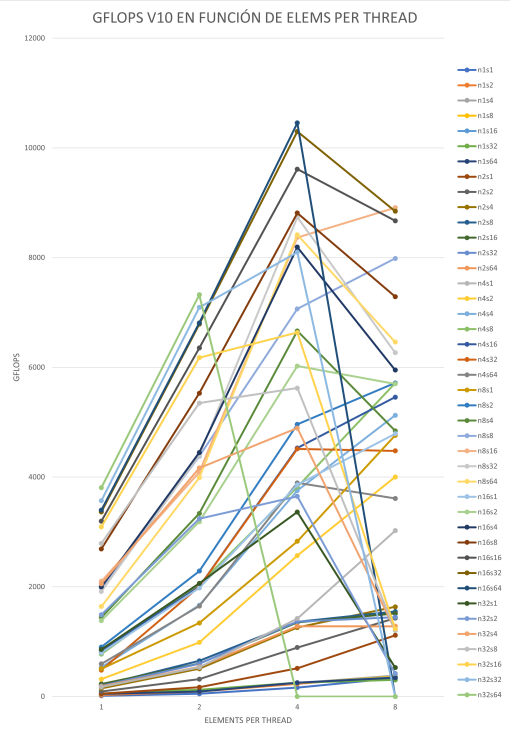

Figura 3.49: Gráfico del rendimiento de la versión V10 en función de elements per thread en la máquina Volta.

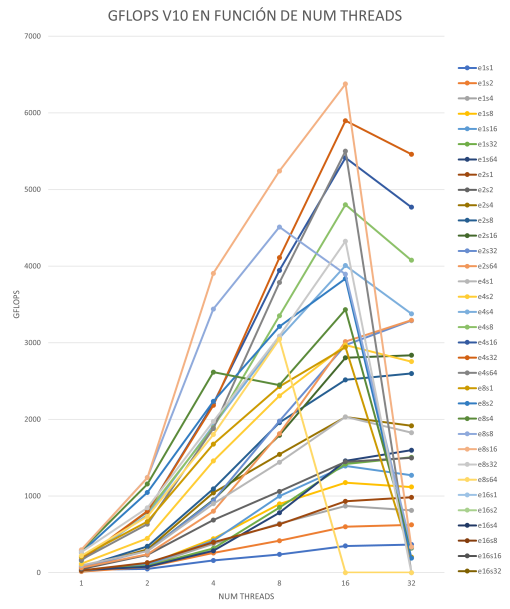

Figura 3.50: Gráfico del rendimiento de la versión V10 en función de num threads en la máquina Quadro.

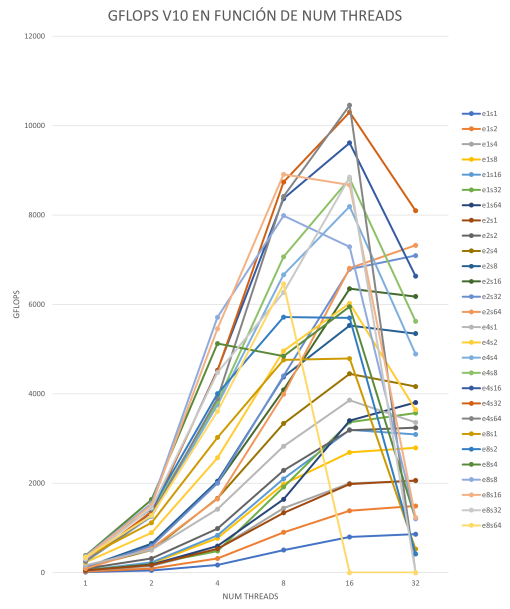

Figura 3.51: Gráfico del rendimiento de la versión V10 en función de num threads en la máquina Volta.

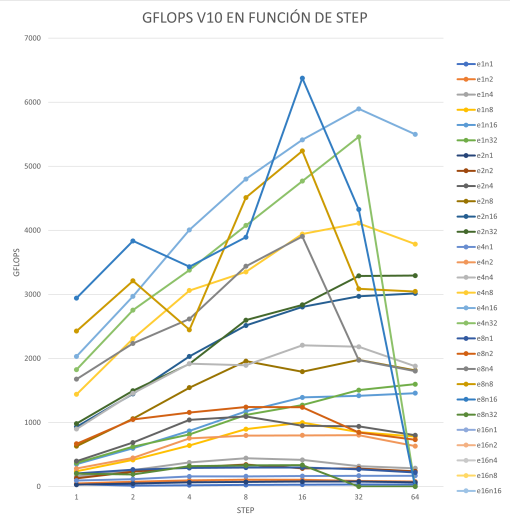

Figura 3.52: Gráfico del rendimiento de la versión V10 en función de step en la máquina Quadro.

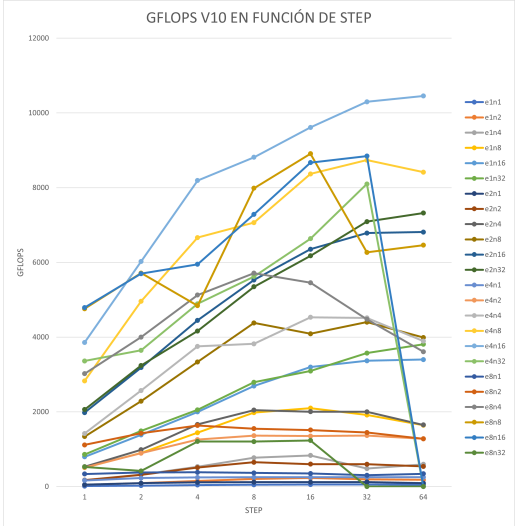

Figura 3.53: Gráfico del rendimiento de la versión V10 en función de step en la máquina Volta.

En la máquina Quadro (Figuras 3.48, 3.50 y 3.52), la implementación de esta versión no ha permitido incrementar el rendimiento respecto de la versión anterior. En esta versión la combinación de parámetros más eficiente, que pasa a ser (8, 16, 16), alcanza un valor en torno a los 6400 GFLOPS (ligeramente por debajo de V9 y similar a V8). En cambio, la combinación óptima de las versiones anteriores, (8, 16, 32), se queda en sólo 4300 GFLOPS para esta versión.

Por otra parte, esta mejora sí ha conseguido obtener más prestaciones en la máquina Volta (Figuras 3.49, 3.51 y 3.53), en la cual los parámetros se comportan de forma casi idéntica a como lo hacen en la versión V9, manteniéndose la combinación óptima de (4, 16, 64). De todas formas, este incremento en el rendimiento es muy reducido, del orden de 100 GFLOPS. El único cambio apreciable es que en esta versión el rendimiento se ve afectado en menor medida por los incrementos en el parámetro *step*.

Versión Final

Esta última optimización va encaminada a mitigar un problema conocido como *bank conflict*, el cual puede ocasionar mermas en el rendimiento pese a todas las mejoras que han sido implementadas. Con el *scheduling* empleado en las versiones anteriores, todos los threads leen las memorias AA y BB (memorias compartidas de lectura para las matrices A y B, respectivamente) de forma simultánea, lo cual puede disminuir las prestaciones debido a este *bank conflict* [55].

Pese a que un bloque de threads cuenta con $num\_threads \times num\_threads$, debido a las limitaciones en los recursos no es posible ejecutar todos ellos al mismo tiempo, sino que el Streaming Multiprocessor selecciona grupos de threads, los cuales ejecuta de forma simultánea, y va alternando rápidamente entre dichos grupos. Cada uno de estos grupos es conocido como *warp*, y consta de 32 threads con un id consecutivo.

Cada thread perteneciente a un mismo *warp* puede acceder a los datos situados en la memoria compartida de forma simultánea, por lo que ésta ha sido diseñada para soportar la carga y almacenamiento paralelos. La unidad básica de la memoria compartida es la palabra (*word*). Cada palabra consta de 4 bytes, que pueden almacenar un número en coma flotante de 32 bits. Estas palabras están agrupadas en 32 módulos conocidos

como *bancos*, y éstos están organizados de tal forma que las palabras consecutivas están asignadas a bancos consecutivos (la palabra j está en el banco i si $j\,\%32 == i$).

Cada uno de estos bancos únicamente puede atender una sola petición al mismo tiempo, y los 32 bancos son procesados en paralelo, por lo que la forma de obtener una mayor velocidad de uso de la memoria consiste en que cada banco reciba una petición de un único thread del *warp*, pudiendo así accederse a 32 palabras al mismo tiempo. En caso contrario, es decir, cuando haya dos threads que demanden datos de un mismo banco, habrá que serializar ambas peticiones, lo que se conoce como *bank conflict* y decrementa el ancho de banda efectivo. Una excepción es el caso especial en el que todos (o algunos) los threads de un *warp* solicitan la misma palabra de un banco, en este caso la palabra se leerá una única vez y se retransmitirá (*broadcast*) a cada thread, por lo que no se producirá *bank conflict*.

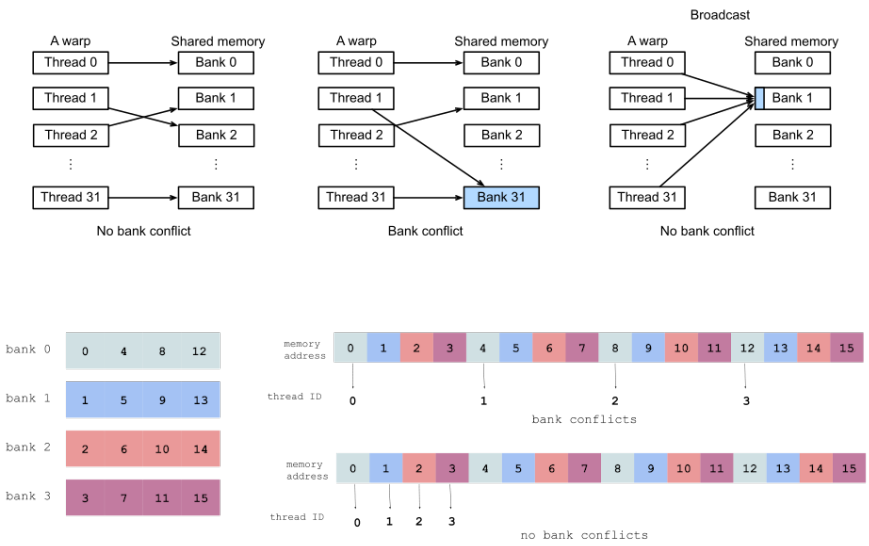

Figura 3.54: Accediendo a la memoria compartida, ejemplo de un bank conflict.

Para solventar el *bank conflict* de la memoria compartida, TVM proporciona un mecanismo conocido como *virtual thread*, que permite implementar un patrón de acceso a datos con *stride* por parte de los threads. El uso de *virtual threads* permite dividir los datos en partes (*chunks*) que se asignan a cada uno de los *virtual threads*, y cada uno de esos *chunks* se divide en las partes que se asignan a los threads de CUDA (en la práctica, los threads de CUDA en la posición i de cada *virtual thread* son fusionados en un único thread de CUDA). Esto permite que los threads accedan a datos no consecutivos, evitando el *bank conflict* [55].

Para la implementación de esta mejora (Figura 3.56), se ha decidido utilizar un total de 2 *virtual threads* por cada coordenada de threads (línea 10). Se definen los *virtual threads* xz e yz (líneas 17-18), y se realiza un *split* (líneas 26-27) sobre los ejes de cómputo de C y el bucle más externo de ellos será posteriormente asignado usando *bind* a los *virtual threads* (líneas 34-35). Mediante un *reorder* (línea 30) se sitúan los bucles correspondientes a los *virtual threads* entre los bucles correspondientes a los bloques de threads (by y bx) y los correspondientes a los threads (ty y tx).

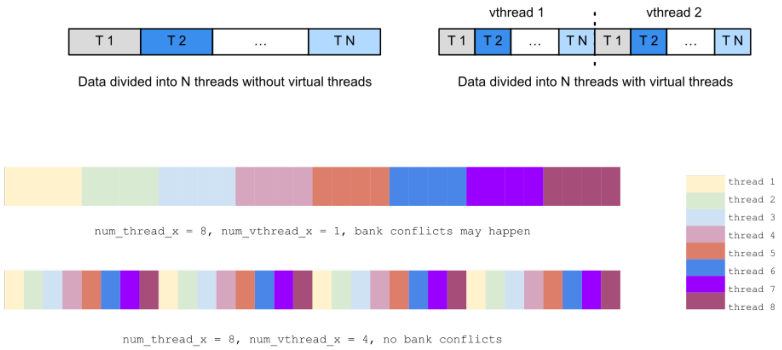

Figura 3.55: Evitando el bank conflict mediante el uso de virtual threads de TVM.

```
# Creamos el schedule

# Desginamos la jerarquia de memoria

# Establecemos las opciones de tiling
elements_per_thread = 8
num_thread = 8
block_factor = elements_per_thread * num_thread
step = 8
vthread = 2

# Obtenemos los indices de thread de GPU
block_x = te.thread_axis("blockIdx.x")
block_y = te.thread_axis("blockIdx.y")
thread_x = te.thread_axis((0, num_thread), "threadIdx.x")
thread_y = te.thread_axis((0, num_thread), "threadIdx.y")
thread_xz = te.thread_axis((0, vthread), "vthread", name="vx")
thread_yz = te.thread_axis((0, vthread), "vthread", name="vy")

# Dividimos las cargas de trabajo

# Asignamos las variables de iteracion a los indices de thread de GPU
s[C].bind(by, block_y)
s[C].bind(bx, block_x)

tyz, mi = s[C].split(mi, nparts=vthread)  # virtual thread split
txz, ni = s[C].split(ni, nparts=vthread)  # virtual thread split
ty, mi = s[C].split(mi, nparts=num_thread)
tx, ni = s[C].split(ni, nparts=num_thread)
s[C].reorder(by, bx, tyz, txz, ty, tx, mi, ni)
s[C].unroll(mi)
s[C].vectorize(ni)

s[C].bind(tyz, thread_yz)
s[C].bind(txz, thread_xz)
s[C].bind(ty, thread_y)
s[C].bind(tx, thread_x)

# Schedule de la memoria de escritura local CL

# Unimos el computo a las variables de iteracion

# Optimizamos AL y BL

# Schedule para la carga en la memoria compartida AA

# Schedule para la carga en la memoria compartida BB
```

Figura 3.56: Schedule de la Versión final.

