Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Optimización del Cálculo en el Modelado de Estructuras en Puentes Ferroviarios y Entrenamiento de Redes Neuronales con GPUs y CUDA

Trabajo fin de grado

Grado en Ingeniería Informática

Autor: Medina Chaveli, Laura

Tutor: Flich Cardo, José
Museros Romero, Pedro

Curso 2019-2020

Resum

L'objectiu d'aquest treball és l'optimització de dues aplicacions que contenen un elevat cost computacional. La primera d'elles tracta d'un sistema enfocat a l'estudi de les estructures dels ponts ferroviaris on hi ha dos nivells d'optimització. La primera optimització se centra en la implementació d'un sistema distribuït mitjançant MPI i segona de la utilització de les GPUs de NVIDIA mitjançant CUDA per als càlculs. La segona es tracta d'HELENNA, una aplicació per a l'entrenament i inferència de xarxes neuronals. En aquest treball es desenvolupa el suport per GPUs de NVIDIA a través de CUDA i la seua combinació amb cuBLAS.

Paraules clau: GPU, CUDA, cuBLAS, MPI, Xarxes Neuronals, NVIDIA

Resumen

El objetivo de este trabajo es la optimización de dos aplicaciones que tienen un elevado coste computacional. La primera de ellas se trata de un sistema enfocado al estudio de las estructuras de los puentes ferroviarios donde existen dos niveles de optimización. La primera optimización se centra en la implementación de un sistema distribuido mediante MPI y segunda de la utilización de las GPUs de NVIDIA mediante CUDA para los cálculos. La segunda se trata de HELENNA, una aplicación para el entrenamiento e inferencia de redes neuronales. En este trabajo se desarrolla el soporte para GPUs de NVIDIA a través de CUDA y su combinación con cuBLAS.

Palabras clave: GPU, CUDA, cuBLAS, MPI, Redes Neuronales, NVIDIA

Abstract

The goal of this project is the optimization of two applications with a high computational cost. The first application is focused on the analysis of the structure of train bridges and its maintenance. The first optimization is focused on the implementation of a distributed system with MPI communication whereas the second optimization is focused on the use of GPUs from NVIDIA progframmed with CUDA. The second application is HELENNA, a neural network application used for training and inferencing. In this project we have developed the whole support for NVIDIA GPUs using CUDA and combined with cuBLAS.

Key words: GPU, CUDA, cuBLAS, MPI, Neural Networks, NVIDIA

Índice general

Índice general v
Índice de figuras vii
Índice de tablas viii

1 Introducción 1
1.1 Sistemas de computación heterogéneos . . . 1
1.2 Motivación . . . 3
1.3 Objetivos . . . 5
1.4 Estructura de la memoria . . . 5
1.5 Colaboraciones . . . 6
2 CUDA, cuBLAS, MKL y MPI 7
2.1 CUDA . . . 7
2.1.1 Arquitectura CUDA . . . 8
2.1.2 Modelo de Programación . . . 9
2.1.3 Gestión de Memoria . . . 11
2.1.4 Sincronización . . . 11
2.1.5 *Streams* . . . 12
2.2 cuBLAS . . . 13
2.3 MPI . . . 15
2.3.1 Procesos . . . 15
2.3.2 Comunicadores . . . 16
2.3.3 Creación de Procesos . . . 16
2.3.4 Tipos de Mensajes . . . 17
2.3.5 Transferencia de Mensajes . . . 17
2.3.6 Tipos de comunicaciones . . . 18
2.4 MKL . . . 19
3 Aplicación para Cálculo Dinámico por Elementos Finitos de puentes ferroviarios 21
3.1 Contexto de la Aplicación . . . 21
3.2 Descripción de la Aplicación . . . 22
3.3 Soporte de comunicación remota con MPI . . . 23
3.3.1 *Master* . . . 24
3.3.2 *Kernel* . . . 24
3.3.3 *Server* . . . 26
3.4 Soporte de GPUs con cuBLAS . . . 27
3.5 Evaluación . . . 29
4 Aplicación de entrenamiento de redes neuronales 31
4.1 Descripción de la Aplicación . . . 31
4.2 Soporte de GPU con cuBLAS . . . 33
4.2.1 Capa convolucional . . . 36
4.2.2 Capa *dropout* . . . 42
4.2.3 Soporte de creación, lectura y escritura de *buffers* . . . 44

4.2.4 Sincronización 45
4.3 Evaluación 46

5 Conclusiones **51**
5.1 Relación del trabajo desarrollado con los estudios cursados 52

6 Trabajo futuro **53**
6.1 Aplicación para Cálculo Dinámico por Elementos Finitos de puentes ferroviarios 53
6.1.1 Envío de datos circular 53
6.1.2 Almacenamiento persistente de matrices en la memoria de la GPU 53
6.1.3 Implementación de un sistema de colas en el *server* 54
6.1.4 Soporte para otras rutinas 54
6.2 Aplicación HELENNA 54
6.2.1 Implementación de *streams* de CUDA 54
6.2.2 Mejora de la función implementada col2im de la capa convolucional 55

Apéndices

A Código de la implementación de la aplicación para el Cálculo Dinámico por Elementos Finitos de puentes ferroviarios **61**
A.1 Creación de procesos 61
A.2 Identificación de GPU 62

B Configuración de las pruebas de evaluación **63**

Índice de figuras

1.1 Organización típica de la CPU y la GPU [1]. 1
1.2 Estructura de las redes neuronales profundas [2]. 2
1.3 Objetos detectados por YOLO [4] 3
1.4 Historia del rendimiento de la CPU y la GPU [6]. 4

2.1 GFLOPS de diferentes implementaciones [12]. 8
2.2 GPU Pascal GP100 [15] 9
2.3 SM de la GPU Pascal GP100 [15] 9
2.4 Tensor Cores. [17] 10
2.5 Modelo de programación [1]. 10
2.6 Jerarquía de memoria [19]. 12
2.7 Mejoras con el uso de *streams* [20]. 13
2.8 Comparación entre cuBLAS y MKL [24]. 13
2.9 Comparación entre cuBLAS y clBLAS [25]. 14
2.10 Creación de procesos. 16
2.11 Operaciones colectivas de MPI. 18
2.12 Comunicación síncrona vs asíncrona. 19
2.13 Resultados de MKL [34]. 20

3.1 Puente de hormigón armado construido a principios de los años 70 (antigüedad aproximada 60 años). 22
3.2 Arquitectura de la aplicación final. 23
3.3 Ejecución de una simulación 25
3.4 Mensajes de envío entre el *kernel* y el *server* 26
3.5 Multiplciaciones de matrices con Fortran y cuBLAS 27
3.6 Multiplicaciones de matrices incluyendo el tiempo de envío mediante MPI. 28
3.7 Simulaciones con varios *kernels*. 30
3.8 Ejecución de una simulación con Fortran y cuBLAS. 30

4.1 Red Neuronal Convolucional [45] 36
4.2 Convolución [46] 37
4.3 Proceso im2col por una sola imagen con un solo canal. 38
4.4 Funcionamiento del *stride*. 40
4.5 Resultados acumulados de la ejecución de *im2col* en las diferentes topologías de entrenamiento. 40
4.6 Funcionamiento col2im. 41
4.7 Resultados acumulados de la ejecución de *col2im* en las diferentes topologías en el proceso de entrenamiento. 43
4.8 Ejemplo de *overfitting* [47] 43
4.9 Método *dropout*. [49] 44
4.10 Ejecución de la creación de la máscara 45
4.11 Precisión de la ejecución de las topologías con CUDA y MKL 46
4.12 Comparación de la ejecución de la clasificación de MNIST. 47
4.13 Comparación del coste de la ejecución de las funciones. 48

4.14 Comparación de la ejecución de la clasificación CIFAR. 48
4.15 Comparación del coste de la ejecución de las funciones. 49
4.16 Aceleración de cuBLAS respecto MKL de las ejecuciones de las dos clasificaciones. 49

B.1 Configuración de las topologías 63
B.2 Topología MLP-Medium. 64
B.3 Topología MLP-Big. 64
B.4 Topología MLP-Large. 64
B.5 Topología convolucional. 65
B.6 Topología convolucional-16 canalaes. 66
B.7 Topología VGG1. 67
B.8 Topología VGG2. 68
B.9 Topología VGG3. 69
B.10 Topología VGG3-dropout. 70
B.11 Topología VGG3-dropout-20-30-40-50. 71
B.12 Topología VGG3-dropout-20-30-40-50-bn. 72

Índice de tablas

3.1 Protocolo de envío de mensajes 25

4.1 Dispositivos actualmente soportados. 32
4.2 Capas soportadas en HELENNA y contribución de este proyecto. 32
4.3 Funciones implementadas. 33
4.4 Funciones implementadas. 34
4.5 Funciones implementadas. 35
4.6 Funciones implementadas relacionadas con el soporte de *buffers*. 45

CAPÍTULO 1

Introducción

1.1 Sistemas de computación heterogéneos

Los sistemas de computación heterogéneos son aquellos que tienen como objetivo utilizar más de un tipo de procesador para la computación. Estos sistemas seleccionan y distribuyen las tareas a las diferentes unidades dependiendo del tipo de cálculo y su posible adecuación. De esta manera, se consigue aumentar tanto la eficiencia computacional como la energética y presentar un mayor rendimiento de las aplicaciones.

Las GPUs (*Graphical Processing Units*) han sido diseñadas para el procesamiento gráfico y las operaciones de coma flotante. Presentan un elevado paralelismo y una mayor potencia computacional respecto a la CPU (*Central Processing Unit*) dado que las GPUs contienen un número mayor de núcleos. Sin embargo, cada núcleo funciona a una frecuencia más baja y cuentan con cachés de menor tamaño. Por consiguiente, las GPUs son adecuadas para aplicaciones críticas en rendimiento o en las que se requiera operar con grandes cantidades de datos. Como se observa en la figura 1.1, la GPU está diseñada para que más recursos se dediquen al procesamiento de datos en lugar del almacenamiento en cache de datos y control de flujo [1].

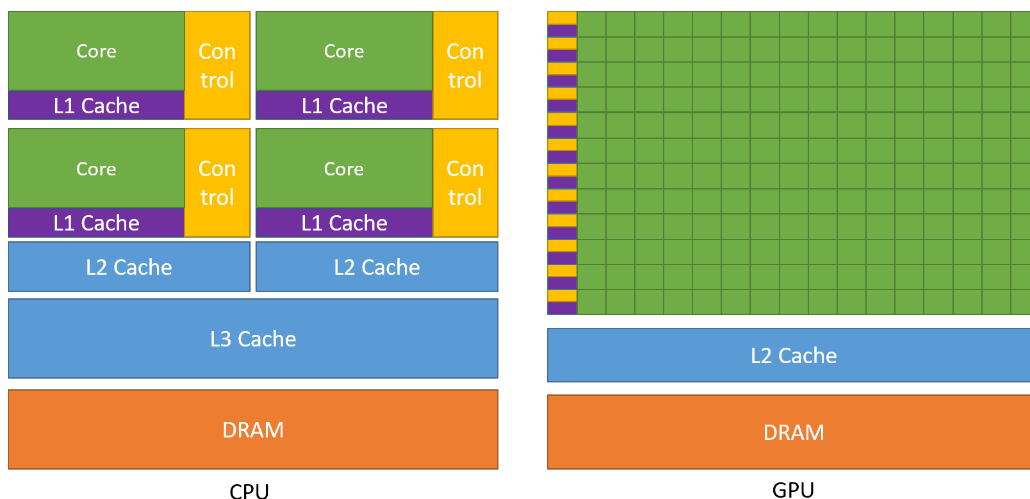

Figura 1.1: Organización típica de la CPU y la GPU [1].

Si comparamos estos dos elementos convencionales (CPUs frente a GPUs) podemos ver que las CPUs modernas utilizan, como mucho, pocas decenas de núcleos. No obstante, cada núcleo funciona a una elevada frecuencia y contiene cachés de gran tamaño. Por todo esto, las CPUs son recomendables para aplicaciones críticas en latencia. También

son recomendables en aplicaciones donde las transferencias de datos suponen el mayor tiempo de ejecución o en aquellas en las que la aplicación requiera procesar pequeñas cantidades de datos. Además, las CPUs poseen un repertorio de instrucciones más ámplio por lo que son adecuadas para un mayor espectro de aplicaciones.

Por estas razones, un sistema heterogéneo CPU-GPU proporciona una mayor eficiencia energética y computacional en aplicaciones con un conjunto de tareas diferenciadas entre sí. Estos sistemas nos permiten dar soporte a diferentes tipos de aplicaciones de cálculo intensivo como son las redes neuronales profundas o el modelado de puentes ferroviarios. **Este proyecto se centra en la optimización de estas dos aplicaciones**.

El transporte ferroviario es uno de los hitos de la modernidad y también una pieza clave de nuestros sistemas de transporte. Algunas de sus infraestructuras se construyeron hace ya varias décadas (años 50 y 60) y su estado de conservación, aunque todavía bueno, necesita cada vez más herramientas de mantenimiento predictivo eficientes.

Para su mantenimiento, la simulación por ordenador mediante cálculo numérico se ha convertido en herramienta esencial para analizar el comportamiento de los puentes ferroviarios. Para estas simulaciones se necesitan repetir cálculos idénticos para combinaciones diferentes de velocidad de paso y tipo de tren. Estos cálculos son altamente paralelizables pero requieren gestionar eficazmente los recursos de memoria y optimizar el código para realizar los millones de cálculos matriciales en el menor tiempo posible. **En este proyecto realizamos una implementación eficiente en GPUs para una aplicación de cálculo dinámico de puentes.**

También resulta interesante la aplicación de estos sistemas para las Redes Neuronales profundas (DNN; *Deep Neural Network*). Las Redes Neuronales profundas son redes neuronales artificiales (ANN; *Artificial Neural Network*) con múltiples capas conectadas entre sí, con la finalidad de resolver problemas del mundo real (figura 1.2). Estas aplicaciones realizan un conjunto de algoritmos matemáticos complejos que requieren una gran potencia computacional. Alcanzar la potencia necesaria no sería posible si solo se usara como unidad de procesamiento la CPU, por lo que, actualmente, se utilizan las GPUs para acelerar su rendimiento. **En este proyecto realizamos una implementación eficiente del cálculo en redes neuronales en GPU.**

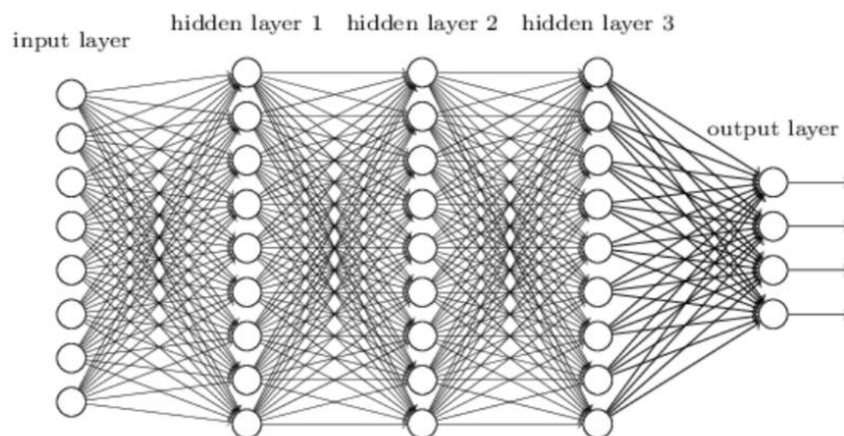

Figura 1.2: Estructura de las redes neuronales profundas [2].

Como ejemplo actual de la importancia de las redes neuronales, existe un sistema para la detección de objetos en tiempo real llamado YOLO (*You Only Look Once*) (Figura 1.3).

YOLO se basa en aprendizaje profundo y Redes Neuronales Convolucionales y destaca por ser el más rápido en su campo ya que funciona a más de 30 fotogramas por segundo (FPS)[3].

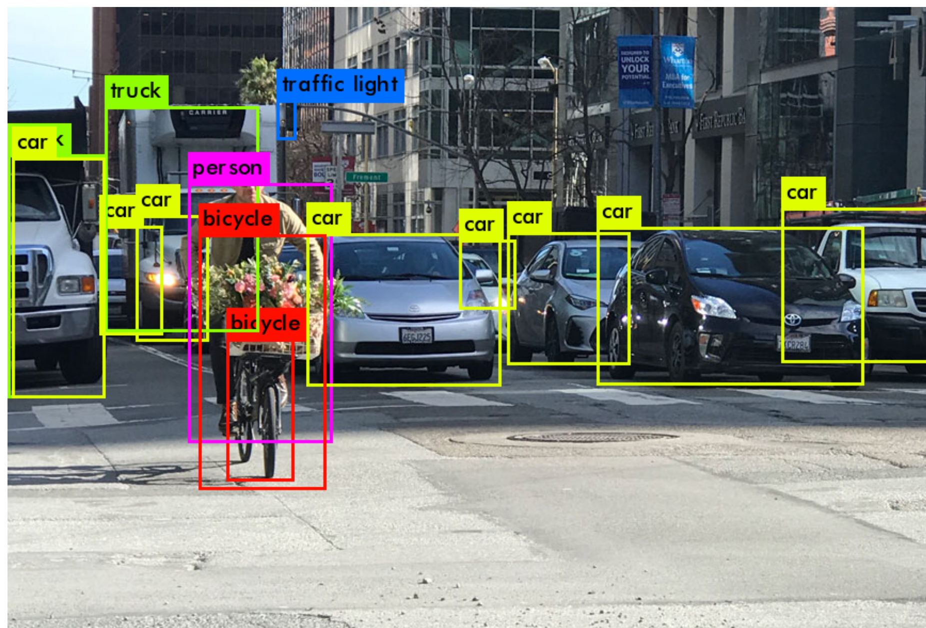

Figura 1.3: Objetos detectados por YOLO [4]

A la vez que se utilizan nuevas y mejores arquitecturas heterogéneas, la comunidad científica también utiliza la computación distribuida como una forma de mejorar aún más al rendimiento de las aplicaciones. De hecho, es común utilizar *clusters* de GPUs en el entrenamiento de redes neuronales complejas. En un *cluster* de GPUs, diferentes nodos tienen cada uno una serie de GPUs y estos nodos se conectan por medio de una red de altas prestaciones. La distribución de carga y las comunicaciones entre procesos se suele implementar por medio de la librería MPI[5], la cual permite la transferencia eficiente de mensajes. **En este proyecto aportamos una implementación eficiente de comunicación basada en MPI para la aplicación de cálculo dinámico de puentes ferroviarios.**

1.2 Motivación

Con el paso de los años, la diferencia computacional de la GPU frente la CPU se ha vuelto más notoria. Esta diferencia está reflejada en la figura 1.4, donde se observa que los GFLOPS teóricos de las GPUs son superiores a los de las CPUs modernas. Este constante aumento de la mejora de las arquitecturas GPU hace necesario el soporte continuado de librerías y software que permite el mejor rendimiento posible.

Ahora bien, no todos los campos de aplicación evolucionan adecuadamente con la mejor de las tecnologías. Tal es el caso de la simulación numérica del comportamiento dinámico de puentes ferroviarios, aspecto de importancia clave en el cálculo de estructuras de cualquier nueva línea de ferrocarril proyectada para velocidades superiores a 200 km/h (según normativa europea vigente), y también en reacondicionamiento y mantenimiento para dichas velocidades rápidas de puentes ya existentes. En este tipo de aplicaciones el uso de GPUs es todavía muy limitado y localizado en pocos entornos industriales y de investigación.