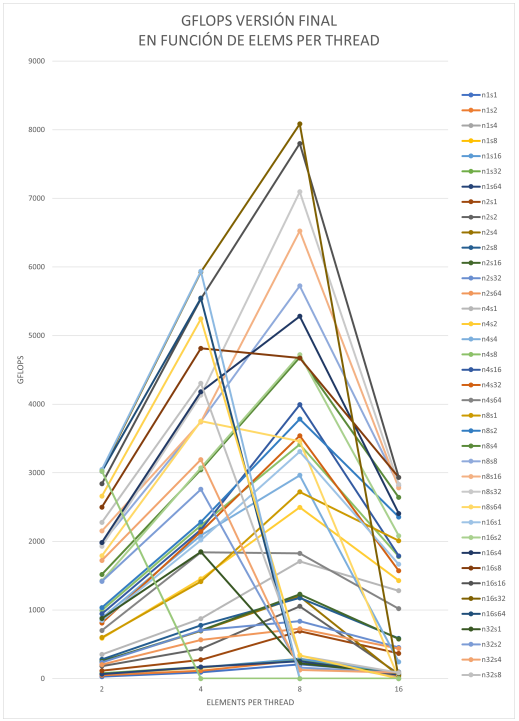

Figura 3.57: Gráfico del rendimiento de la versión VFINAL en función de elements per thread en la máquina Quadro.

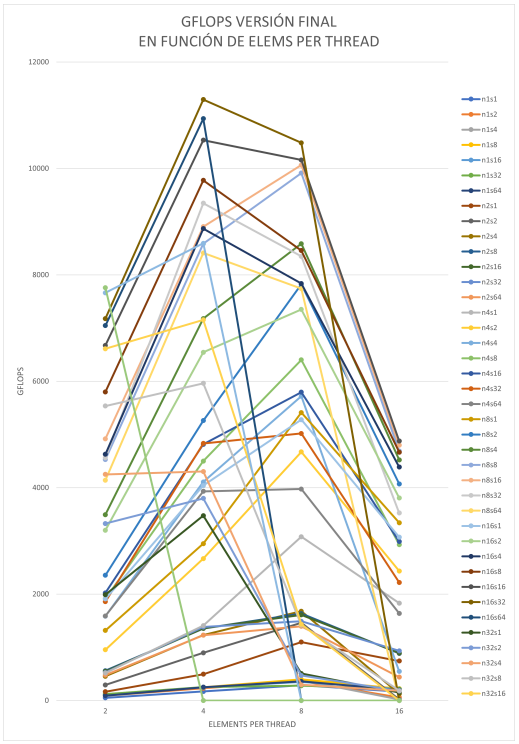

Figura 3.58: Gráfico del rendimiento de la versión VFINAL en función de elements per thread en la máquina Volta.

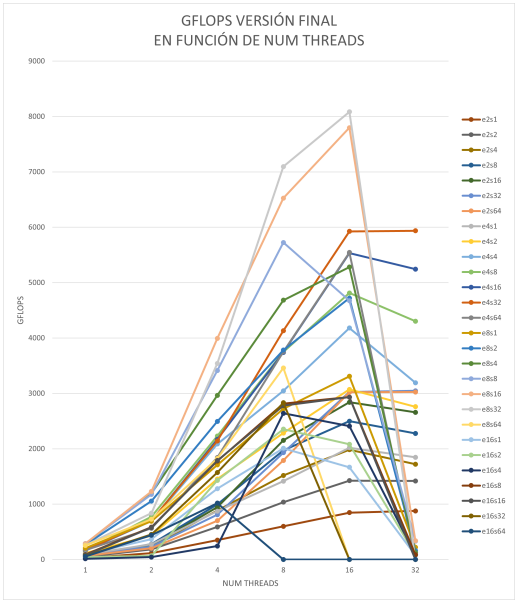

Figura 3.59: Gráfico del rendimiento de la versión VFINAL en función de num threads en la máquina Quadro.

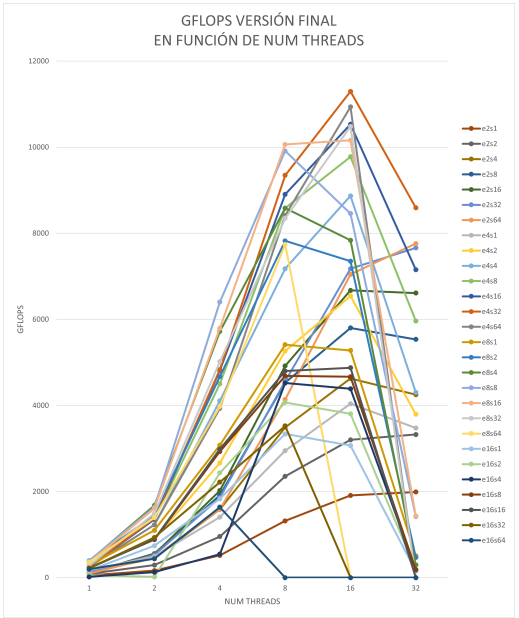

Figura 3.60: Gráfico del rendimiento de la versión VFINAL en función de num threads en la máquina Volta.

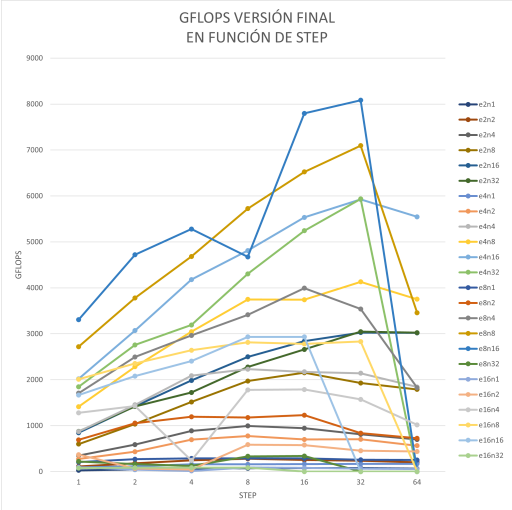

Figura 3.61: Gráfico del rendimiento de la versión VFINAL en función de step en la máquina Quadro.

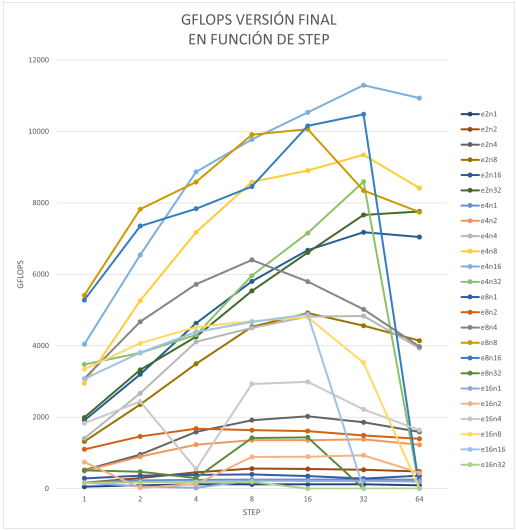

Figura 3.62: Gráfico del rendimiento de la versión VFINAL en función de step en la máquina Volta.

Con esta última modificación se ha conseguido obtener un rendimiento final de 8000 GFLOPS en la máquina Quadro (Figuras 3.57, 3.59 y 3.61) y de más de 11000 GFLOPS en la máquina Volta (Figuras 3.58, 3.60 y 3.62), siendo una mejora mucho más apreciable que las últimas que se realizaron. La combinación óptima de parámetros en la máquina Quadro es (8, 16, 32) y en la máquina Volta (4, 16, 32). Los gráficos de esta versión en la máquina Quadro son muy similares a los de V9, no sucediendo la pérdida de rendimiento ocasionada al implementar la vectorización en V10. Por otra parte en la máquina Volta los parámetros se comportan de una forma más similar a la versión anterior. Es destacable que en la máquina Quadro existe una gran diferencia entre las 4 mejores combinaciones de parámetros y el resto de ejecuciones, cosa que no sucede en la máquina Volta, donde hay más ejecuciones que muestran un buen rendimiento.

3.3 Resumen de las versiones

Los gráficos de las Figuras 3.63, 3.64 realizan una comparativa de las prestaciones medidas en GFLOPS en ambas máquinas para las 11 distintas versiones que han sido desarrolladas, tomando para cada una de ellas los parámetros que le han permitido obtener un mayor rendimiento.

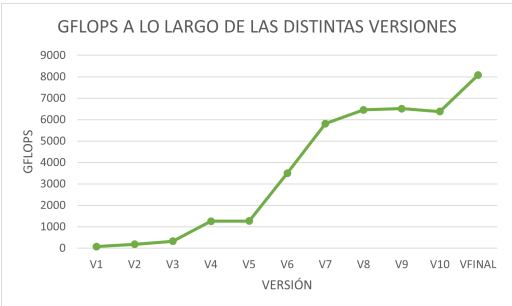

Figura 3.63: Comparativa del rendimiento a lo largo de las diferentes versiones implementadas en la máquina Quadro.

Figura 3.64: Comparativa del rendimiento a lo largo de las diferentes versiones implementadas en la máquina Volta.

Como vemos, las optimizaciones que han supuesto un mayor incremento en las prestaciones son V6 y V7, que se corresponden con la introducción de una memoria local de escritura para la matriz C, y de memorias compartidas de lectura para las matrices A y B, respectivamente. Esto permite hacerse una idea de lo crucial que es tener en cuenta el acceso a memoria en el desarrollo de algoritmos de Computación de Altas Prestaciones, pues supone un importante cuello de botella debido a que leer y escribir de memoria es sumamente más lento que realizar operaciones de coma flotante, por lo que si se pueden transferir los datos a una memoria que tenga un acceso más rápido (como pueden ser la memoria local del thread o la compartida del bloque respecto de la memoria global de la GPU) se conseguirá un rendimiento mucho mayor.

También destaca V4, que implementa una estructura de threads y bloques en 2 dimensiones, permitiendo repartir el trabajo de la matriz de forma eficiente y teniendo en cuenta que los threads contiguos operen con posiciones de memoria contiguas).

Finalmente también supone una gran mejora, especialmente en la máquina Quadro, la introducción de los *virtual threads* para paliar los *bank conflicts* que suceden al acceder a las memorias compartidas.

CAPÍTULO 4
Resultados numéricos

En el siguiente capítulo se presentan los resultados de los experimentos realizados para medir el rendimiento del código implementado y compararlo con otras implementaciones de altas prestaciones. Se realizarán pruebas tanto con las matrices empleadas en el estudio del Apéndice como con las correspondientes a un caso de estudio real de una red neuronal.

4.1 Comparativa con otras librerías

Una vez desarrollado el código en su versión final, se ha realizado una serie de pruebas para evaluar su rendimiento en función del tamaño del problema (es decir, el tamaño de las matrices a multiplicar), ya que el tamaño de 2048 que se utilizó para comparar las distintas versiones puede quedarse corto para mostrar todo el potencial de cómputo, pues es posible que el tiempo computacional dedicado a operaciones de coma flotante no suponga un porcentaje lo suficientemente elevado respecto del tiempo computacional total o que los tamaños sean demasiado reducidos como para poder aprovechar todo el potencial de las jerarquías de memoria empleadas, entre otros factores.

También se han realizado pruebas con productos de matrices rectangulares, para ver si se mantiene el rendimiento obtenido con las matrices cuadradas, así como para ver si se mantienen del mismo modo las combinaciones de los parámetros, *elements_per_thread*, *num_threads* y *step* que permitían obtener las mejores prestaciones. Todas estas pruebas están documentadas en el **Apéndice** B.

Pero para poder medir el potencial real de nuestro código, es preciso compararlo con otras implementaciones de alto rendimiento, para así poder ver hasta qué punto se está sacando el máximo partido posible a las capacidades de cómputo de la máquina.

Uno de los *benchmark* empleados será la librería *cuBLAS*, que proporciona una implementación acelerada por GPU de BLAS (Basic Linear Algebra Subprograms) usando CUDA, lo que permite aprovechar al máximo el poder de cómputo de las GPU de Nvidia para realizar las operaciones de álgebra lineal especificadas en BLAS (entre las cuales se encuentra GEMM, especificada en BLAS Nivel 3). La biblioteca cuBLAS está altamente optimizada a la arquitectura en la que se ejecuta y permite alcanzar unas cotas de rendimiento especialmente elevadas. Concretamente se usará la versión cuBLAS 10.

El otro *benchmark* que se utilizará corresponde a la utilización de la función *Tuning* que ofrece Apache TVM, la cual, como se mencionó anteriormente, permite optimizar un modelo (en este caso el cómputo de una operación GEMM) para que se ejecute más eficientemente en un objetivo dado (en este caso las GPU de Nvidia que se utilizarán). Concretamente se utilizará el *AutoScheduler*, el cual a diferencia de la otra herramienta

ofrecida por TVM (*AutoTVM*), no requiere plantillas de *schedule* predefinidas, sino que genera un espacio de búsqueda automáticamente analizando la definición del cómputo y busca el mejor *schedule* en dicho espacio.

Al AutoScheduler se le pasan como parámetros la función donde se definen las *Tensor Expressions* correspondientes a la operación GEMM, los argumentos de dicha función (que se corresponden con los tamaños de matriz empleados) y la especificación del dispositivo (*target*).

Dentro de las opciones (*tune_options*), especificaremos un *num_measure_trials* de 1000. Este parámetro controla el número de intentos que se realizarán. La política de búsqueda mide tantos schedule como se indique en éste y devuelve el mejor de todos ellos. La opción *early_stopping*, que detiene el proceso de tuning si no se ha producido una mejora tras un número de mediciones configurable, no será activada.

En la Figura 4.1 se muestra el código Python correspondiente al uso del AutoScheduler para realizar el proceso de *tuning*.

```
os.environ["TVM_NUM_THREADS"] = str(num_threads)
# Creamos la tarea de busqueda
task = tvm.auto_scheduler.SearchTask(func=auto_tune_gemm, args=(M,N,K),
    target=target)

# Ajustamos los parametros del auto-scheduler
# num_measure_trials es el numero de intentos de medida que podemos
    utilizar durante la busqueda
# Usamos :code:‘RecordToFile‘ para exportar los registros de las medidas a
    un fichero.
log_file = "gemm_{}_{}_{}.json".format(M,N,K)
if os.path.exists('tuned/') == False:
    os.mkdir('tuned')
mypath='tuned/{}'.format(log_file)
if os.path.exists(mypath) == True:
    print("Skipping autotune because it already exists!")
else:
    tune_option = None
    measure_ctx = None

    tune_option = auto_scheduler.TuningOptions(
        num_measure_trials=1000,
        measure_callbacks=[auto_scheduler.RecordToFile(mypath)],
        verbose=1,
    )

    # Ejecutar el auto-tuning (busqueda)
    task.tune(tune_option)
# Aplicar el mejor schedule
sch, args = task.apply_best(mypath)
print(task.print_best(mypath))
print("==============")
m = tvm.lower(sch, args, simple_mode=True)
print(m)
print("==============")

#Construir la funcion a partir del schedule
func = tvm.build(sch, args, target, name="gemm")
```

Figura 4.1: Empleo del AutoScheduler de TVM para optimizar un modelo.