Sin embargo, y pese a esta falta de actualización en cuanto a herramientas informáticas para cálculo –las cuales permitirían acortar los tiempos de proyecto en nuevos di-

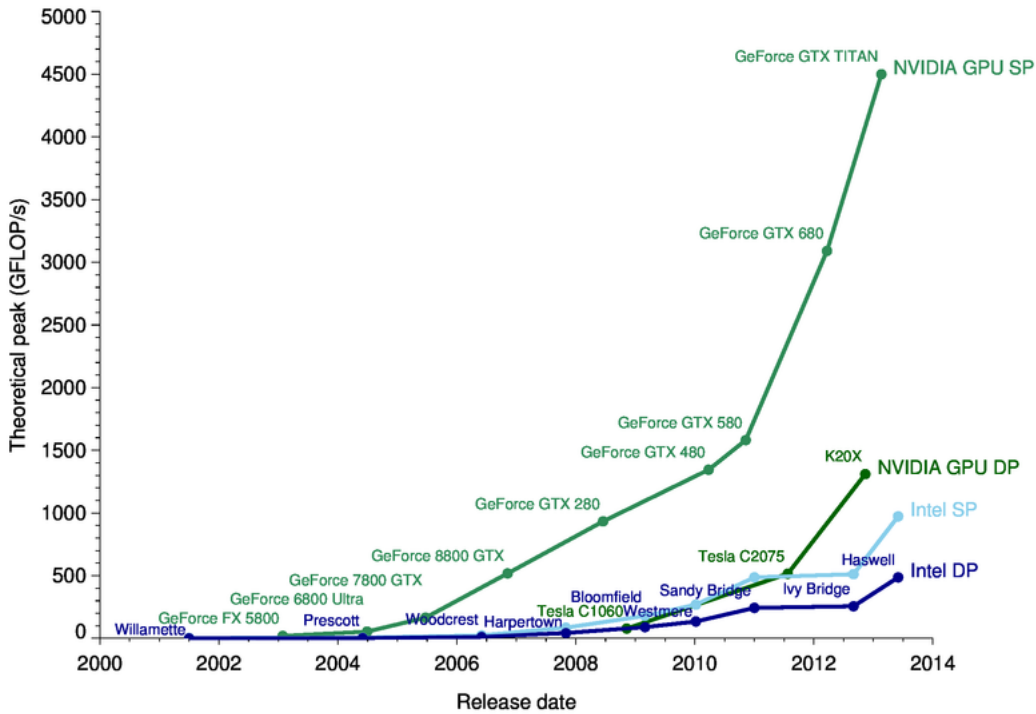

Figura 1.4: Historia del rendimiento de la CPU y la GPU [6].

seños, así como en operaciones de análisis para mantenimiento–, la importancia del ferrocarril (FFCC) se encuentra en continuo auge debido al aumento de la demanda en cuanto a número de pasajeros y transporte de mercancías. Además, más del 35 % de los puentes ferroviarios existentes en Europa tienen una antigüedad de 100 años, por lo que su fiabilidad a medio plazo impacta directamente la seguridad de la red ferroviaria [7]. Inevitablemente, se requiere la reevaluación de las estructuras ferroviarias existentes mediante un conjunto de cálculos realizados en empresas de ingeniería dedicadas a este fin. En España, ADIF publica en su página web licitaciones de reparación y mantenimiento de puentes con regularidad [8].

Por estas claras necesidades, se necesita realizar un estudio fiable (y su posterior análisis) de puentes basado en complejos cálculos de simulaciones dinámica y poder así realizar una tarea de desarrollo y mantenimiento de la red de FFCC. Para este análisis se emplea el llamado Método de los Elementos Finitos, que discretiza la estructura mediante interpolación polinómica de las ecuaciones de campo (vibraciones de medios continuos) y consigue soluciones numéricas de gran precisión.

Por otra parte, en el campo de la inteligencia artificial podemos ver un soporte totalmente diferente. El uso de librerías y software como CUDA, cuBLAS, TensorFlow o Keras permite una adaptación adecuada para sacar el máximo rendimiento a las nuevas arquitecturas. Ahora bien, el desarrollo de nuevas herramientas tanto docentes como de investigación en temas de inteligencia artificial precisa una adaptación inicial a estas arquitecturas y tecnologías. Tal es el caso de HELENNA, una aplicación para el desarrollo de procesos de entrenamiento e inferencia tanto para temas docentes como de investigación. HELENNA tiene como objetivo el uso de arquitecturas heterogéneas.

Asimismo, el uso de sistemas distribuidos debe ser soportado de forma eficiente, principalmente por el uso de la librería MPI.

1.3 Objetivos

El objetivo principal de este proyecto es el de dar soporte al uso de arquitecturas GPU tanto en una aplicación de cálculo dinámico de puentes de FFCC como en la aplicación HELENNA. La contribución del proyecto se centra en el soporte eficiente en arquitecturas NVIDIA. Ahora bien, el contexto de aplicación va a imponer múltiples aspectos a considerar como, por ejemplo, el diseño e implementación de un sistema distribuido mediante MPI, así como la adaptación de librerías de cálculo específicas o el diseño de nuevos algoritmos optimizados para GPUs. En concreto, el objetivo principal lo podemos desglosar en los siguientes subjetivos:

- Soporte de la librería cuBLAS en HELENNA.
- Soporte de las capas convencionales en HELENNA con GPUs.
- Soporte de las capas *dropout* en HELENNA con GPUs.
- Soporte de las capas *batch normalization* en HELENNA con GPUs.
- Soporte para la creación de procesos remotos con tareas diferenciadas mediante MPI en la aplicación para cálculo dinámico de puentes de FFCC.
- Soporte para la comunicación entre procesos remotos mediante MPI en la aplicación de cálculo dinámico de puentes de FFCC.
- Integración de la aplicación de cálculo dinámico de puentes de FFCC, escrito en lenguaje Fortran, con el lenguaje C.
- Soporte de la librería cuBLAS en la aplicación de cálculo dinámico de puentes de FFCC.
- Soporte para la lectura de datos mediante ficheros externos en la aplicación de cálculo dinámico de puentes de FFCC.
- Evolución y análisis de resultados para los dos dominios de aplicación.

1.4 Estructura de la memoria

El presente trabajo se divide en cinco capítulos:

- Introducción: en este capítulo especificamos los sistemas de computación heterogéneos, los objetivos a alcanzar del proyecto y la motivación para llevarlo a cabo.
- CUDA, cuBLAS, MKL y MPI: en este segundo capítulo detallamos las características principales de CUDA y su arquitectura, así como las librerías cuBLAS y MKL. También exponemos para qué sirve MPI y su funcionamiento básico.
- Aplicación para el cálculo dinámico por elementos finitos de puentes ferroviarios: en este capítulo describimos la aplicación, la contribución realizada y los resultados obtenidos.
- Aplicación de entrenamiento de redes neuronales (HELENNA): en este capítulo explicamos la aplicación, la contribución realizada y los resultados obtenidos.

- Conclusiones: en este capítulo se exponen las conclusiones extraídas de los resultados obtenidos del proyecto.
- Trabajo futuro: finalmente, se presentan las futuras mejoras a desarrollar sobre las dos aplicaciones descritas.

1.5 Colaboraciones

El presente trabajo se divide en dos grandes bloques, donde la contribución ha sido diferente. La optimización del cálculo para el entrenamiento de Redes Neuronales mediante el uso de CUDA se ha realizado de manera individual. Sin embargo, la parte relacionada con la optimización de cálculo en Modelado de Estructuras en Puentes Ferroviarios se ha implementado en colaboración con otro alumno de la Universitat Politècnica de València llamado Marc Gavilán Gil.

Concretamente, la implementación de un sistema distribuido, la cual se basa en la implementación de una estructura de procesos mediante el uso de MPI, se ha realizado conjuntamente, así como el paso de mensajes necesario entre los procesos y el protocolo creado. En cambio, la parte relacionada con la optimización de las multiplicaciones de las matrices mediante el uso de cuBLAS se ha realizado individualmente.

El trabajo se ha realizado en los laboratorios del grupo de investigación de arquitecturas paralelas (GAP) de la Universitat Politècnica de València. El trabajo ha conllevado la integración en el equipo humano que conforma el grupo y en la participación activa en un contexto de colaboración en equipo.

CAPÍTULO 2

CUDA, cuBLAS, MKL y MPI

En este capítulo describimos las diferentes herramientas que hemos utilizado para la paralelización de las dos aplicaciones mencionadas anteriormente. Principalmente, describimos el funcionamiento básico de CUDA, así como las razones que nos han llevado a elegir esta herramienta y la arquitectura que presenta. En segundo lugar, explicamos qué son las librerías cuBLAS y MKL y qué resultado se obtiene con su implementación. Finalmente, exponemos las características principales de la interfaz MPI.

2.1 CUDA

CUDA (*Compute Unified Device Architecture*) [9] es una plataforma de computación paralela y un modelo de programación creada por NVIDIA [10]. CUDA se utiliza para acelerar computacionalmente las aplicaciones altamente paralelas aprovechando las altas prestaciones de las GPUs de NVIDIA. Esta plataforma permite construir aplicaciones híbridas donde el uso de las unidades de CPU y GPU se puede realizar de manera simultánea. De esta manera, podemos construir un sistema de computación heterogéneo para aprovechar el máximo redimiento de estas dos unidades. Además, como explicaremos más adelante, permite realizar una sincronización de las dos unidades cuando lo requiera de manera sencilla.

Cabe mencionar que existen otras plataformas enfocadas a la programación de aplicaciones heterogéneas CPU-GPU. Debido a lo cual, es importante estudiar las principales diferencias de las plataformas más importantes de hoy en día para elegir cuál se adapta mejor al contexto de cada aplicación. Una de las más importantes es OpenCL[11], el principal competidor de CUDA.

OpenCL es un estándar abierto para la programación paralela de CPUs, GPUs y otros procesadores. A diferencia de CUDA, puede ser usado sobre todas las GPUs del mercado independientemente de quién sea su fabricante. Además, también encontramos clBLAS, una librería optimizada compatible con OpenCL.

Sin embargo, aunque OpenCL ofrezca más versatilidad, nos hemos centrado en la programación de GPUs mediante CUDA dado que el rendimiento que presenta CUDA respecto otros *frameworks* es muy superior. En la figura 2.1 se comparan los GFLOPS resultantes de la ejecución de diversas funciones de las librerías cuBLAS [13] y clBLAS y la ejecución de unas funciones de OpenCL y otras de CUDA. Todas estas implementaciones tienen el mismo objetivo: multiplicar dos matrices cuadradas del mismo tamaño. Como podemos ver, los GFLOPS de la implementación de la librería cuBLAS son los más altos con diferencia, ya que presenta una mejora de más de 3,8 respecto a la librería clBLAS. Además, si seguimos observando esta figura, notamos que las implementaciones me-

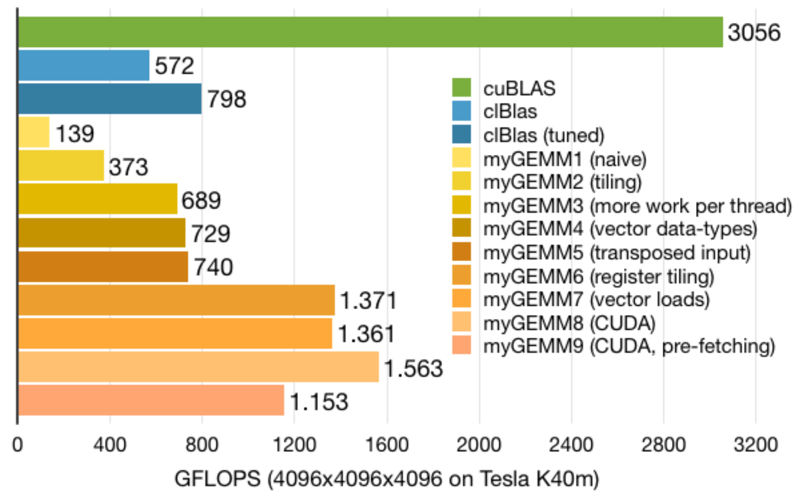

Figura 2.1: GFLOPS de diferentes implementaciones [12].

diante CUDA (myGEMM8) sigue siendo superior a la implementación más optimizada de OpenCL(myGEMM7). El principal motivo para esta diferencia de prestaciones radica en que NVIDIA provee tanto el hardware (GPUs) como el software CUDA, por lo que tiene una ventaja competitiva sobre sus rivales comerciales al explotar antes que nadie las novedades incorporadas en sus nuevas arquitecturas GPU.

Otra ventaja que convierte a CUDA en la mejor opción es la cantidad de recursos y herramientas que proporciona la propia empresa NVIDIA para ayudar a la formación de los programadores. En la página de NVIDIA, dentro de la zona dedicada a CUDA [9] podemos encontrar desde una completa documentación de las librerías hasta diversas herramientas de gran utilidad como CUDA-MEMCHECK [14], una aplicación usada para encontrar errores de código.

Otro aspecto importante de CUDA es la compatibilidad que tiene con los lenguajes de programación más usados hoy en día. Aunque CUDA use un lenguaje muy similar a C las funciones de CUDA pueden ser programados en Python, Fortran y Java.

2.1.1. Arquitectura CUDA

La arquitectura de CUDA está basada en un conjunto de *Streaming Multiprocessors* (SMs). En la figura 2.2 podemos observar la arquitectura completa de la GPU NVIDIA Tesla P100, la cual se basa en la arquitectura Pascal de NVIDIA . Concretamente, podemos ver que los elementos predominantes en esta arquitectura son los multiprocesadores SM.

Cada SM está compuesto por un conjunto de *buffers* de instrucciones, planificadores de *warp*, unidades de *dispatch* y los núcleos de cómputos llamados CUDA *cores*. Estos *cores* son los encargados de ejecutar las instrucciones del código.

En la figura 2.3, se observa la arquitectura detallada de un SM de la GPU mencionada anteriormente. Cada SM de esta arquitectura contiene 32 núcleos de doble precisión, 64 núcleos de simple precisión, dos *buffers* de instrucciones, dos planificadores de *warp*, cuatro unidades de *dispatch* y cuatro unidades de textura. El número de hilos por *warp* es de 32, mientras que el número de hilos por SM es de 64.

La arquitectura más moderna de las tarjetas de NVIDIA es la Ampere, la cual incluye los últimosm*Tensor Cores* desarrollados por NVIDIA. Estas GPUs son altamente eficientes para los cálculos sobre la Inteligencia Artificial (IA) y la computación de alto rendimiento (*High performance Computing*) [16].

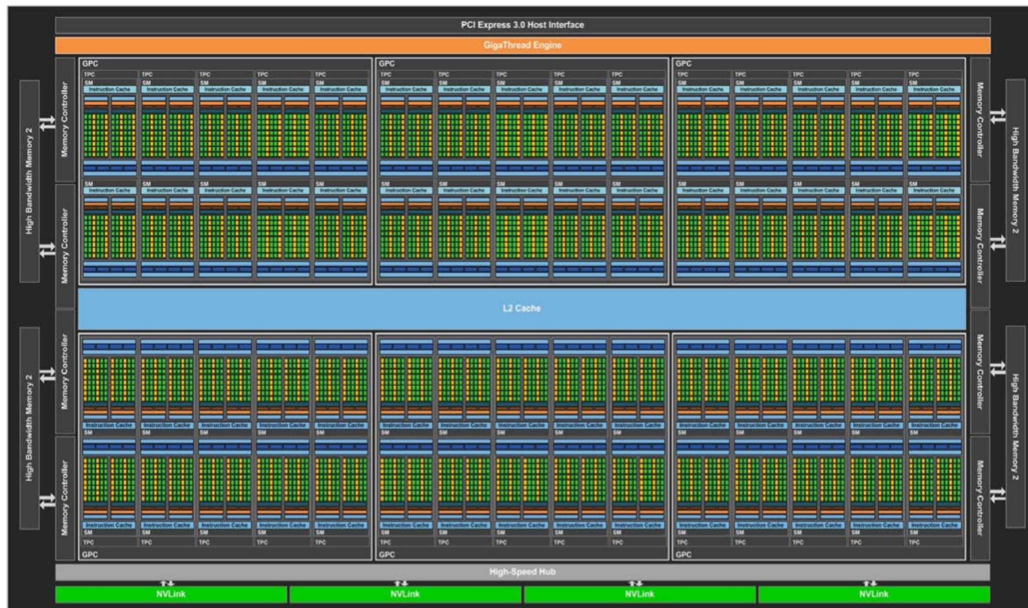

Figura 2.2: GPU Pascal GP100 [15]

Figura 2.3: SM de la GPU Pascal GP100 [15]

Los *Tensor Cores*, son unos núcleos enfocados a la computación de alto rendimiento. Estos núcleos realizan un procesamiento basado en una matemática matricial, donde realizan una suma multiplicada fusionada sobre matrices de diferentes tamaños, según el tipo de *Tensor Core* (imagen 2.4) [17].

2.1.2. Modelo de Programación

CUDA permite al programador crear funciones secuenciales en un lenguaje que extiende de C++ para poder ser ejecutadas en la GPU. Estas funciones son conocidas como

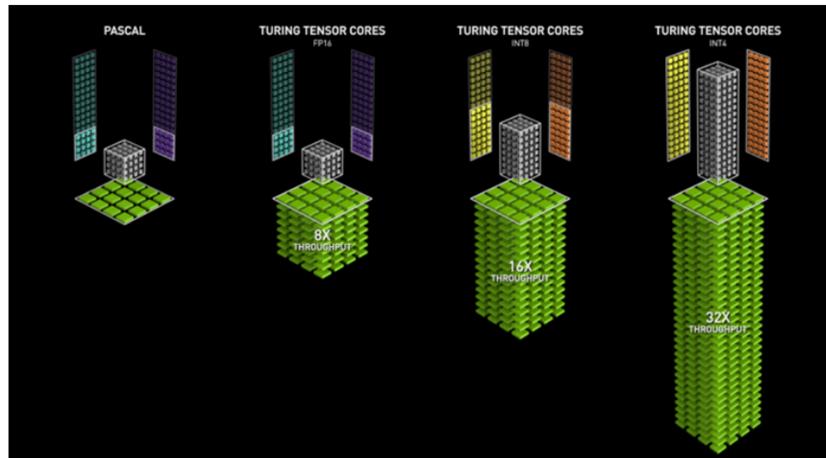

Figura 2.4: Tensor Cores. [17]

kernels y son subrutinas capaces de ejecutarse paralelamente por un conjunto de hilos. Un *kernel* se define usando la etiqueta *__global__* y, para poder lanzarlo, se necesita especificar el número de hilos que ejecutará dicho *kernel*. A cada hilo se le proporciona un identificador único para poder acceder a él desde el *kernel* mediante las variables de CUDA [1].

La agrupación de un conjunto de hilos representa un bloque de hilos. Antiguamente, el número máximo de hilos que podía agrupar un bloque era de 512. Sin embargo, aunque la capacidad varíe según la limitación de la arquitectura de cada GPU, el número máximo actual es de 1024. Todos los hilos de un bloque se ejecutan en el mismo SM. CUDA también ofrece la posibilidad de que los hilos de un mismo bloque puedan comunicarse y sincronizarse.

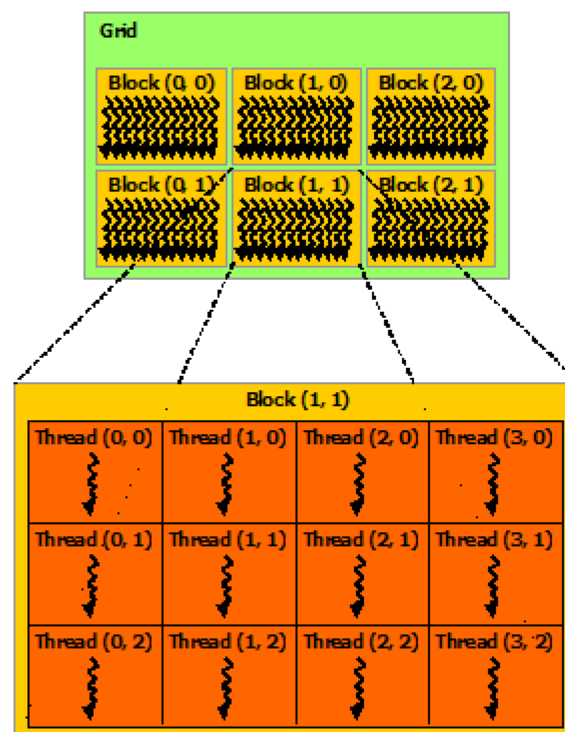

Figura 2.5: Modelo de programación [1].

Como se observa en la figura 2.5, los bloques se organizan en mallas de una, dos o tres dimensiones. El tamaño de los bloques de una malla suele estar calculado según el tamaño de los datos a procesar. Con este diseño podemos conocer el identificador de cada hilo dado que CUDA permite calcular la dimensión y el ID de cada bloque. Además, dentro de cada bloque podemos saber el ID del hilo [1].

El SM ha sido diseñado para ejecutar cientos de hilos paralelamente. Para poder hacer esto posible, se utiliza la arquitecuta SIMT *(Single-Instruction, Multiple-Thread)*. Esta arquitectura se basa en ejecutar los hilos en grupos de 32 hilos paralelos, lo que se conoce como *warp*. Los hilos de cada *warp* se ejecutan en paralelo y empiezan por la misma instrucción. Sin embargo, pueden ramificarse y ejecutarse independientemente. La máxima eficiencia se obtiene cuando los 32 hilos de cada *warp* coinciden en la ruta de ejecución ya que si los hilos de un *warp* divergen a través de una rama condicional dependiente de datos, el *warp* ejecuta cada ruta de rama tomada, deshabilitando los hilos que no están en esa ruta [1].

2.1.3. Gestión de Memoria

Referente a la memoria, en un primer lugar, nos encontramos con la memoria *host*, la cual corresponde a la memoria accesible mediante la CPU del sistema. Esta memoria es paginable (*pageable*), lo que significa que la ubicación física de la memoria puede cambiar sin previo aviso. Debido a esto, los periféricos como las GPUs no pueden acceder a ella. Para habilitar el acceso directo a esta memoria, los sistemas operativos deben permitir que la memoria del *host* quede estática. CUDA incluye una API que facilita esta característica al sistema operativo para un acceso directo a esta memoria [18]. Sin embargo, no vamos a profundizar en este tipo de memoria dado que no ha sido desarrollada en este trabajo.