Los gráficos de las Figuras 4.2 y 4.3 ilustran los resultados de la comparativa de rendimiento medido en GFLOPS para estas tres implementaciones de la operación GEMM (cuBLAS, AutoScheduler y nuestro código) con distintos tamaños de problema, empleando tanto matrices cuadradas como rectangulares.

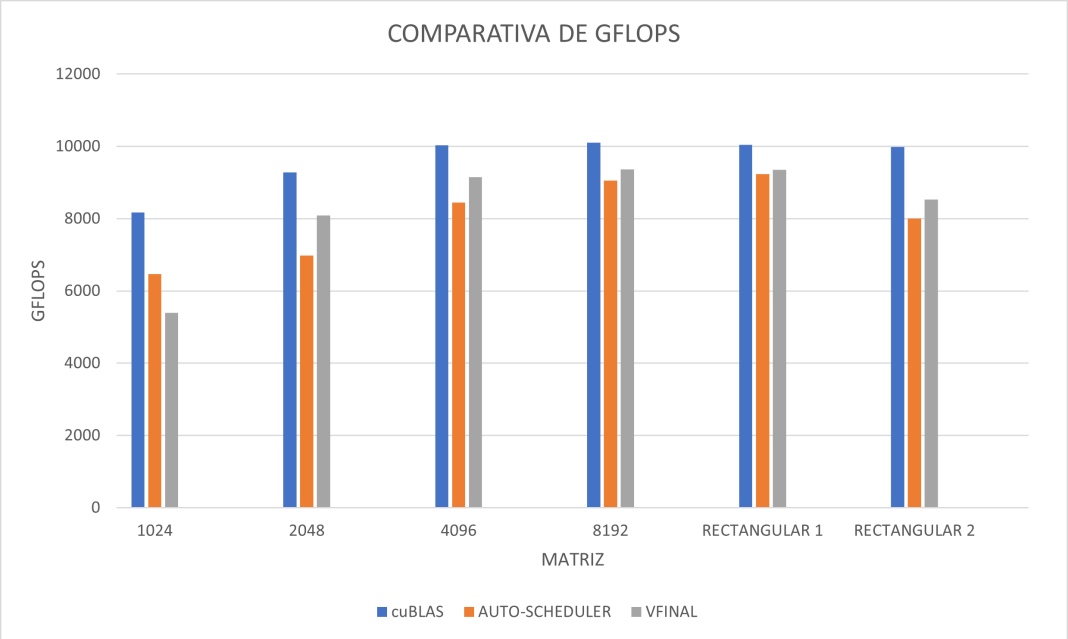

Figura 4.2: Gráfico comparativo de rendimiento de GEMM usando la librería cuBLAS, la función AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro.

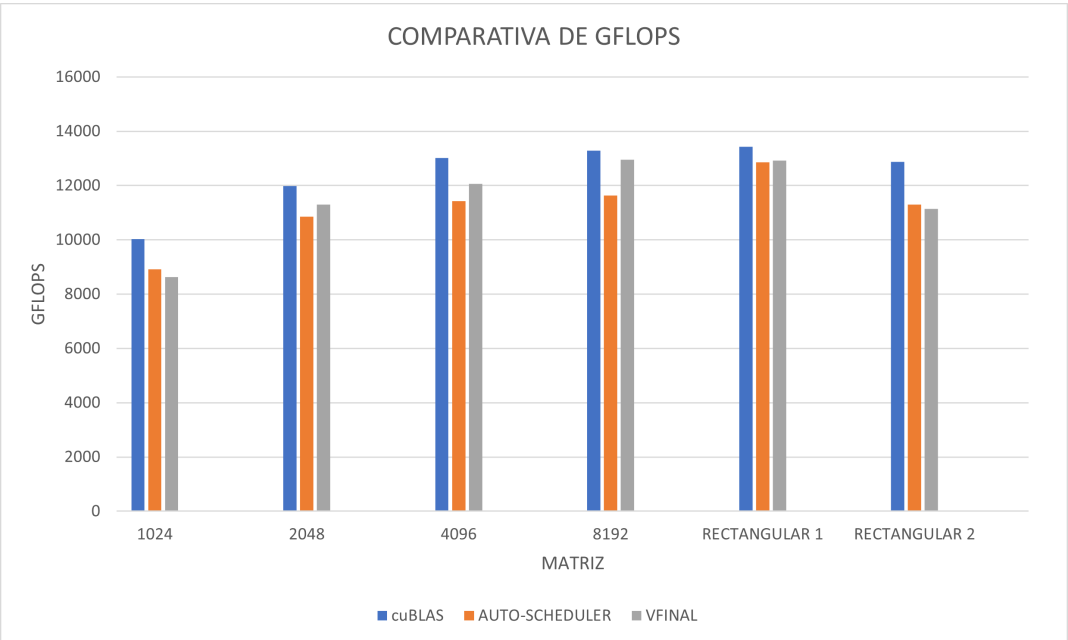

Figura 4.3: Gráfico comparativo de rendimiento de GEMM usando la librería cuBLAS, la función AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta.

Ni el código implementado en este trabajo ni la función AutoScheduler de TVM han sido capaces de desbancar a cuBLAS, lo cual era de esperar, ya que esta librería utiliza código optimizado específicamente para la arquitectura en la cual se está ejecutando. Sin embargo, los resultados de rendimiento obtenidos por nuestro algoritmo son muy positivos, consiguiendo acercarse bastante a los GFLOPS obtenidos por cuBLAS e incluso superando al AutoScheduler en la mayoría de los casos. La distancia entre cuBLAS y nuestro código se reduce conforme se incrementa el tamaño del problema, siendo además el caso de las matrices más pequeñas (tamaño de 1024) el único en el que no se ha conseguido aventajar al AutoScheduler.

Es especialmente llamativo que el AutoScheduler ha obtenido su mejor rendimiento en el caso del problema "rectangular 1" ($M = 8192$, $N = 8192$, $K = 2048$), a diferencia de los otros dos, que lo consiguieron en las matrices cuadradas de 8192. También cabe destacar que en el problema "rectangular 2" ($M = 2048, N = 2048, K = 8192$) nuestra implementación y el AutoScheduler presentan una caída de prestaciones mucho mayor a la observada en cuBLAS.

De estos datos puede inferirse que tanto el AutoScheduler de TVM como el código implementado son especialmente eficientes en los casos de matrices muy grandes y rectangulares (teniendo además nuestro código unos resultados ligeramente mejores que los del AutoScheduler en casi todos los casos), acercándose mucho a la implementación de altas prestaciones cuBLAS, y no pudiéndose aprovechar todo su potencial en las operaciones GEMM de matrices de tamaño demasiado reducido, donde se alejan bastante de los resultados de cuBLAS.

Estos resultados prueban que nuestro código podría tener un buen desempeño ejecutando modelos de Deep Learning, en los cuales la mayor parte del tiempo computacional se invierte en realizar productos de matrices muy rectangulares [56].

4.2 Caso de estudio: ResNet50-v1.5

4.2.1. Descripción de la red ResNet-50v1.5

Una vez probado y medido el desempeño del código desarrollado en productos de matrices de diversos tamaños y dimensiones, es el momento de enfrentarse a un caso de uso real: las operaciones GEMM empleadas en un modelo de Deep Learning (es decir, una red neuronal). La red escogida es *ResNet-50v1.5*, la cual en su ejecución consta de 20 distintas operaciones de producto matricial, empleadas a lo largo de las 50 capas con las que cuenta la red.

La red ResNet-50v1.5 es una red ResNet (Residual Neural Network), que es una familia de Redes Neuronales Convolucionales (CNN) creada por Kaiming He et al. en 2015 en su artículo "Deep Residual Learning for Image Recognition" [57]. Fue diseñada para paliar los problemas de *desvanecimiento de gradiente* y de *gradiente explosivo* [58], los cuáles padecían las redes muy profundas (con un gran número de capas). Éstos consisten en que cuando se tiene un elevado número de capas, si los gradientes de la función de pérdida son muy pequeños/grandes, éstos disminuirán/aumentarán de forma exponencial al retropropagarse por el modelo.

La arquitectura de ResNet permite que la red tenga múltiples capas de aprendizaje sin quedarse atrapada en mínimos locales, un problema común de las redes profundas. Las redes ResNet se inspiran en el hecho biológico de que algunas neuronas se conectan con neuronas en capas no necesariamente contiguas, saltando capas intermedias. El modelo ResNet consiguió el primer puesto en la competición ILSVRC (ImageNet Large Scale Visual Recognition Challenge) 2015 en las tareas de clasificación de imágenes, detección y localización, así como en la MS COCO (Microsoft Common Objects in Context) 2015 en las categorías de detección y segmentación.

Las características clave de la arquitectura ResNet son [59]:

- *Conexiones Residuales:* ResNet incorpora "conexiones residuales", las cuales permiten entrenar redes neuronales muy profundas y aliviar el problema de *desvanecimiento de gradiente*.

- *Función identidad:* ResNet usa la función identidad como función residual, lo que permite facilitar el proceso de entrenamiento aprendiendo el mapeo residual en vez del verdadero mapeo.
- *Profundidad:* ResNet permite la implementación de redes neuronales muy profundas (de gran número de capas), lo cual permite mejorar el rendimiento en tareas de reconocimiento de imágenes.
- *Menos parámetros:* Resnet obtiene mejores resultados con un menor número de parámetros, siendo por tanto más eficiente computacionalmente.
- *Resultados state-of-the-art:* ResNet ha obtenido resultados *state-of-the-art*, (es decir, a la última y entre lo más puntero en la investigación actual) en diversas tareas de reconocimiento de imágenes y se ha convertido en un *benchmark* muy usado para este tipo de tareas.
- *Enfoque general y efectivo:* Los autores concluyen que las conexiones residuales son un enfoque general y efectivo para poder desarrollar redes más profundas.

ResNet funciona añadiendo *conexiones residuales* a la red, las cuales permiten mantener el flujo de información a lo largo de la red y evitar que los gradientes se desvanezcan. Las conexiones residuales son "atajos" que permiten que la información se salte una o más capas de la red y alcance la salida directamente. Esto permite que la red aprenda la función residual y realice pequeños ajustes en los parámetros, lo que ayuda a la red a converger más rápido y obtener un mejor rendimiento. La idea de las conexiones residuales se basa en la idea de que es más fácil aprender la *función residual*, que mapea las entradas en las salidas deseadas, en vez de intentar aprender el complejo mapeado entre las entradas y las salidas.

La red ResNet50-v1.5 es una versión modificada del modelo original ResNet50. ResNet50-v1.5 es ligeramente más precisa (0.5 %) y tiene un rendimiento ligeramente inferior (5 %) en comparación con ResNet50. Por lo tanto, ResNet50-v1.5 se emplea en aplicaciones en las cuales esa ligera mejora en la precisión es determinante y la reducción en el rendimiento no es algo crucial [60].

La diferencia entre las arquitecturas de ResNet50 y ResNet50-v1.5 es que, en los bloques *bottleneck* que requieren de un submuestreo, v1 tiene un *stride* de 2 en la primera convolución 1×1, mientras que v1.5 tiene un *stride* de 2 en la convolución de 3×3.

La Tabla 4.1 ilustra la arquitectura de las redes ResNet. La ResNet50-v1.5 se corresponde con la columna de *50-layer*, pues posee 50 capas.

Tabla 4.1: Arquitectura de las redes de la familia ResNet.

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

Se puede ver, por tanto, que la arquitectura ResNet50-v1.5 consta de:

- Una convolución con un tamaño de *kernel* de 7×7 y 64 *kernels* diferentes, todo ello con un *stride* de 2. Esto constituye una única capa.
- Un *max pooling* con un *stride* de 2.
- Tres convoluciones, la primera de 1×1 y 64 *kernels*, la segunda de 3×3 y 64 *kernels* y la tercera de 1×1 y 256 *kernels*. Estas tres capas se repiten un total de 3 veces constituyendo así 9 capas de la red.
- Convoluciones, de 1×1 y 128 *kernels*, de 3×3 y 128 *kernels* y de 1×1 y 512 *kernels*. Estas tres capas se repiten un total de 4 veces constituyendo así 12 capas.
- Convoluciones, de 1×1 y 256 *kernels*, de 3×3 y 256 *kernels* y de 1×1 y 1024 *kernels*. Estas tres capas se repiten un total de 6 veces constituyendo así 18 capas.
- Convoluciones, de 1×1 y 512 *kernels*, de 3×3 y 512 *kernels* y de 1×1 y 2048 *kernels*. Estas tres capas se repiten un total de 3 veces constituyendo así 9 capas.
- A continuación un *average pool* y para finalizar una capa *fully connected* con 1000 nodos y al final una función *softmax*, esto constituye 1 capa.

Esto finalmente supone un total de 1 + 9 + 12 + 18 + 9 + 1 = 50 capas. NOTA: No se cuentan las funciones de activación y las funciones de *pooling*.

El motivo por el que se ha decidido emplear la red ResNet50-v1.5 es debido a que es el modelo que ejecuta el *benchmark* MLPerf de MLCommons [62], un *benchmark* que mide la rapidez con la que los sistemas pueden procesar entradas y producir resultados usando un modelo entrenado. La misión de MLPerf es construir justos y útiles *benchmarks* que proporcionen evaluaciones no sesgadas del rendimiento de entrenamiento e inferencia de hardware, software y servicios [63]. El modelo escogido por el consorcio de MLPerf para la tarea de clasificación de imágenes es precisamente Resnet50-v1.5. MLPerf también establece una serie de modelos para otras tareas de Machine Learning, como pueden ser Detección de objetos - Retinanet, Segmentación de imágenes médicas - 3D UNET, Speech-to-text - RNNT, Procesamiento de lenguaje - BERT-large y Recomendación - DLRM.

Como se puede ver en la Figura 4.4, la mayor parte del tiempo computacional de la ejecución de una red neuronal profunda convolucional (como es el caso de la ResNet50-v1.5 estudiada) se invierte en las capas convolucionales y en las capas *fully-connected*.