Además, existe otro tipo de memoria la cual está localizada en la GPU y se accede a ella mediante un controlador de memoria dedicado. Este tipo de memoria se conoce como memoria *device*. Para utilizar este tipo de memoria, los datos deben copiarse explícitamente entre el espacio de memoria del *host* y del *device* para que la GPU pueda procesarlos. En este proyecto, nos centramos en el uso de la memoria de la GPU.

En la figura 2.6 se muestra la jerarquía de memoria *device* de CUDA. Cada hilo tiene memoria local privada y cada bloque tiene memoria compartida visible para todos los hilos del mismo bloque. Además, todos los hilos de la GPU tienen acceso a una zona de memoria global. Cabe destacar que la memoria compartida es más rápida que la local y la global. La latencia de la memoria compartida es 100 veces más baja que la global, por lo que una buena práctica es copiar los datos que van a ser accedidos habitualmente por los hilos en la zona de memoria compartida [19].

2.1.4. Sincronización

Al utilizar la memoria compartida paralelamente entre los hilos, es importante evitar las condiciones de carrera que puedan desencadenar un resultado erróneo. Para esto, cuando los hilos cooperan en paralelo, debemos sincronizarlos. CUDA nos ofrece una solución fácil a este problema, ya que propone la utilización de barreras mediante la primitiva *__syncthreads()*. Esta barrera obliga a que la ejecución solo continúe cuando todos los subprocesos hayan llegado a dicha barrera. De este modo, podemos evitar las condiciones de carrera llamando a esta función después de almacenar datos en la memoria compartida y antes que algún hilo necesite cargar datos de la memoria compartida [19].

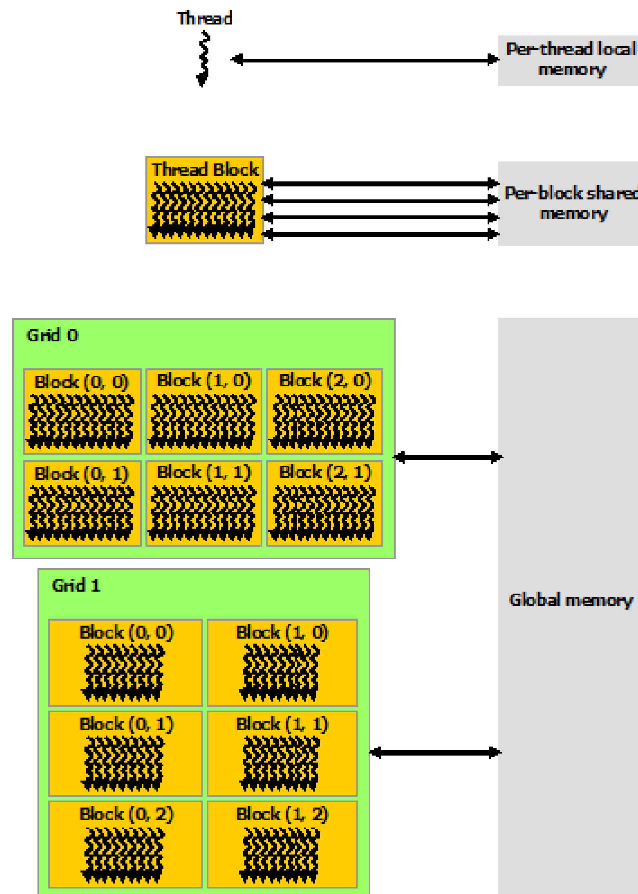

Figura 2.6: Jerarquía de memoria [19].

Además, las llamadas a CUDA se realizan de manera asíncrona. Normalmente, cuando se necesitan datos albergados en la memoria de la GPU, se utiliza la función *cudaMemcpy*, la cual realiza una sincronización de la GPU con la CPU para que los datos sean correctos. Sin embargo, en un escenario donde se ejecuten diversas funciones de la GPU, puede que se necesite forzar la sincronización para, por ejemplo, calcular correctamente las prestaciones de cada función de CUDA, como ocurre en este proyecto. Esta sincronización se puede lograr mediante la función *cudaDeviceSynchronize*.

2.1.5. *Streams*

CUDA tiene un modelo de concurrencia basado en *streams*. Estos *streams* permiten ejecutar diversas tareas paralelamente dentro de la GPU. Con este método se intenta conseguir explotar la paralelidad que ofrece la GPU. Estas tareas pueden ser diferentes *kernels*, funciones de librerías como cuBLAS o copias entre la GPU y la CPU.

En la figura 2.7 podemos observar diversas posibles implementaciones de una copia de datos entre la CPU y la GPU, la ejecución de un *kernel* y, finalmente, la copia de los resultados desde la GPU hacia la CPU. Como podemos observar, se obtiene una clara mejora si se hace uso de los *steams* para sobreponer estas tareas. El uso de los *steams* no ha sido implementado en este proyecto.

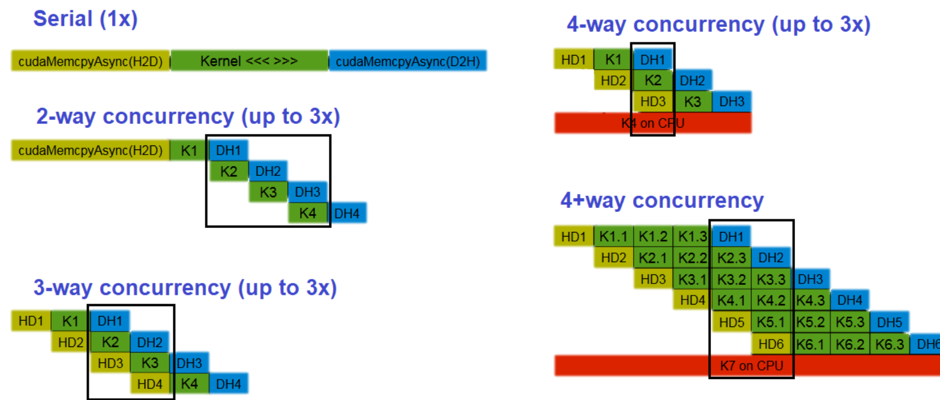

Figura 2.7: Mejoras con el uso de *streams* [20].

2.2 cuBLAS

La librería NVIDIA cuBLAS es una implementación de la librería BLAS (Basic Linear Algebra Subprograms) para la plataforma CUDA, la cual contiene rutinas de álgebra lineal que sirven para realizar operaciones básicas sobre vectores y matrices. Con la API de cuBLAS se puede acelerar las aplicaciones mediante la implementación de funciones de cómputo intensivas en la GPU. Esta librería sirve para desarrollar algoritmos acelerados por la GPU en áreas de computación de alto rendimiento, aprendizaje automático o análisis de imágenes. [21]

En la figura 2.8 se refleja la comparación entre cuBLAS y MKL[1] [22] sobre la función ZGEMM [23]. En los dos casos, esta función realiza la multiplicación de dos matrices. Podemos observar cómo crece la aceleración respecto a MKL según aumenta el tamaño de las matrices a ser multiplicadas. Para la función de cuBLAS se utiliza la GPU Tesla K40M de NVIDIA y para MKL la CPU Intel IvyBridgesingle socket 12-core E5-2697 v2 @ 2.70GHz.

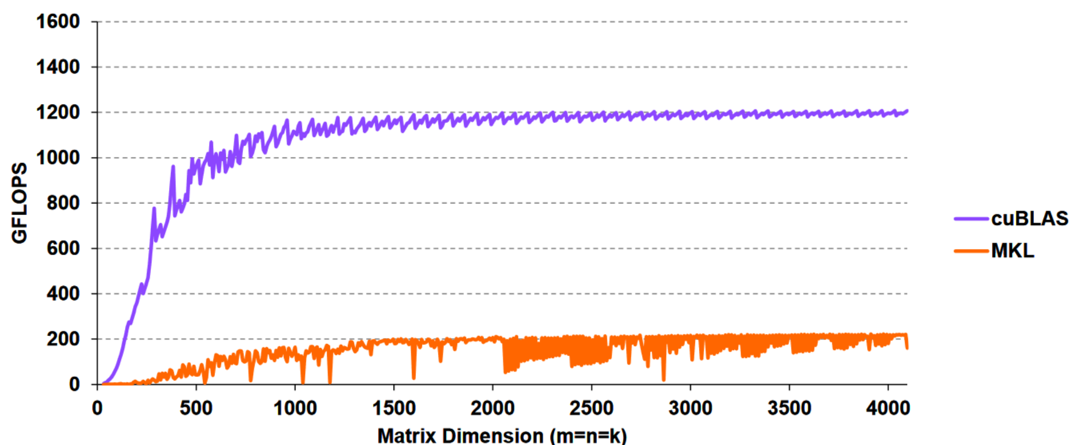

Figura 2.8: Comparación entre cuBLAS y MKL [24].

[1]MKL es una libreria de Intel con el mismo objetivo pero que se utiliza en CPUs. Posteriormente describimos MKL.

Además, cuBLAS también ofrece una mejora de resultados frente a la librería clBLAS. En la figura 2.9 se han plasmado los GFLOPS resultantes de multiplicar dos matrices cuadradas en varias implementaciones usando la misma GPU. Podemos observar cómo cuBLAS está muy por encima de las implementaciones comparadas.

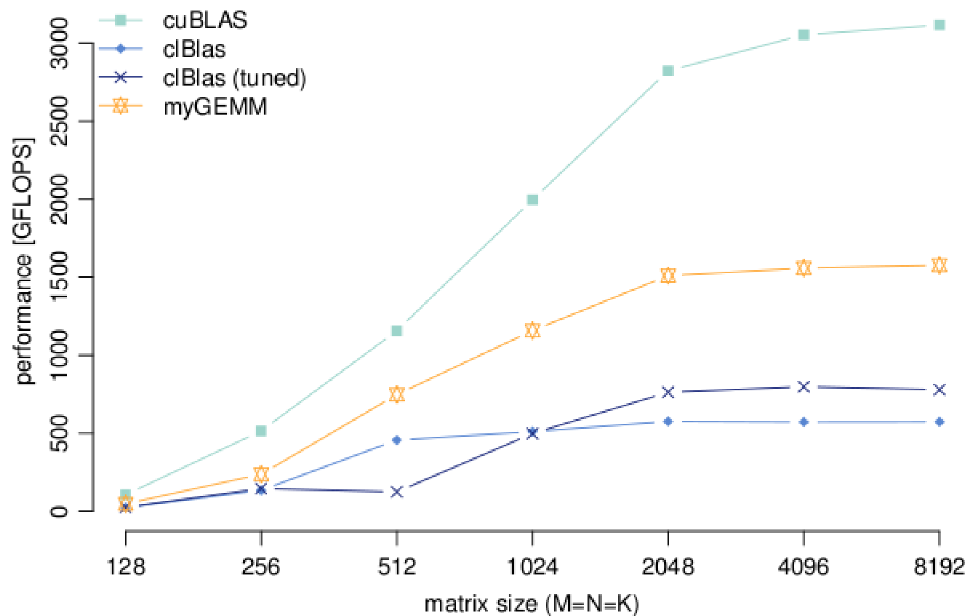

Figura 2.9: Comparación entre cuBLAS y clBLAS [25].