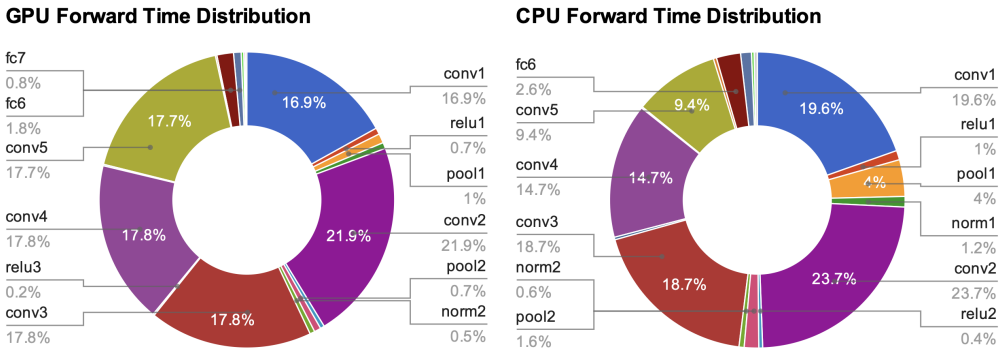

Figura 4.4: Distribución del tiempo para el modelo AlexNet con ImageNet.

Se puede ver que se emplea el 95 % del tiempo en la versión GPU, y el 89 % en la CPU en capas *conv* (convolucionales) y fc (*fully-connected*), por lo que encontrar una forma de incrementar el rendimiento de la red pasa por optimizar los cálculos realizados en dichas capas.

Las capas *fully-connected* pueden ser expresadas de forma sencilla como una operación GEMM (GEneral Matrix to Matrix Multiplication, es decir, un producto matricial general) [56]. En esta operación se toman los valores de las neuronas de entrada como una matriz de tamaño $1 \times k$ siendo k el número de neuronas de la capa de entrada, y se multiplican por la matriz de los pesos, que será de tamaño $k \times n$ siendo n el número de neuronas de la capa de salida. En esta segunda matriz, la posición $[X, Y]$ representará el peso de la conexión entre la neurona X de la capa de entrada con la neurona Y de la capa de salida. Finalmente, la salida corresponderá a una matriz de tamaño $1 \times n$, donde cada columna represente el resultado de cada neurona de la capa de salida (Figura 4.5).

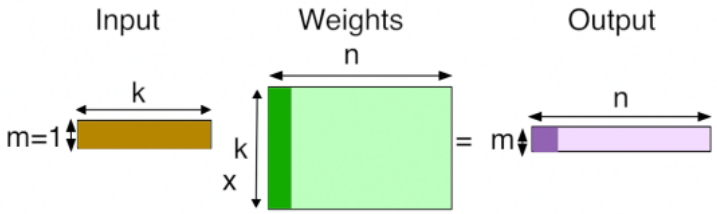

Figura 4.5: Capa fully-connected expresada como una operación GEMM.

Por otra parte, también es posible convertir las operaciones realizadas en las capas convolucionales en operaciones GEMM, aunque es más complicado [56]. Una capa convolucional trata su entrada como una imagen bidimensional con un número de canales por cada píxel, como si fuera una imagen con anchura, altura y profundidad. La operación de convolución produce su salida tomando una serie de *kernels* de pesos, y aplicándolos a lo largo de la imagen. Cada *kernel* es otro arreglo tridimensional de números, con la misma profundidad que la imagen de entrada, pero con una anchura y altura mucho menores, típicamente con valores como 7×7. Para generar un resultado, un *kernel* es aplicado a un *grid* de puntos a lo largo de la imagen de entrada. En cada punto en el que se aplica, los valores de entrada y los pesos correspondientes son multiplicados, y posteriormente sumados para producir un único valor de salida para ese punto (Figura 4.6).

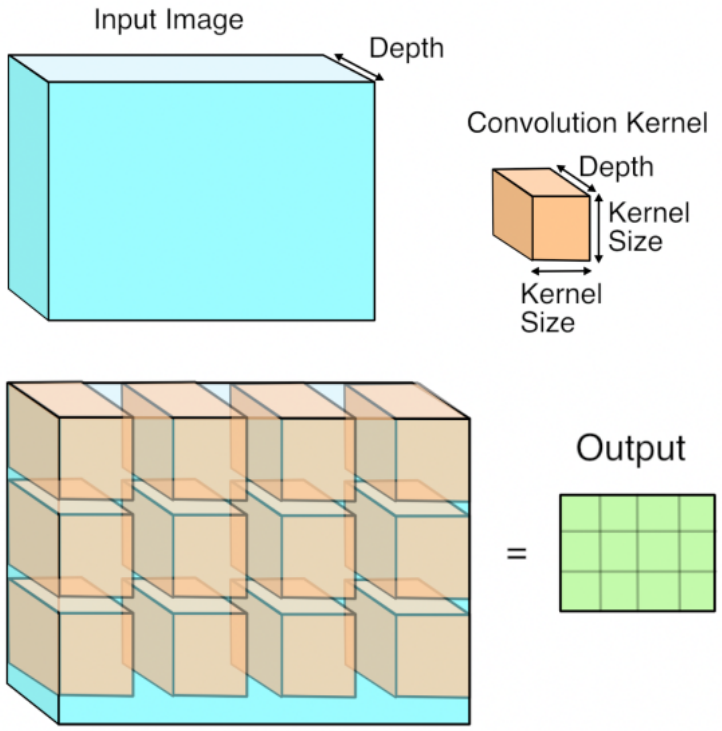

Figura 4.6: Capa convolucional expresada de forma matricial.

Para poder transformar estos cálculos en operaciones GEMM, el primer paso consiste en convertir la entrada, que es una imagen expresada como un array tridimensional, a un array bidimensional que pueda ser tratado como una matriz (Figura 4.7). Dividiremos la imagen en pequeños cubos tridimensionales (*patches* en la imagen) que se corresponden a los *grids* sobre los cuales son aplicados los *kernels*, y copiaremos cada uno de esos cubos de valores de entrada a cada una de las filas de la matriz.

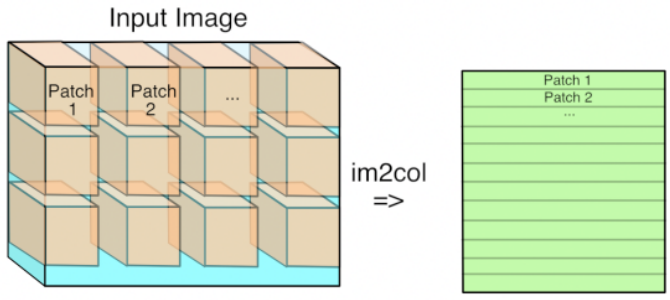

Figura 4.7: Imagen de entrada expresada como una matriz bidimensional.

A continuación se realizará el mismo procedimiento con los pesos de los *kernels*, serializando los cubos tridimensionales como filas de la segunda matriz de la operación, quedado la operación GEMM de la manera expresada en la Figura 4.8.

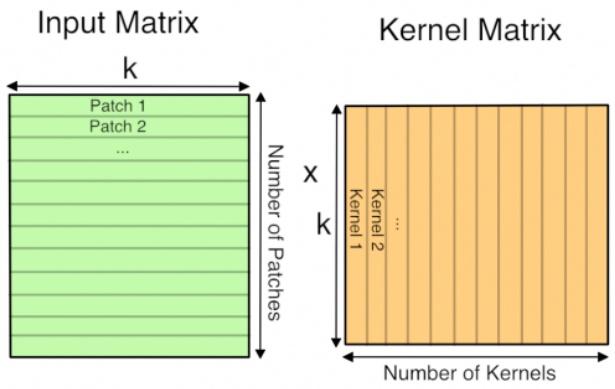

Figura 4.8: Capa convolucional expresada como una operación GEMM.

Aquí, k es el número de valores en cada *patch* y *kernel*, por lo que se corresponde con $anchura \times altura \times profundidad$ del *kernel*. La matriz resultante de la operación tendrá tantas filas como *patches* tenga la imagen y tantas columnas como *kernels* haya. Esta matriz será posteriormente tratada como un arreglo tridimensional por las siguientes operaciones, tomando el número de *kernels* como profundidad y volviendo a transformar los *patches* en filas y columnas en función de su posición original en la imagen de entrada.

Las ventajas de poder expresar las operaciones tanto de las capas *fully-connected* como de las capas convolucionales como operaciones GEMM residen en que es una operación que durante décadas ha estado en el punto de mira de los expertos en computación científica y de altas prestaciones, los cuales han desarrollado algoritmos que permiten optimizar al máximo esta clase de operación. Un ejemplo de ello es la librería BLAS (Basic Linear Algebra Subprograms), que define las rutinas estándar de facto de bajo nivel para bibliotecas de álgebra lineal.

Claramente los beneficios resultantes de estas optimizaciones y sus patrones de acceso a memoria muy regulares sobrepasan al incremento en el coste de almacenamiento (resultante de la transformación de "3D a 2D"), por lo que poder transformar la práctica totalidad del coste computacional de la red (recordemos que suponía un 89 % en CPU y un 95 % en GPU) en un cálculo que puede ser optimizado de manera muy eficiente y específica para la arquitectura de la máquina empleada es algo crucial y determinante.

La Tabla 4.1 expone los tamaños de matriz de las 20 operaciones GEMM utilizadas en la red ResNet50-v1.5, así como las capas de la red en las cuales se utiliza cada una de estas 20 operaciones.

Tabla 4.2: Operaciones GEMM de la red ResNet50-v1.5.

Layer id.	Layer numbers in ResNet50 v1.5	m	n	k	Layer id.	Layer numbers in ResNet50 v1.5	m	n	k
1	001	1,605,632	64	147	11	080	100,352	256	512
2	006	401,408	64	64	12	083/095/105/115/125/135	25,088	256	2,304
3	009/021/031	401,408	64	576	13	086/098/108/118/128/138	25,088	1,024	256
4	012/014/024/034	401,408	256	64	14	088	25,088	1,024	512
5	018/028	401,408	64	256	15	092/102/112/122/132	25,088	256	1,024
6	038	401,408	128	256	16	142	25,088	512	1,024
7	041/053/063/073	100,352	128	1,152	17	145/157/167	6,272	512	4,608
8	044/056/066/076	100,352	512	128	18	148/160/170	6,272	2,048	512
9	046	100,352	512	256	19	150	6,272	2,048	1,024
10	050/060/070	100,352	128	512	20	154/164	6,272	512	2,048

4.2.2. Resultados obtenidos

En primer lugar, es crucial mencionar que en cada una de las 20 *capas* (cada una de las distintas operaciones GEMM que se implementan en ResNet50-v1.5, lo cual no se corresponde con el número de capas reales de esta red, ya que hay capas distintas de la red que utilizan una misma operación GEMM, así como capas donde no se realiza ninguna), se ha utilizado la combinación de los parámetros *elements_per_thread*, *num_threads* y *step* que ha permitido obtener un mayor rendimiento.

En la Tabla 4.3 se muestran dichas combinaciones, siendo éstas en muchas ocasiones muy similares a las combinaciones óptimas para las operaciones GEMM de matrices cuadradas con las que se fue probando la implementación.

Tabla 4.3: Combinaciones de parámetros óptimas para las operaciones GEMM correspondientes a las capas de la red ResNet50-v1.5.

Capa	PARÁMETROS ÓPTIMOS EN LA MÁQUINA QUADRO	PARÁMETROS ÓPTIMOS EN LA MÁQUINA VOLTA
1	(4, 16, 32)	(4, 16, 32)
2	(4, 16, 64)	(4, 16, 64)
3	(4, 16, 32)	(4, 16, 32)
4	(8, 16, 16)	(4, 16, 64)
5	(4, 16, 32)	(4, 16, 32)
6	(8, 16, 16)	(8, 16, 32)
7	(8, 16, 32)	(8, 16, 32)
8	(8, 16, 32)	(8, 16, 32)
9	(8, 16, 32)	(8, 16, 32)
10	(8, 16, 16)	(8, 16, 32)
11	(8, 16, 32)	(8, 16, 32)
12	(8, 16, 32)	(8, 16, 32)
13	(8, 16, 16)	(8, 16, 32)
14	(8, 16, 16)	(8, 16, 32)
15	(8, 16, 16)	(4, 16, 32)
16	(8, 16, 16)	(8, 16, 32)
17	(8, 16, 32)	(4, 16, 32)
18	(8, 16, 16)	(8, 16, 32)
19	(8, 16, 16)	(8, 16, 32)
20	(8, 16, 32)	(4, 16, 32)

En las Figuras 4.9 y 4.10 se realiza una comparativa del rendimiento obtenido, medido en GFLOPS, para las 20 operaciones GEMM empleadas en las capas de la red neuronal ResNet50-v1.5, utilizando cuBLAS, la función AutoScheduler de TVM y la versión final de nuestro algoritmo.

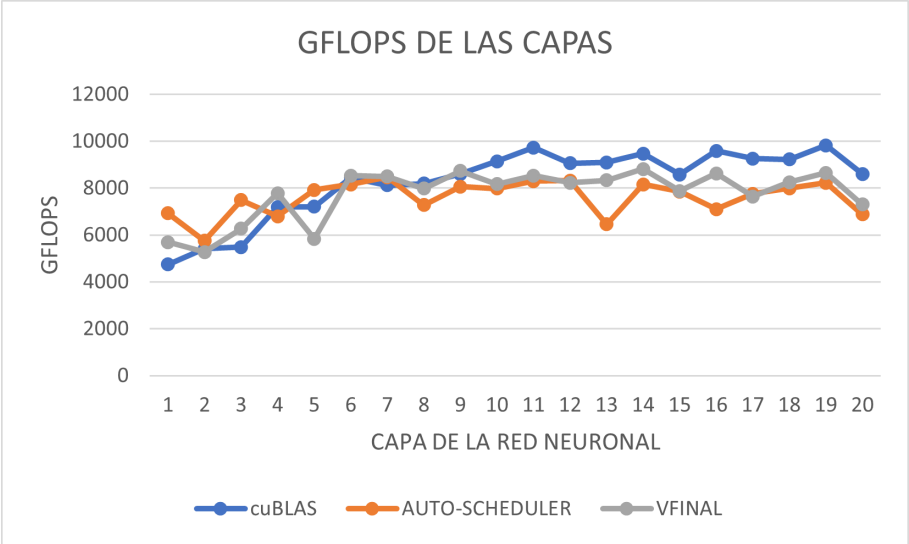

Figura 4.9: Gráfico comparativo de rendimiento para las capas de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro.

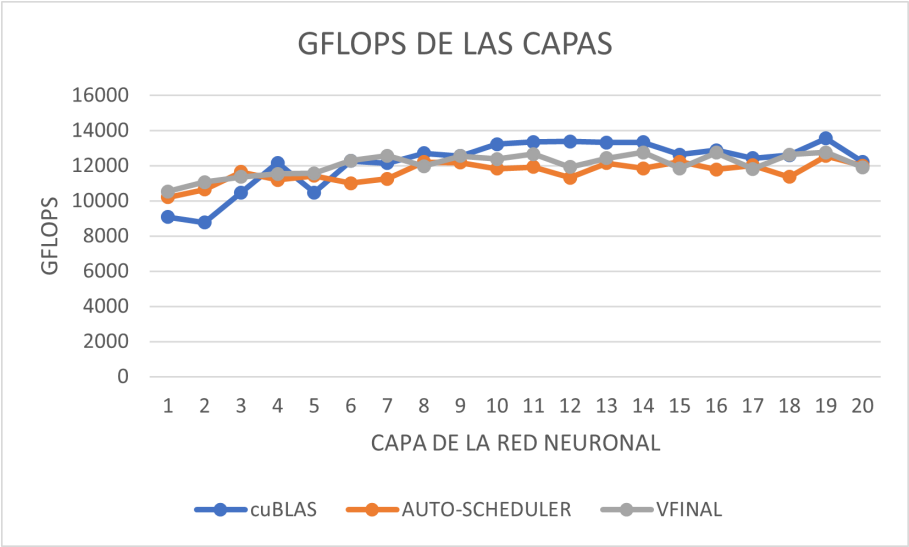

Figura 4.10: Gráfico comparativo de rendimiento para las capas de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta.

Claramente se pueden dividir los resultados en dos partes: las capas 1-9 (que se corresponden con las matrices más grandes y más rectangulares), donde tanto nuestra versión final como el AutoScheduler han sido capaces de superar a cuBLAS; y las capas 10-20, donde cuBLAS es el que ha obtenido un mayor rendimiento, seguido por nuestra implementación y en último lugar el AutoScheduler.

Esto corrobora la afirmación que hicimos anteriormente enunciando que nuestro código (y también el AutoScheduler) permitía obtener unas muy buenas prestaciones en las operaciones GEMM con matrices rectangulares de gran tamaño, habiendo sido posible incluso superar en ciertos casos a la implementación estándar de facto para GPU de NVIDIA, la cual es cuBLAS.

Finalmente, se ha calculado el tiempo agregado total de las operaciones a realizar en la red ResNet50-v1.5 (Tabla 4.4 y Figuras 4.11, 4.12), teniendo en cuenta el número de ocasiones en las que se utiliza cada una de las capas a lo largo de las capas con las que cuenta la red neuronal.

Tabla 4.4: Resultados de tiempo agregado total de la ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en ambas máquinas.

	cuBLAS	AUTO-SCHEDULER	VFINAL
QUADRO	0.129375	0.135905	0.134348
VOLTA	0.084895	0.089987	0.086966

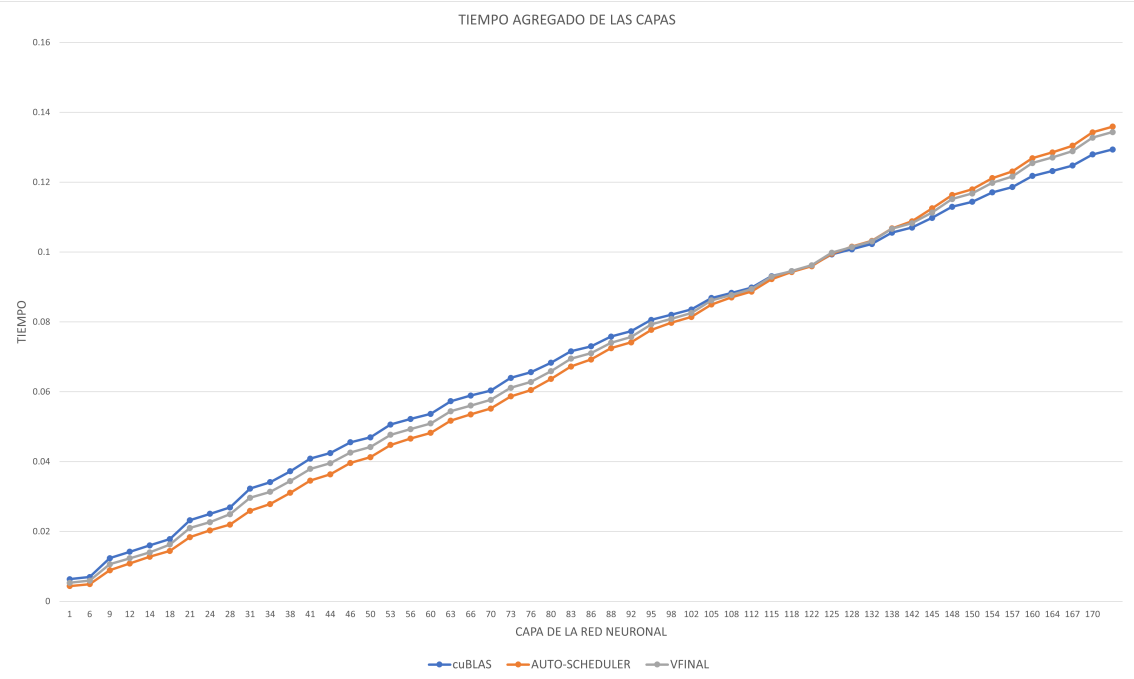

Figura 4.11: Gráfico del tiempo agregado total de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Quadro.

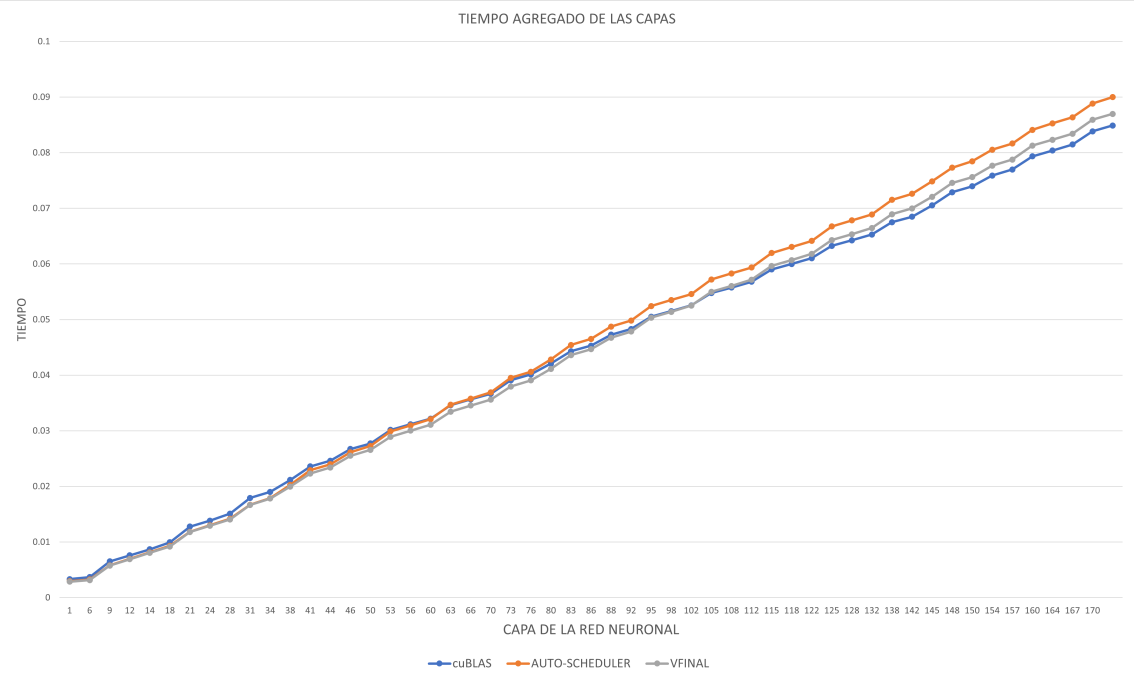

Figura 4.12: Gráfico del tiempo agregado total de la red ResNet50-v1.5 usando cuBLAS, AutoScheduler de TVM y la versión final del código desarrollado en la máquina Volta.

De nuevo, es cuBLAS quien gana la batalla, aunque únicamente comienza a tomar ventaja a partir de la capa 120 en la máquina Quadro y la capa 100 en la máquina Volta, siendo en la primera máquina el AutoScheduler quien llevaba la delantera hasta entonces, seguido de nuestra implementación, invirtiéndose posteriormente las tornas para quedar en última posición, aunque muy seguido de cerca por nuestro código. Por otra parte, en la máquina Volta es nuestra versión final la que presenta un menor tiempo agregado durante las primeras 100 capas, mientras que el AutoScheduler cada vez se va alejando más de los otros dos. En esta máquina la diferencia entre nuestro código y cuBLAS es notablemente menor que la distancia que presenta con el AutoScheduler.

Es importante destacar que las capas donde se consigue mejorar los resultados de cuBLAS se corresponden en la mayoría de los casos a operaciones realizadas con matrices muy rectangulares. Esto es debido a que cuBLAS está especialmente optimizado para matrices cuadradas o no excesivamente rectangulares, siendo imposible alcanzarlo para estos casos debido a que ha sido implementado para sacar el máximo partido posible a la arquitectura en la que se ejecute. Debido a ello, el tiempo agregado presenta este comportamiento, pues es en las primeras capas donde se realizan las operaciones más rectangulares.

Estos resultados muestran que, pese a que en cuanto al tiempo total de cómputo de las operaciones empleadas en la red ResNet50-v1.5 no ha sido posible superar los resultados de cuBLAS, sí que es posible mejorar su rendimiento, tanto empleando nuestro algoritmo como empleando la función AutoScheduler de TVM, en ciertas operaciones con matrices muy rectangulares.

CAPÍTULO 5
Conclusiones

En este Trabajo Fin de Máster se ha implementado un código que permite aprovechar las prestaciones que ofrece una GPU para ejecutar de forma eficiente la operación GEMM utilizando el framework de generación de código Apache TVM.

Se ha definido el cómputo a realizar empleando las Tensor Expressions de TVM y mediante las operaciones de scheduling se ha llevado a cabo una serie de optimizaciones que han permitido incrementar el rendimiento de manera notable, alcanzando, para un tamaño de problema de 2048, en las máquinas Quadro y Volta unos valores de GFLOPS de 8000 y 11000, respectivamente.

A continuación se ha realizado una serie de pruebas incrementando el tamaño de problema, consiguiéndose un rendimiento máximo de 9300 (Quadro) y 13000 (Volta) GFLOPS, no viéndose mermado apenas el rendimiento al trabajar con matrices rectangulares.

Seguidamente se ha efectuado una comparativa entre el código desarrollado, la librería de álgebra lineal de altas prestaciones cuBLAS y la función de tuning AutoScheduler que ofrece Apache TVM. Los resultados obtenidos prueban que nuestro código es capaz de mejorar el rendimiento obtenido por la función AutoScheduler, pero no el de cuBLAS. La ventaja de cuBLAS se reduce considerablemente al incrementar el tamaño de las matrices.

Finalmente, se ha medido el rendimiento correspondiente a las operaciones GEMM empleadas en un caso de estudio real, la red neuronal ResNet50-v1.5. Estudiando el rendimiento capa a capa se puede ver que en algunas de las capas de la red, concretamente en las que se correspondían a los productos de matrices más rectangulares y de mayor tamaño, sí que ha sido posible superar a cuBLAS. Cabe destacar que en la totalidad de las capas en la máquina Volta y en la gran mayoría en la máquina Quadro se han obtenido mejores resultados con nuestro código que los que permite el AutoScheduler de TVM, habiendo, además, en la máquina Volta una diferencia de rendimiento mucho menor entre cuBLAS y nuestro código para las matrices de menor tamaño donde no es posible aventajar a cuBLAS.

Por otra parte, analizando el tiempo agregado total se puede ver que en ambas máquinas es cuBLAS quien gana, siendo mayor la ventaja de éste en la máquina Quadro que en la máquina Volta. Nuestro código tiene en la primera máquina unos resultados de tiempo total muy similares (aunque mejores) a los del AutoScheduler mientras que existe una mayor diferencia entre éstos y una mayor proximidad al rendimiento de cuBLAS en la máquina Volta.

Estos resultados muestran que el código desarrollado en TVM es capaz de competir con implementaciones de altas prestaciones de vanguardia como puede ser cuBLAS, e

incluso mejorar su rendimiento en matrices grandes y muy rectangulares, debido a que muchas librerías de álgebra lineal están especialmente enfocadas a productos de matrices cuadradas. Además, ha sido posible obtener un rendimiento mejor que el que se puede conseguir usando la funcionalidad de AutoScheduler que ofrece el propio TVM.

Finalmente, cabe mencionar que nuestro código tiene un mucho mejor resultado de GFLOPS tanto absoluto como relativo en comparación con cuBLAS y AutoScheduler en la máquina Volta. Esto es debido a que las GPU de la familia Tesla (a la cual pertenece) están enfocadas a la Computación de Altas Prestaciones y el GPGPU, mientras que la familia Quadro tiene como principal destinatario su uso en estaciones de trabajo de diseño profesional. Debido a esto, la GPU presente en la máquina Volta, pese a ser más antigua, está más especializada en realizar cómputos como los que se emplean en la operación de producto matricial.

Durante el desarrollo del Trabajo se han puesto en práctica diversas materias estudiadas en el Máster, especialmente las referidas a Computación de Altas Prestaciones y programación de GPU. Ha resultado interesante y motivador poder poner en práctica todos esos conocimientos para un caso que se emplea en el mundo real, pues es de vital importancia contar con implementaciones de alto rendimiento de los algoritmos de Machine Learning.

En cuanto a posibles futuras investigaciones podría ser interesante realizar este estudio de rendimiento en una GPU NVIDIA más moderna que cuente con la arquitectura Ampere, sucesora de Volta y Turing, así como probar con otras GPU de otros fabricantes (puesto que Apache TVM también soporta compiladores de OpenCL, lenguaje abierto e independiente de la plataforma). Otro interesante estudio futuro sería la implementación de otros núcleos computaciones (como por ejemplo una convolución) o de un modelo completo de red neuronal optimizado.

Bibliografía

[1] DataScientest - Inteligencia artificial : definición, historia, usos, peligros Consultado en febrero de 2023 https://datascientest.com/es/inteligencia-artificial-definicion.