Esta librería, al igual que las BLAS, está categorizada en 3 niveles según el tipo de operación que realiza. En el nivel 1 encontramos las operaciones escalares sobre vectores y productos escalares, en el nivel 2 operaciones matriz-vector y, finalmente, en el nivel 3 operaciones matriz-matriz. CuBLAS permite cierta flexibilidad de datos ya que cada rutina admite diferentes tipos de datos. [13]

La librería de cuBLAS expone tres APIs diferentes [13]:

- CuBLAS API: es la que hemos utilizado para otorgar paralelismo a nuestras aplicaciones. Para utilizar esta API los datos deben estar guardados en la memoria de la tarjeta gráfica. CuBLAS contiene funciones para poder escribir y leer los datos desde la memoria de la GPU.

- CuBLASXT API: expone una interfaz de *host* compatible con múltiples GPUs. Las matrices solo necesitan estar almacenadas en la memoria del *host*. CublasXt se encarga de asignar la memoria a través de una o más GPUs y distribuye la carga de trabajo entre ellas y, finalmente, recupera los resultados de nuevo al *Host*. La API cublasXt solo admite las rutinas de nivel 3 donde las transferencias PCI de ida y vuelta desde la GPU se pueden amortizar.

- CuBLASLt API: cuBLASLt es una librería dedicada a las operaciones GEMM (*GEneral Matrix-to-matrix Multiply*). Esta biblioteca agrega flexibilidad en diseños de datos de matriz, tipos de entrada, tipos de cómputo y también en la elección de implementaciones algorítmicas y heurísticas a través de la capacidad de programación de parámetros.

Dado que las librerías originales BLAS estaban escritas en FORTRAN, cuBLAS asume un método de almacenamiento por columnas e indexación basada en 1 (*1-based indexing*)

que corresponde a este lenguaje. Este método no encaja con el método de almacenamiento por filas existente en el lenguaje C/C++ que se utiliza hoy en día. Por esta razón, las aplicaciones escritas en los lenguajes con almacenamiento por filas no pueden usar la semántica de matrices bidimensionales nativa [13].

Si la aplicación realiza tareas de cálculo independientes, se pueden utilizar los llamados *streams* de cuBLAS para superponer el cálculo de estas tareas. Para lograr esto, el usuario debe crear tantos *streams* como necesite y asignar a cada uno de ellos la tarea correspondiente. Luego, el cálculo realizado en flujos separados se superpondrá automáticamente cuando sea posible en la GPU. Cabe destacar que este enfoque es especialmente útil cuando el cálculo a realizar por una sola tarea es relativamente pequeño y no es suficiente para llenar la GPU con trabajo. De esta manera, podemos aprovechar toda la capacidad de la GPU. Actualmente, el número máximo de núcleos en ejecución es de 32. [13]

Cabe decir que con múltiples tareas pequeñas no podemos conseguir la misma tasa de GFLOPS que con una tarea grande. Sin embargo, en caso de que los cálculos de la aplicación no supongan un porcentaje elevado del total ejecución de la GPU y nos encontremos ante tareas pequeñas, realizarlas en conjunto serían una mejor opción en comparación a hacerlo secuencialmente [13].

2.3 MPI

MPI (*Message Passing Interface*) [5] es un interfaz estandarizado para el desarrollo de aplicaciones paralelas basadas en paso de mensajes. Gracias a MPI podemos crear aplicaciones para ser ejecutadas en *clusters* de alto rendimiento (HPC; *High Performance Computing*).

Cabe aclarar que MPI se trata de una especificación, no de una implementación ni de un lenguaje. Permite otorgar al programador una colección de funciones para C y Fortran y está enfocado a aquellos programadores que deseen crear aplicaciones paralelas. Existen diversas implementaciones de MPI. Las implementaciones de código abierto más comunes son OpenMPI [26] y MPICH [27]. Esta última ha sido la utilizada en este proyecto.

2.3.1. Procesos

Un programa MPI está formado por un conjunto de procesos autónomos creados mediante un conjunto de prácticas explicadas más adelante. Estos procesos ejecutan su propio código los cuales no han de ser, necesariamente, idénticos. Para la comunicación entre los procesos, se utilizan las operaciones colectivas MPI o comunicación básica basada en *MPI_Send* y *MPI_Recv*. Generalmente, cada proceso tiene su propia memoria local, por lo que los procesos no necesitan compartir memoria. Sin embargo, existen implementaciones de memoria compartida.

Para poder realizar el paso de mensajes de una manera sencilla, MPI asigna un identificador a cada uno de los procesos creados con valores desde 0 hasta n-1, donde n es el número de procesos totales. Estos identificadores, llamados *rank*, son cruciales para el control de la ejecución del programa, ya que para todos los mensajes se necesita identificar al emisor y al receptor, lo cual se realiza mediante este *rank*.

2.3.2. Comunicadores

Otro aspecto importante en MPI son los comunicadores. Todos los intercambios de mensajes se realizan entre procesos pertenecientes al mismo comunicador especificado. Esto significa que, cuando queremos realizar un intercambio de mensajes, debemos indicar sobre qué entorno de comunicación queremos que se realice para asegurarnos de que solo los procesos necesarios forman parte de la operación colectiva deseada. Un comunicador es una colección de procesos que pueden interactuar entre sí. Existe un comunicador predefinido en MPI, el cual permite la comunicación entre todos los procesos creados después de realizar la inicialización, llamado *MPI_COMM_WORLD*. Sin embargo, en un sistema donde los procesos se crean dinámicamente, este comunicador puede representar simultáneamente grupos disjuntos de diferentes procesos [28].

2.3.3. Creación de Procesos

Los programas MPI pueden lanzarse con múltiples procesos y pueden crearse otros dinámicamente [29]. En primer lugar, para lanzar una aplicación MPI se necesita el comando *mpiexec*. Este comando ofrece diferentes argumentos de gran utilidad. Sin embargo, a continuación, se destacan los relevantes para este proyecto.

Si se desea crear un conjunto de procesos iguales al lanzar la aplicación, podemos indicarlo mediante el argumento -n seguido del número de procesos deseado. El comando *mpiexec*, también permite indicar en qué máquina se desean crear los procesos mediante el argumento -*host* seguido del nombre de la máquina.

Además, MPI ofrece la posibilidad de crear procesos dinámicamente los cuales pueden ejecutar código diferenciado. En un primer lugar, mediante *MPI_Comm_spawn*, se permite la creación de un grupo de procesos en máquinas remotas. Este grupo de procesos ejecutará el programa indicado, pudiendo ser uno diferente al que está siendo ejecutado por el proceso que ha llamado a dicha función.

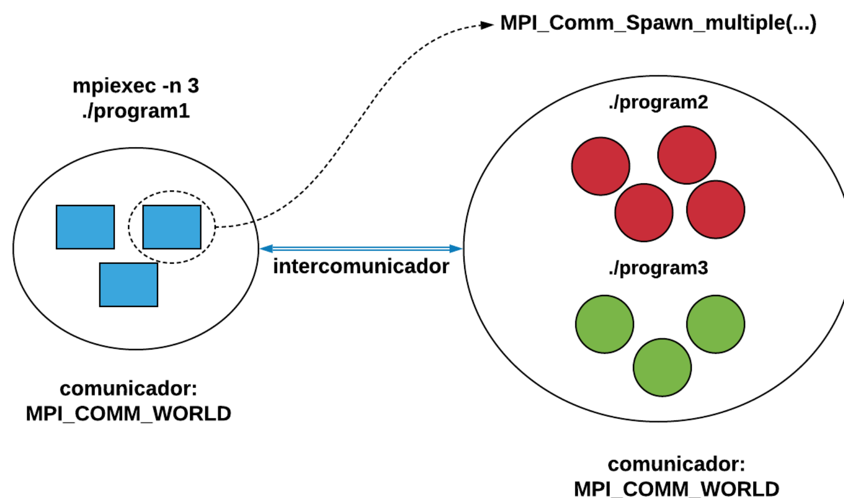

Figura 2.10: Creación de procesos.

Además, existe otra función similar a la anterior llamada *MPI_Comm_spawn_multiple*. Esta función permite la ejecución de más de un código llevada a cabo por diferentes procesos creados dinámicamente, y es la que se ha utilizado en este proyecto. Este funcionamiento está reflejado en la figura 2.10. En esta figura podemos observar un conjunto de procesos iniciales, los cuales han sido creados mediante el comando *mpiexec*. Más tarde,

uno de estos procesos ejecuta la función *MPI_Comm_spawn_multiple*, donde crea cuatro procesos que ejecutan el programa *./program2* y otros tres que ejecutan el programa *./program3*.

Otro aspecto a destacar de la creación dinámica de procesos es la necesidad de un intercomunicador para conectar el grupo de procesos creados inicialmente mediante *mpiexec* y los procesos creados dinámicamente por alguna de estas funciones. Tanto la función *MPI_Comm_spawn_multiple* como *MPI_Comm_spawn* crean un intercomunicador que sirve para comunicar el proceso padre que ha llamado a la función con el nuevo grupo de procesos hijos. Además, cuando un proceso hijo desee comunicarse con el padre, necesitará llamar a la función *MPI_Comm_get_parent* y utilizar este. Finalmente, los procesos hijos se pueden comunicar entre ellos usando su propio comunicador global *MPI_COMM_WORLD*, el cual tiene un identificador diferente que el comunicador global de los procesos iniciales. Todo esto se puede observar en la figura 2.10.

Dado que esta función crea los procesos de manera que sus rangos dentro de su propio comunicador *MPI_COMM_WORLD* corresponden directamente al orden en que se especifican los comandos en esta función, podemos conocer los identificadores de cada proceso creado, lo cual simplifica la gestión de paso de mensajes.

2.3.4. Tipos de Mensajes

Un mensaje MPI está compuesto por una dirección inicial de memoria (*buffer*) donde se encuentra el conjunto de datos de salida o la región donde van a ser almacenados, el identificador de los procesos implicados en el envío de dicho mensaje, el tipo de dato, el número de elementos, la etiqueta de cada mensaje y el comunicador.