[2] Turing A. M. Computing Machinery and Intelligence. *Technology and Culture*, 59, 433–460, octubre, 1950.

[3] ComputerWeekly - Inteligencia Artificial o IA Consultado en febrero de 2023 https://www.computerweekly.com/es/definicion/Inteligencia-artificial-o-IA.

[4] KeepCoding - ¿Cuáles son los tipos de Inteligencia Artificial que existen? Consultado en febrero de 2023 https://keepcoding.io/inteligencia-artificial/tipos-de-inteligencia-artificial/.

[5] Grupo Atico34 - Inteligencia artificial: Definición, tipos y aplicaciones Consultado en febrero de 2023 https://protecciondatos-lopd.com/empresas/inteligencia-artificial/.

[6] IBM - Inteligencia artificial (IA) Consultado en febrero de 2023 https://www.ibm.com/es-es/cloud/learn/what-is-artificial-intelligence/.

[7] Hewlett Packard Enterprise - ¿Qué es el aprendizaje automático? Consultado en febrero de 2023 https://www.hpe.com/lamerica/es/what-is/machine-learning.html/.

[8] Google Cloud - Inteligencia artificial (IA) vs. aprendizaje automático (AA) Consultado en febrero de 2023 https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning?hl=es-419/.

[9] Microsoft Azure - ¿Qué es el aprendizaje automático? Consultado en febrero de 2023 https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-machine-learning-platform/.

[10] Iberdrola - QUÉ ES EL 'MACHINE LEARNING' Consultado en febrero de 2023 https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico/.

[11] Revista de Robots - Qué es el Machine Learning y el Aprendizaje Automático Consultado en febrero de 2023 https://revistaderobots.com/inteligencia-artificial/que-es-el-machine-learning-y-el-aprendizaje-automatico/.

[12] TensorFlow - Crea modelos de aprendizaje automático de nivel de producción con TensorFlow Consultado en febrero de 2023 https://www.tensorflow.org/?hl=es-419/.

[13] dlib C++ Library Consultado en febrero de 2023 http://dlib.net/.

[14] mlpack - fast, header-only C++ machine learning library Consultado en febrero de 2023 https://www.mlpack.org/.

[15] Apache Spark - MLlib Consultado en febrero de 2023 https://spark.apache.org/mllib/.

[16] Weka 3: Machine Learning Software in Java Consultado en febrero de 2023 https://www.cs.waikato.ac.nz/ml/weka/.

[17] AWS - ¿Qué es una red neuronal? Consultado en febrero de 2023 https://aws.amazon.com/es/what-is/neural-network/.

[18] ATRIA Innovation - Qué son las redes neuronales y sus funciones Consultado en febrero de 2023 https://www.atriainnovation.com/que-son-las-redes-neuronales-y-sus-funciones/.

[19] UNIR - ¿Qué son las redes neuronales? Concepto y usos principales Consultado en febrero de 2023 https://www.unir.net/ingenieria/revista/redes-neuronales-artificiales/.

[20] IBM - El modelo de redes neuronales Consultado en febrero de 2023 https://www.ibm.com/docs/es/spss-modeler/saas?topic=networks-neural-model/.

[21] ThinkBig - ¿Sabes en qué se diferencian las redes neuronales del Deep Learning? Consultado en febrero de 2023 https://blogthinkbig.com/redes-neuronales-deep-learning/.

[22] Crehana - ¿Qué es un perceptrón? La red neuronal artificial más antigua Consultado en febrero de 2023 https://www.crehana.com/blog/transformacion-digital/que-es-perceptron-algoritmo/.

[23] Xataka - Las redes neuronales: qué son y por qué están volviendo Consultado en febrero de 2023 https://www.xataka.com/robotica-e-ia/las-redes-neuronales-que-son-y-por-que-estan-volviendo/.

[24] AprendeIA - ¿Qué es el Perceptrón? Perceptrón Simple y Multicapa Consultado en febrero de 2023 https://aprendeia.com/que-es-el-perceptron-simple-y-multicapa/.

[25] Hewlett Packard Enterprise - Inteligencia artificial Consultado en febrero de 2023 https://www.hpe.com/lamerica/es/what-is/artificial-intelligence.html.

[26] Hewlett Packard Enterprise - Aprendizaje Profundo Consultado en febrero de 2023 https://www.hpe.com/es/es/what-is/deep-learning.html/.

[27] Google Cloud - ¿Qué es la inteligencia artificial o IA? Consultado en febrero de 2023 https://cloud.google.com/learn/what-is-artificial-intelligence?hl=es-419.

[28] NetApp - ¿Qué es la computación de alto rendimiento? Consultado en febrero de 2023 https://www.netapp.com/es/data-storage/high-performance-computing/what-is-hpc/.

[29] IBM - ¿Qué es la computación de alto rendimiento (HPC)? Consultado en febrero de 2023 https://www.ibm.com/es-es/topics/hpc/.

[30] Jose Aguilar, Ernst Leiss *Introducción a la Computación Paralela*. Mérida, Venezuela, primera edición, 2004.

[31] Amdahl G. M. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings, 30, 483–485, abril, 1967.

[32] Javier Del Campo *Análisis y aceleración de algoritmos metaheurísticos de optimización discreta*. Universidad Miguel Hernández de Elche, 2021.

[33] AMD - Computación de Alto Rendimiento (HPC) Consultado en febrero de 2023 https://www.amd.com/es/technologies/hpc-explained/.

[34] Hewlett Packard Enterprise - Computación de alto rendimiento Consultado en febrero de 2023 https://www.hpe.com/lamerica/es/what-is/high-performance-computing.html/.

[35] Heavy.ai - Hardware Acceleration Consultado en febrero de 2023 https://www.heavy.ai/technical-glossary/hardware-acceleration/.

[36] ADSLZone - Qué es la GPU en un ordenador y por qué es importante Consultado en febrero de 2023 https://www.adslzone.net/esenciales/preguntas/que-es-gpu/.

[37] SoftwareLab - ¿Qué es la GPU o tarjeta gráfica de un ordenador? Consultado en febrero de 2023 https://softwarelab.org/es/que-es-la-gpu-o-tarjeta-grafica-de-un-ordenador/.

[38] MuyComputer - Qué es una GPU, qué elementos la forman y por qué es tan importante Consultado en febrero de 2023 https://www.muycomputer.com/2021/01/19/que-es-una-gpu-importante/.

[39] HardZone - Seguro que sabes lo que es una tarjeta gráfica, pero ¿qué es una GPU? Consultado en febrero de 2023 https://hardzone.es/reportajes/que-es/gpu-caracteristicas-especificaciones/.

[40] CUDA C++ Programming Guide Consultado en enero de 2023 https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[41] Universidad de Burgos - Introducción a la programación en CUDA Consultado en febrero de 2023 https://riubu.ubu.es/bitstream/handle/10259/3933/Programacion_en_CUDA.pdf/.

[42] THE WORLD'S FIRST RAY TRACING GPU NVIDIA QUADRO RTX 5000 Consultado en marzo de 2023 https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-5000-data-sheet-us-nvidia-704120-r4-web.pdf/.

[43] NVIDIA TESLA V100 GPU ACCELERATOR Consultado en marzo de 2023 https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf/.

[44] NVIDIA TESLA V100 GPU ARCHITECTURE - THE WORLD'S MOST ADVANCED DATA CENTER GPU Consultado en marzo de 2023 https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf/.

[45] NVIDIA TURING GPU ARCHITECTURE - Graphics Reinvented Consultado en marzo de 2023 https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf/.

[46] The Apache Software Foundation Blog - THE APACHE SOFTWARE FOUNDATION ANNOUNCES APACHE® TVM™ AS A TOP-LEVEL PROJECT Consultado en febrero de 2023 https://news.apache.org/foundation/entry/the-apache-software-foundation-announces69/.

[47] Apache TVM Consultado en febrero de 2023 https://tvm.apache.org/.

[48] Apache TVM Docs - Introduction Consultado en febrero de 2023 https://tvm.apache.org/docs/tutorial/introduction.html#sphx-glr-tutorial-introduction-py/.

[49] Apache TVM Docs - Compiling and Optimizing a Model with TVMC Consultado en febrero de 2023 https://tvm.apache.org/docs/tutorial/tvmc_command_line_driver.html/.

[50] Apache TVM Docs - Working with Operators Using Tensor Expression Consultado en febrero de 2023 https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html/.

[51] Apache TVM Docs - Schedule Primitives in TVM Consultado en febrero de 2023 https://tvm.apache.org/docs/how_to/work_with_schedules/schedule_primitives.html/.

[52] Apache TVM Docs - How to optimize GEMM on CPU Consultado en marzo de 2023 https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html/.

[53] Apache TVM Docs - How to optimize convolution on GPU Consultado en marzo de 2023 https://tvm.apache.org/docs/how_to/optimize_operators/opt_conv_cuda.html/.

[54] Daniel Lemire's blog - Why are unrolled loops faster? Consultado en marzo de 2023 https://lemire.me/blog/2019/04/12/why-are-unrolled-loops-faster/.

[55] Dive into Deep Learning Compiler - 5. Convolution Consultado en marzo de 2023 https://tvm.d2l.ai/chapter_gpu_schedules/conv.html/.

[56] Pete Warden's blog - Why GEMM is at the heart of deep learning Consultado en febrero de 2023 https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[57] Kaiming H. *Deep Residual Learning for Image Recognition*. Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition, 770-778, junio, 2016.

[58] TowardsDataScience - The Vanishing/Exploding Gradient Problem in Deep Neural Networks Consultado en febrero de 2023 https://towardsdatascience.com/the-vanishing-exploding-gradient-problem-in-deep-neural-networks-191358470c11/.

[59] Analytics Vidhya - Deep Residual Learning for Image Recognition (ResNet Explained) Consultado en febrero de 2023 https://www.analyticsvidhya.com/blog/2023/02/deep-residual-learning-for-image-recognition-resnet-explained/?utm_source=related_WP&utm_medium=https://www.analyticsvidhya.com/blog/2021/06/understanding-resnet-and-analyzing-various-models-on-the-cifar-10-dataset/.

[60] Opengenus - ResNet50 v1.5 architecture Consultado en febrero de 2023 https://iq.opengenus.org/resnet50-v1-5/.

[61] Stuart Russell, Peter Norvig *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, tercera edición, 2010.

[62] MLCommons - v2.1 Results Consultado en febrero de 2023 https://mlcommons.org/en/inference-datacenter-21/.

[63] NVIDIA - MLPerf benchmarks Consultado en febrero de 2023 https://www.nvidia.com/en-us/data-center/resources/mlperf-benchmarks/.

APÉNDICE A
Comparativa de GPU

En la figura A.1 se realiza un cuadro comparativo de las dos GPU utilizadas en el desarrollo del Trabajo: Quadro RTX 5000 y Tesla V100.

Características	Quadro RTX 5000	Tesla V100
Arquitectura	Turing	Volta
GPCs	6	NA
SMs	48	80
TPCs	24	40
FP32 Cores / SM	64	64
FP32 Cores / GPU	3072	5120
INT32 Cores / SM	64	64
INT32 Cores / GPU	3072	5120
FP64 Cores / SM	NA	32
FP64 Cores / GPU	NA	2560
Tensor Cores / SM	8	8
Tensor Cores / GPU	384	640
RT Cores	48	NA
Frecuencia de reloj máxima	1815 MHz	1530 MHz
RTX-OPS (Tera-OPS)	62	NA
Rays Cast (Giga Rays/sec)	1815	NA
Pico de TFLOPS en FP32	11.2	14
Pico de TIPS en INT32	11.2	NA
Pico de TFLOPS en FP64	NA	7
Pico de TFLOPS en FP16	22.3	NA
Pico de TFLOPS en Tensores	89.2	112
Unidades de Textura	192	320
Interfaz de Memoria	256 bits	4096 bits
Tamaño de Memoria y Tipo	16 GB GDDR6	32 GB HBM2
Reloj de la Memoria (Transferencia de Datos)	14 Gbps	NA
Ancho de Banda de la Memoria	448 GB/s	Hasta 900 GB/s
Tamaño de Cache L2	4096 KB	6144 KB
Tamaño de Memoria Compartida / SM	96 KB	Configurable hasta 96 KB
Tamaño de Banco de Registros / SM	256 KB	256 KB
Tamaño de Banco de Registros / GPU	12288 KB	20480 KB
TDP	230 W	300 W
Transistores	13.6 mil millones	21.1 mil millones
Tamaño del Die de la GPU	545 mm^2	815 mm^2
Proceso de Manufacturación	12 nm FFN	12 nm FFN

Figura A.1: Comparativa de las dos GPU empleadas.

APÉNDICE B
Estudio sobre el rendimiento

Matrices cuadradas de tamaño 4096

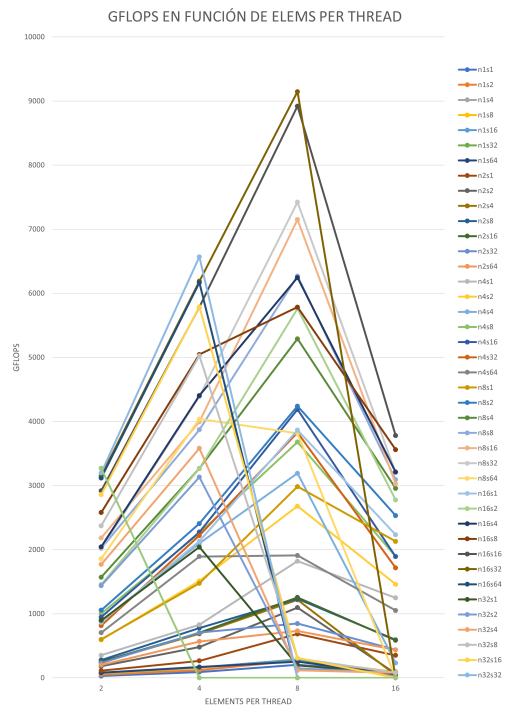

Figura B.1: Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 4096 en la máquina Quadro.

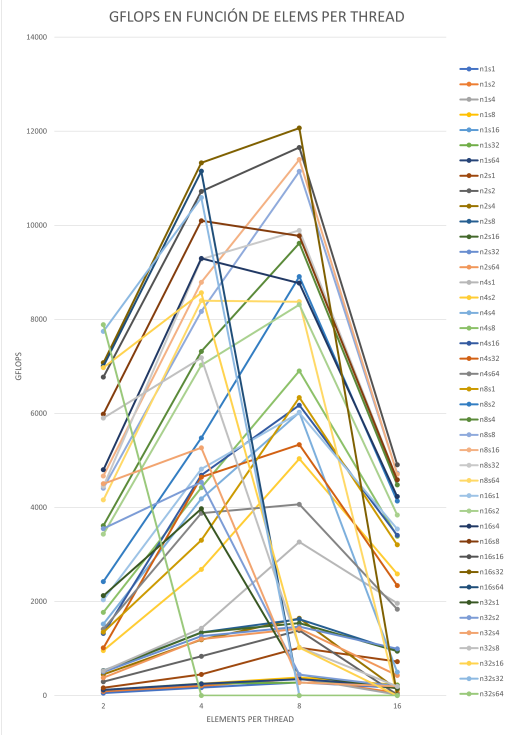

Figura B.2: Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 4096 en la máquina Volta.

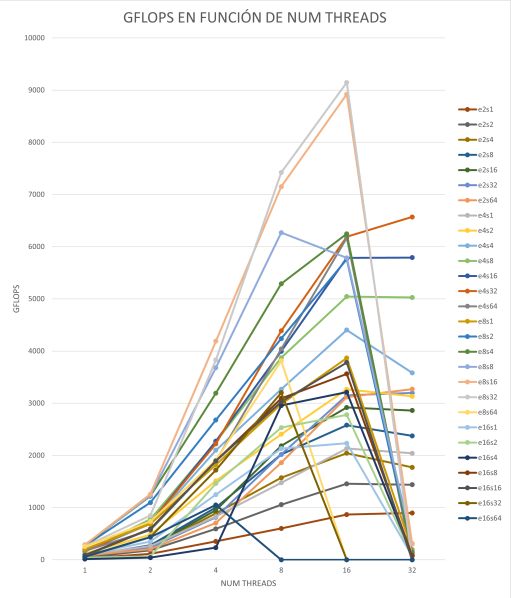

Figura B.3: Gráfico del rendimiento en función de num threads para un tamaño de matriz de 4096 en la máquina Quadro.

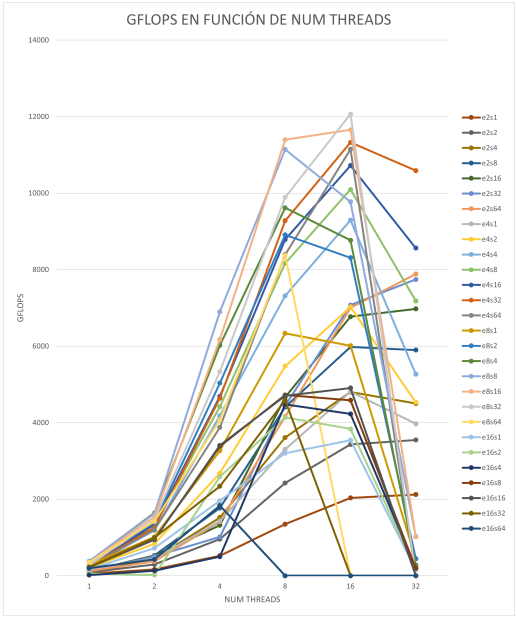

Figura B.4: Gráfico del rendimiento en función de num threads para un tamaño de matriz de 4096 en la máquina Volta.

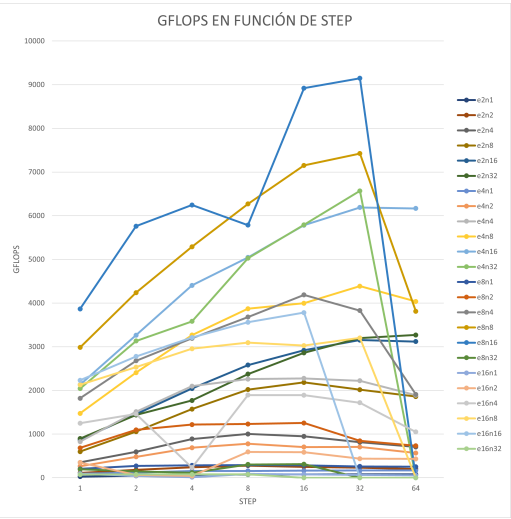

Figura B.5: Gráfico del rendimiento en función de step para un tamaño de matriz de 4096 en la máquina Quadro.

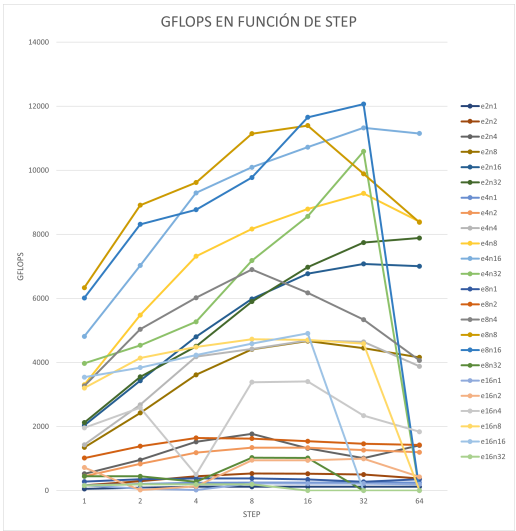

Figura B.6: Gráfico del rendimiento en función de step para un tamaño de matriz de 4096 en la máquina Volta.

Incrementando el tamaño del problema ha sido posible ver que nuestra implementación es realmente capaz de obtener mayores prestaciones que las mostradas anteriormente, siendo por tanto 2048 un tamaño demasiado reducido como para poder observar todo el potencial del código desarrollado. Con un tamaño de matriz de 4096, ha sido posible alcanzar los 9000 GFLOPS en la máquina Quadro (Figuras B.1, B.3 y B.5), y los 12000 en la máquina Volta (Figuras B.2, B.4 y B.6), valores claramente superiores a los de la ejecución con matrices de 2048.

En lo que respecta a los parámetros empleados, se puede apreciar que el comportamiento en la máquina Quadro es prácticamente idéntico a lo que sucedía con matrices más pequeñas, mientras que en la máquina Volta el valor óptimo de *elements_per_thread*

pasa a ser 8, obteniéndose por tanto la misma combinación óptima en ambas máquinas: (8, 16, 32).

Matrices cuadradas de tamaño 8192

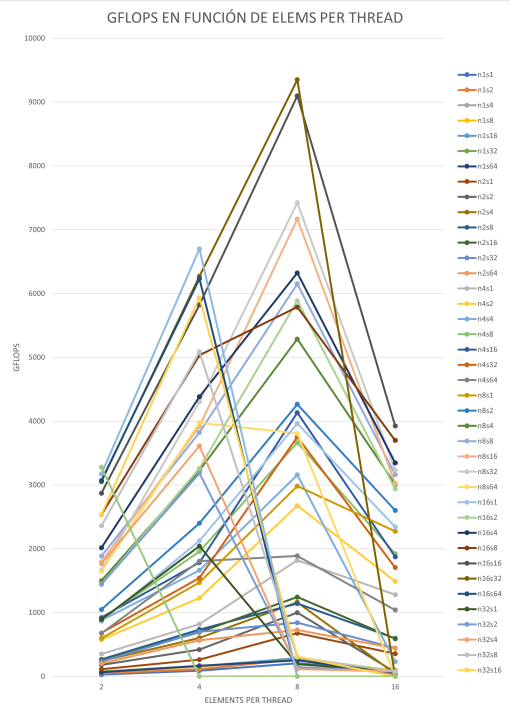

Figura B.7: Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 8192 en la máquina Quadro.

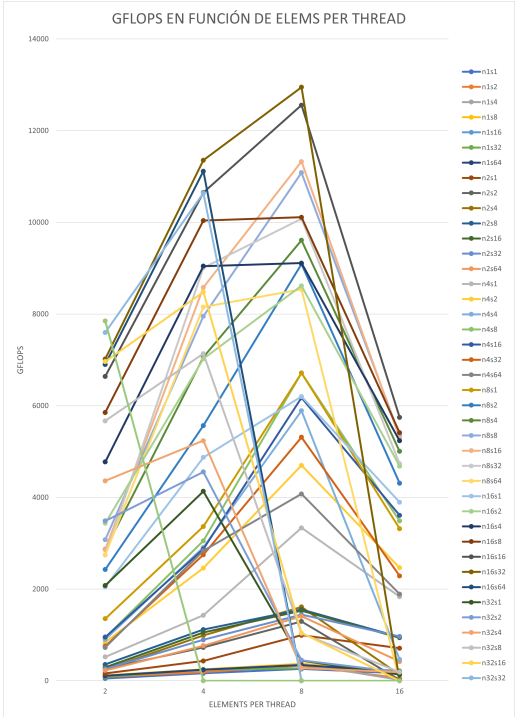

Figura B.8: Gráfico del rendimiento en función de elements per thread para un tamaño de matriz de 8192 en la máquina Volta.

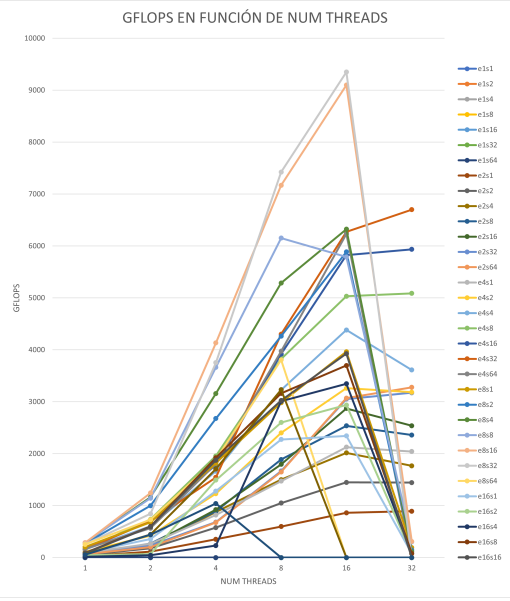

Figura B.9: Gráfico del rendimiento en función de num threads para un tamaño de matriz de 8192 en la máquina Quadro.

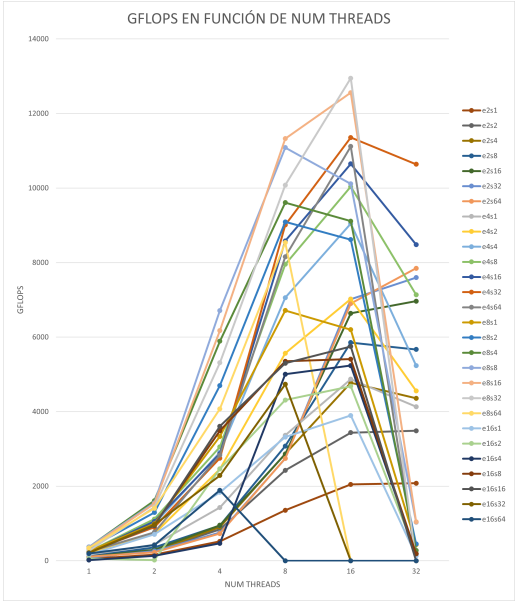

Figura B.10: Gráfico del rendimiento en función de num threads para un tamaño de matriz de 8192 en la máquina Volta.

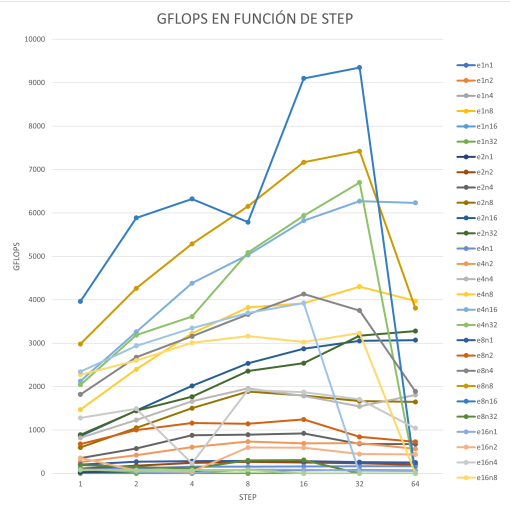

Figura B.11: Gráfico del rendimiento en función de step para un tamaño de matriz de 8192 en la máquina Quadro.

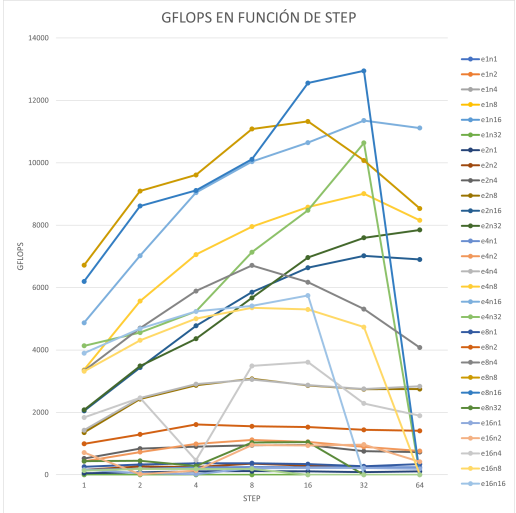

Figura B.12: Gráfico del rendimiento en función de step para un tamaño de matriz de 8192 en la máquina Volta.

En lo que respecta a la máquina Quadro (Figuras B.7, B.9 y B.11), el incremento en prestaciones obtenido al aumentar el tamaño de matriz a 8192 es mucho menor, y el comportamiento de los distintos parámetros no ha variado, por lo que ya se puede hablar de que el rendimiento obtenido es el máximo real, no estando condicionado por un tamaño de problema demasiado pequeño y que no permita aprovechar el potencial real de cómputo. Por tanto, se puede afirmar que el rendimiento límite roza los 9300 GFLOPS y los valores óptimos de los parámetros son (8, 16, 32), no siendo necesaria ninguna prueba con matrices aún más grandes.

Por otra parte, el rendimiento en la máquina Volta (Figuras B.8, B.10 y B.12) crece unos 1000 GFLOPS más, llegando casi a los 13000 GFLOPS. Las combinaciones de parámetros con mayor rendimiento se mantienen, siendo la óptima (8, 16, 32), pero las distancias entre estas mejores combinaciones (8, 8-16, 16-32) y el resto se incrementan, tomando un comportamiento ligeramente más parecido al de la otra máquina.