El tipo de dato debe indicarse mediante los tipos de datos primitivos gestionados por MPI, los cuales pueden ser equivalentes a los existentes en C o Fortran. Sin embargo, MPI también ofrece la posibilidad de definir tipos más complejos no homogéneos (*struct*) para proveer de flexibilidad la construcción de estructuras en aplicaciones más avanzadas.

Todos los mensajes, tanto los básicos como las operaciones colectivas, llevan una etiqueta (*tag*) para poder diferenciar los envíos. Mediante *MPI_ANY_TAG* podemos aceptar cualquier mensaje, independientemente de la etiqueta. Además, existe la posibilidad de conocer información adicional sobre los mensajes recibidos mediante el argumento *status*. Gracias a este objeto podemos conocer quién ha enviado el mensaje, la etiqueta del mensaje o el tamaño del mismo.

2.3.5. Transferencia de Mensajes

Para la transferencia de mensajes entre procesos se realizan llamadas. Mediante estas llamadas se consigue el traspaso de datos desde una dirección de memoria de un proceso emisor a una dirección de memoria de otro proceso receptor. Este sistema de memoria facilita la programación de aplicaciones paralelas sobre sistemas distribuidos. Los procesos pueden ejecutar llamadas básicas y operaciones colectivas.

Las comunicaciones punto a punto se basan en las llamadas a las funciones *MPI_Send()* y *MPI_Recv()*. Estas funciones son simples, ya que, simplemente, el proceso que ejecute dicha llamada enviará o esperará la recepción de un mensaje, respectivamente.

Existen diferentes operaciones colectivas (comunicaciones multipunto a punto o punto a multipunto) que facilitan los diferentes tipos de envíos posibles de mensajes entre los procesos, las cuales se pueden observar en la figura 2.11. Encontramos la posibilidad de realizar una difusión de un mismo mensaje, desde el proceso emisor (o también llama-

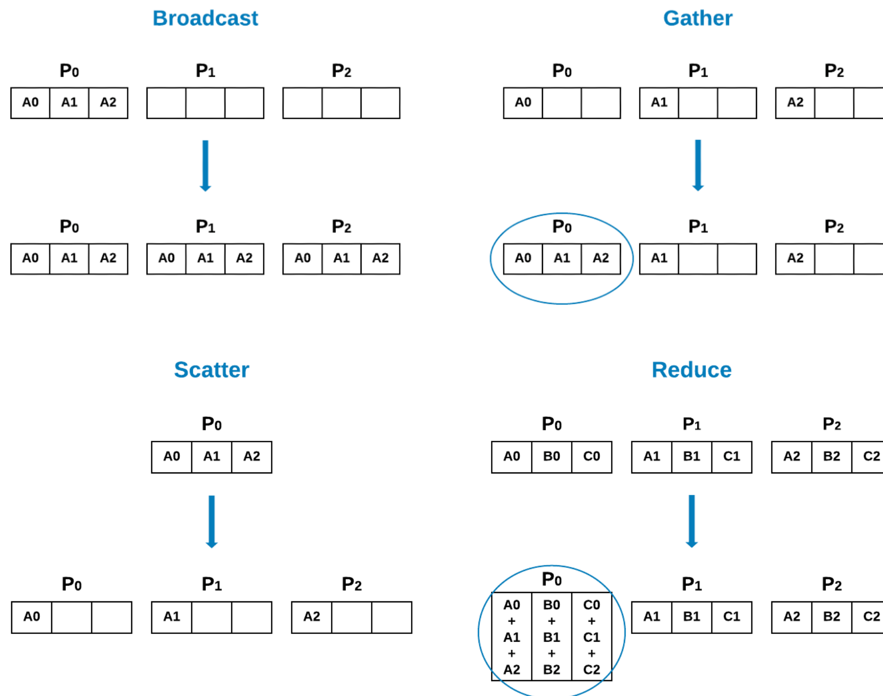

Figura 2.11: Operaciones colectivas de MPI.

do proceso *root*) hacia los diferentes procesos dentro de un mismo comunicador. Esta difusión es comúnmente llamada como *MPI_Bcast*. También tenemos la posibilidad de recolectar un conjunto de datos provenientes de distintos procesos (incluido el proceso destino) dentro de un *buffer* de recepción alojado en el proceso *root*, mediante la colectiva *MPI_Gather*.

Encontramos otras colectivas interesantes como *MPI_Scatter*. Esta colectiva puede usarse para la distribución de un conjunto de datos del proceso *root* en los *buffers* de los otros procesos que se encuentran en el mismo comunicador, de manera que cada proceso obtiene un subconjunto de estos datos. Por último, también podemos encontrar llamadas que realizan operaciones sobre los datos de los procesos. Estas últimas llamadas sirven para el cálculo de los datos y el envío de los resultados de manera óptima. Una de estas colectivas se llama *MPI_Reduce*, la cual sirve para reducir los datos de entrada y dejarlos en el proceso *root*.

Otra operación colectiva importante es *MPI_Barrier*. Esta, aunque no sirve para el envío de mensajes, es importante para forzar una sincronización. Mediante esta función, logramos bloquear todos los procesos del mismo comunicador indicado en un mismo punto hasta que todos los procesos hayan ejecutado la barrera.

2.3.6. Tipos de comunicaciones

El paso de mensajes en MPI se puede realizar de manera síncrona o asíncrona. Estas comunicaciones quedan plasmadas en la figura 2.12 donde, se puede observar las problemáticas que pueden surgir por la implementación de estas comunicaciones.

En el paso de mensajes síncrono el proceso emisor quedará bloqueado hasta que el proceso receptor reciba el mensaje. Esta modalidad es la más segura dado que, al realizarse un bloqueo hasta que el envío finalice, se garantiza que el mensaje recibido no se vea afectado. Sin embargo, una mala implementación de este método puede provocar que

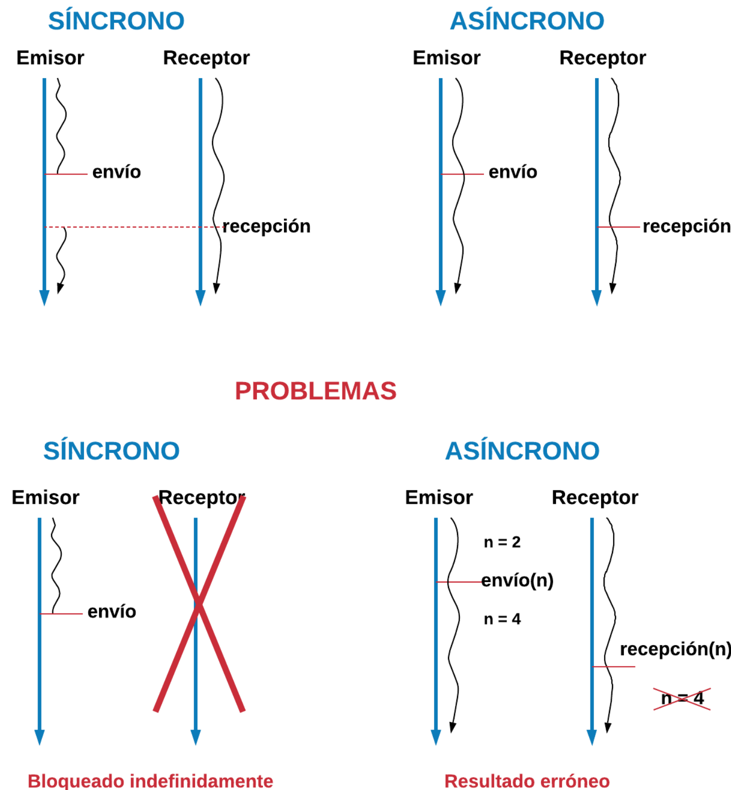

Figura 2.12: Comunicación síncrona vs asíncrona.

el proceso emisor entre en un bloqueo indefinido en caso de que no hubiera un proceso esperando la recepción de dicho mensaje.

En el envío asíncrono, el proceso emisor, al ejecutar la instrucción de envío de los datos, no se bloquea y sigue la ejecución del código. De esta manera, es posible que ejecute otra instrucción de envío, aunque el anterior mensaje no haya sido recibido. Por este motivo, existen *buffers* donde se almacenan los mensajes que están a la espera de ser recibidos. Cuando estos *buffers* están llenos el proceso emisor se bloqueará. Gracias a este tipo de envíos, la eficiencia de la aplicación puede verse altamente incrementada. No obstante, dado que no conocemos cuanto tiempo se tardará en enviar el mensaje, si el programador cambia el contenido del mensaje a enviar antes de que el receptor empiece la instrucción de recepción, dicho mensaje será diferente al que inicialmente queríamos enviar.

2.4 MKL

Intel Math Kernel Library (MKL) [22] es una librería desarrollada por la empresa Intel que proporciona rutinas de álgebra lineal BLAS [30] y LAPACK [31], FFT (fast Fourier transforms), funciones matemáticas vectoriales, funciones de generación de números aleatorios y otras funcionalidades. MKL incluye rutinas y funciones optimizadas para aplicaciones científicas, de ingeniería o financieras que se ejecuten sobre procesadores Intel y se encuentra disponible para los sistemas operativos Windows, Linux y Mac OS. [32] Otra característica de esta librería es que proporciona interfaces C y Fortran, y puede usarse desde diferentes lenguajes de alto nivel, incluidos Python, C# y Java. [33]

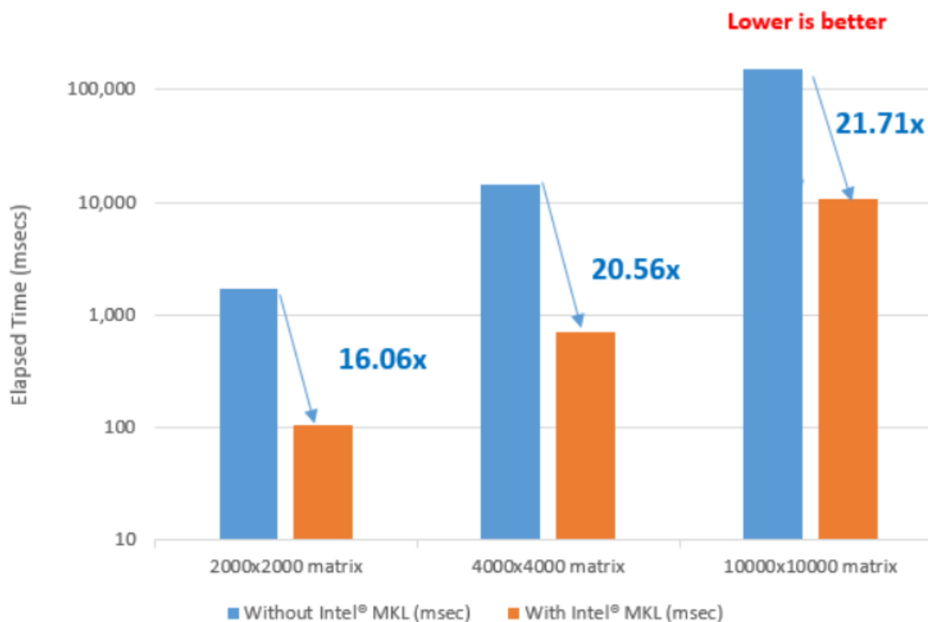

Figura 2.13: Resultados de MKL [34].