Matrices rectangulares: A[8192 × 2048], B[2048 × 8192]

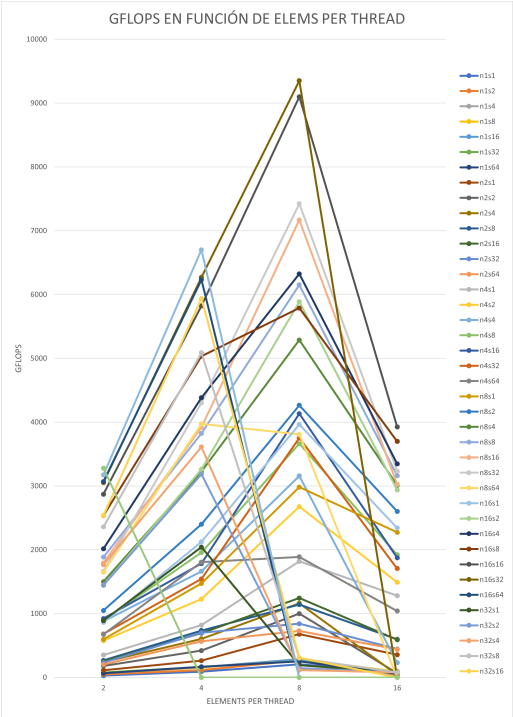

Figura B.13: Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro.

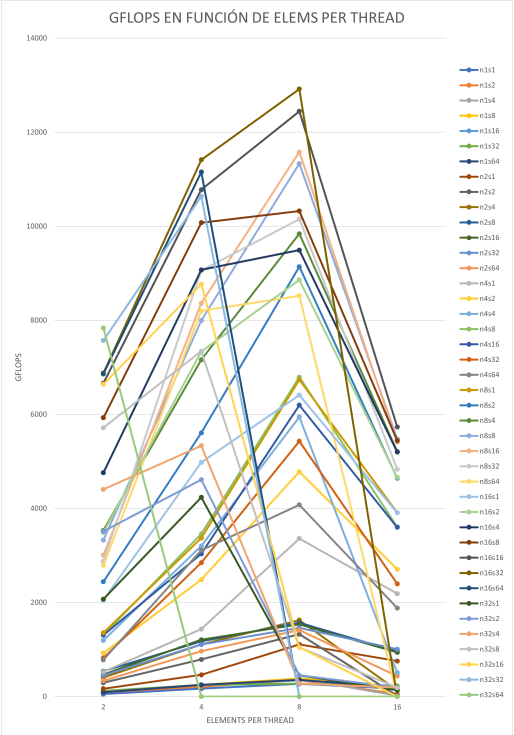

Figura B.14: Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta.

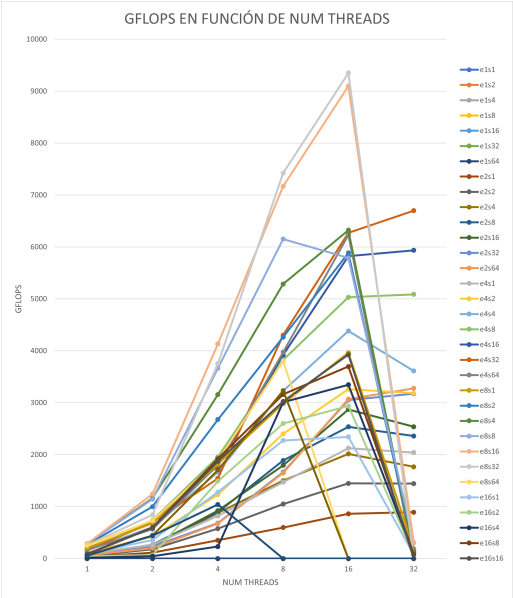

Figura B.15: Gráfico del rendimiento en función de num threads para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro.

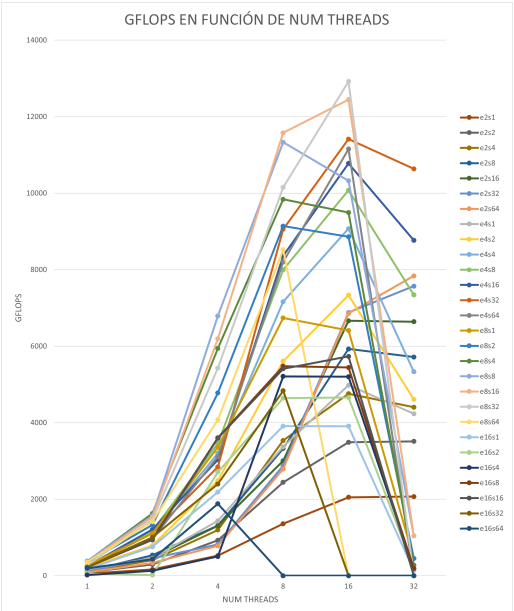

Figura B.16: Gráfico del rendimiento en función de num threads para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta.

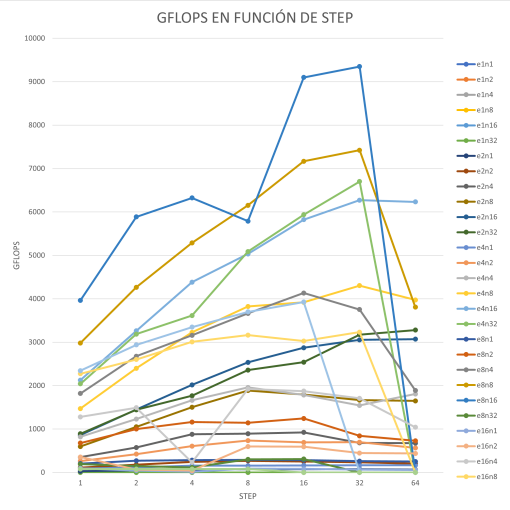

Figura B.17: Gráfico del rendimiento en función de step para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Quadro.

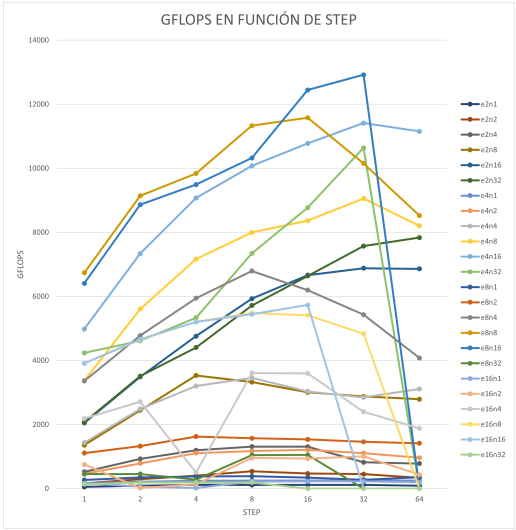

Figura B.18: Gráfico del rendimiento en función de step para matrices rectangulares con M=8192, N=8192, K=2048 en la máquina Volta.

Estos gráficos (Figuras B.13, B.14, B.15, B.16, B.17 y B.18) muestran que en ambas máquinas el rendimiento es igual al obtenido con matrices cuadradas de 8192, por lo que se puede afirmar que el código no presenta un decremento en las prestaciones para este tamaño de matrices rectangulares.

Matrices rectangulares: A[2048 × 8192], B[8192 × 2048]

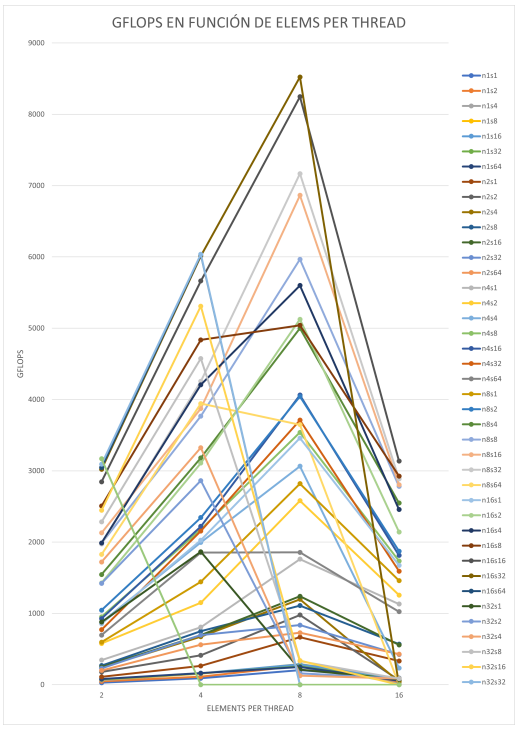

Figura B.19: Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro.

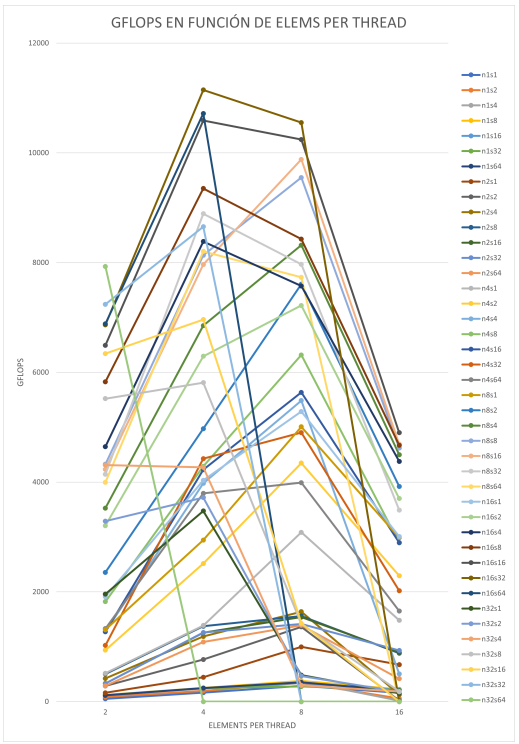

Figura B.20: Gráfico del rendimiento en función de elements per thread para matrices rectangulares con M=2048, N=2048, K=8192, K=2048 en la máquina Volta.

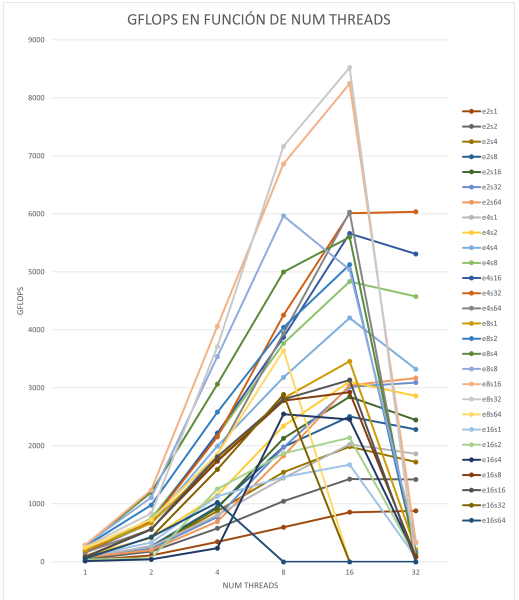

Figura B.21: Gráfico del rendimiento en función de num threads para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro.

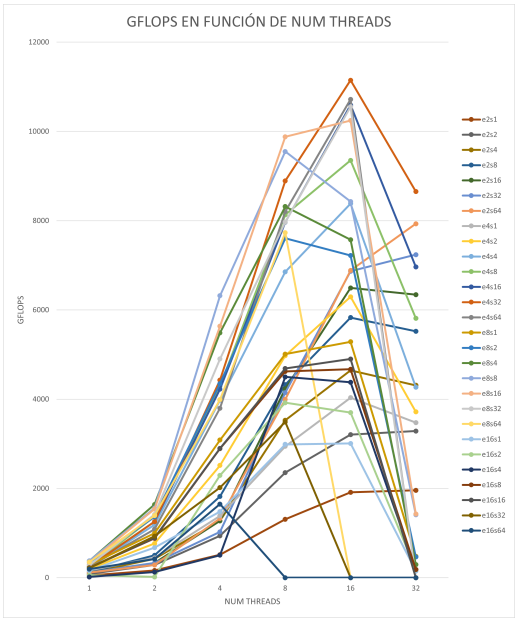

Figura B.22: Gráfico del rendimiento en función de num threads para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Volta.

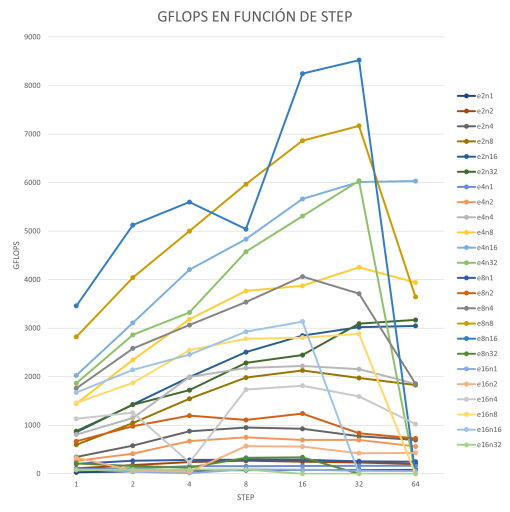

Figura B.23: Gráfico del rendimiento en función de step para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Quadro.

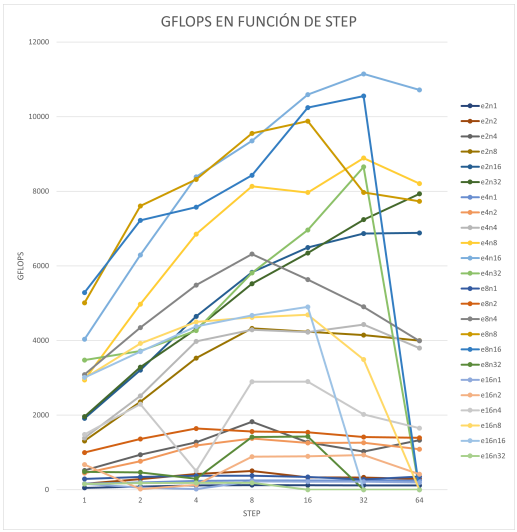

Figura B.24: Gráfico del rendimiento en función de step para matrices rectangulares con M=2048, N=2048, K=8192 en la máquina Volta.

Empleando estas matrices rectangulares, el rendimiento obtenido en la máquina Quadro (Figuras B.19, B.21 y B.23) ha sido de unos 8500 GFLOPS, un valor intermedio entre los correspondientes a los logrados con matrices cuadradas 2048 y 4096 (algo lógico teniendo en cuenta que el número de operaciones a realizar es $M \times N \times K$).

En cambio, en la máquina Volta (Figuras B.20, B.22 y B.24) los resultados son muy similares a los del caso de las matrices cuadradas de 2048 (con 11000 GFLOPS y un *elements_per_thread* óptimo de 4), por lo que la caída de rendimiento en este caso es mayor, posiblemente debido a que en esta máquina, más potente, el tamaño de problema es demasiado reducido como para poder aprovechar toda su potencia de cómputo.