Profundizando en las prestaciones que ofrece esta librería, en la figura 2.13 podemos observar los resultados calculados de la multiplicación de matrices cuadradas utilizando el procesador Intel® Xeon® Platinum 8180 2.5 GHz con 28 núcleos. Los resultados calculados con la librería MKL son, aproximadamente, 21 veces mejor que si no utilizamos esta librería.

La librería MKL es uno de los dispositivos soportados por HELENNA y, en este trabajo, se emplea para realizar una comparación representativa de los resultados obtenidos mediante CUDA.

CAPÍTULO 3

Aplicación para Cálculo Dinámico por Elementos Finitos de puentes ferroviarios

En este capítulo se describe la aplicación para cálculo dinámico de puentes de ferrocarril (FFCC), que fue presentada en un congreso celebrado en Oporto en 2019 [35]. Esta aplicación simula el paso de trenes rápidos a distintas velocidades por un puente y ha sido desarrollada en la Escuela Técnica Superior de Ingeniería de Caminos, Canales y Puertos de la Universitat Politècnica de València con el fin de determinar el estado de dicho puente mediante un análisis probabilístico que emplea como núcleo modelos de elementos finitos para las estructuras y modelos de masas, muelles y disipadores para el tren. El código original se desarrolló en la Universidad Politécnica de Madrid, en el marco de la tesis doctoral del profesor Pedro Museros Romero, de la UPV. En este capítulo también se plasman las estrategias llevadas a cabo para paralelizar la aplicación original.

3.1 Contexto de la Aplicación

Las infraestructuras ferroviarias, según la norma española ITPF-05 [36], necesitan ser conservadas mediante inspecciones periódicas a fin de reducir riesgos de seguridad y costes de mantenimiento. En lo que refiere a los puentes de ferrocarril (figura 3.1), su conservación se ve dificultada, principalmente, por dos necesidades en constante crecimiento con el paso del tiempo a causa del incremento del uso de las líneas. Una de ellas es el potencial aumento de la velocidad de paso de los trenes en busca de un servicio más rápido que cubra las necesidades de desplazamiento de la ciudadanía. Además, el deseo de incrementar la capacidad para mercancías y, también, para transportes que no son aptos para carretera incrementan las capacidades de las cargas máximas permitidas por eje, ya que siempre han aumentado desde los orígenes del ferrocarril.

Por estas razones, para alcanzar las características necesarias de fiabilidad y seguridad es imprescindible, entre otros factores, la actualización de los métodos de cálculo y experimentación para realizar un seguimiento fiable de la natural evolución de los puentes. Esta modernización se vuelve imprescindible ante el envejecimiento natural de las estructuras. Más aún, habida cuenta de que el cálculo presenta en numerosas ocasiones un grado no menor de incertidumbres, se imponen cada vez con mayor frecuencia los cálculos probabilistas, basados en la realización de un elevado número de simulaciones. Ello redunda en la necesidad de aumentar la velocidad de proceso para poder repetir decenas de miles de cálculos y elaborar un tratamiento estadístico de la incertidumbre,

Figura 3.1: Puente de hormigón armado construido a principios de los años 70 (antigüedad aproximada 60 años).

que permita acotar las probabilidades de fallo de los diferentes elementos estructurales [37].

En cuanto a aceleración del cálculo, dos son los retos principales a abordar:

- La aceleración de miles, o decenas de miles de cálculos semejantes mediante paralelización.
- Minimizar el coste computacional de cada uno de dichos cálculos, en especial en casos de problemas no lineales, donde las matrices de los sistemas a resolver pueden alcanzar dimensiones de miles de grados de libertad.

3.2 Descripción de la Aplicación

La aplicación de partida realiza simulaciones de pasos de trenes sobre unos determinados puentes y se encuentra escrita íntegramente en lenguaje Fortran. Además, tiene como entrada de datos un conjunto de ficheros los cuales contienen los parámetros descriptivos tanto de los puentes ferroviarios y como de los trenes. El programa realiza un análisis no determinista simulando el paso del tren descrito en el fichero de entrada sobre variaciones aleatorias de un mismo puente. Estas variaciones sobre los parámetros del puente se crean siguiendo el método de Monte Carlo, asumiendo distribuciones habituales de probabilidad para las variables del modelo: masa lineal, rigidez, dimensiones de la sección transversal del tablero y tasa de amortiguamiento [35]. Para generar dichas variables de entrada aleatorias se emplea un *script* externo de Matlab desarrollado por el profesor Roberto Palma Guerrero, de la Universidad de Granada, en colaboración con el profesor Museros. Un caso típico de aplicación tendrá por encima de 100 simulaciones aleatorias en una primera aproximación, que posteriormente suele refinarse para disminuir las incertidumbres en los resultados.

Las diferentes simulaciones generadas aleatoriamente, a su vez, deben lanzarse fijando como parámetro de entrada una velocidad diferente de paso del tren, dentro de un rango determinado que capture los posibles fenómenos resonantes del puente. Cuando la velocidad de paso es igual (o próxima) a la frecuencia de vibración del puente, multiplicada por la longitud de cada vagón (o submúltiplos de dicha longitud), puede producirse resonancia y debe analizarse.

Valores típicos pueden ser 60 velocidades diferentes por caso, llegándose así a $100 \times 60 = 6000$ simulaciones en total por cada tren a estudiar. Cada simulación concreta ejecuta, pues, un caso de prueba, el cual corresponde el paso de un tren a una velocidad

determinada, con un valor particular de los parámetros del puente. Para realizar la simulación se necesitan un conjunto de cálculos necesarios como rutinas complejas o multiplicaciones de matrices. Dado que esta aplicación de partida se ejecuta solamente en una máquina y únicamente utiliza la CPU para realizar los cálculos, el índice de GFLOPS de potencia obtenido es muy bajo. Además, las multiplicaciones de matrices se realizan mediante la función de multiplicación básica de las librerías de Fortran. Por tanto, cabe decir, que esta aplicación no tiene ningún soporte inicial para GPUs ni para realizar un procesamiento distribuido sobre más de un computador.

Cada simulación calcula la respuesta estructural ante el paso de un tren, a una velocidad determinada, por un puente dado. Por esto, podemos observamos que cada una de estas simulaciones solo depende de los parámetros que ella misma precisa, sin necesidad de interactuar con el resto de simulaciones. Esto significa que cada una de ellas puede realizarse de manera separada e independiente, por lo que nos encontramos ante un contexto fácilmente paralelizable.

3.3 Soporte de comunicación remota con MPI

Para el primer nivel de paralelización de la aplicación original decidimos implementar un sistema distribuido mediante el uso de MPI. Gracias a la implementación MPICH, hemos creado una arquitectura basada en tres tipos de procesos con tareas y códigos diferenciados. Esta arquitectura está formada por los siguientes procesos: *master*, *kernel* y *server*.

La razón por la cual escogemos esta estrategia y no otra, es para conseguir ejecutar un número elevado de simulaciones al mismo tiempo y poder realizar un análisis probabilístico fiable. Como se ha mencionado antes, el número de simulaciones a ejecutar por cada tren puede ser de 6.000, con lo cual, si utilizamos otras estrategias de computación paralela como OpenMP [38] o los POSIX Threads [39], no obtendríamos la capacidad paralela deseada, dado que solo seríamos capaces de ejecutar pocos casos a la vez.

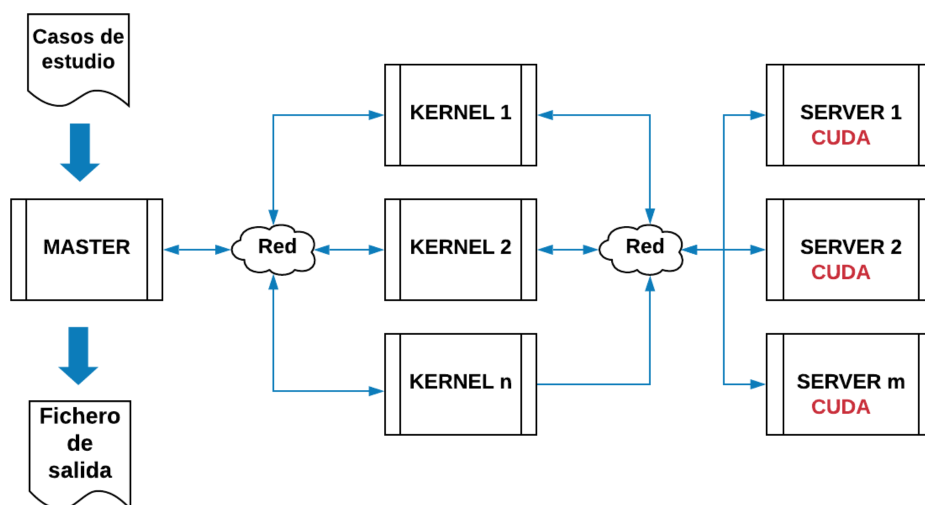

Figura 3.2: Arquitectura de la aplicación final.

Como podemos observar en la figura 3.2 solo existe un proceso *master*, mientras que pueden existir n procesos *kernels*, donde n es el número de procesos *kernels* existentes que debe ser mayor que uno, y m procesos *servers*, donde m es un número de procesos *server* totales también mayor que uno.

Todos los procesos se encuentran comunicados por la red mediante paso de mensajes que ofrece la implementación MPICH. Sin embargo, los datos necesarios para las simulaciones solo se transfieren según el orden que se representado en esta figura.

Además, los tres tipos de procesos pueden ser creados o no en la misma máquina, pudiendo crear así un sistema distribuido. Asimismo, los diferentes *kernels* también pueden crearse en diferentes máquinas según la necesidad. Sin embargo, para la creación de los *servers* es necesario la disponibilidad de una GPU por proceso *server* creado.

3.3.1. *Master*

El proceso *master* es el primer proceso creado y solo puede existir uno de este tipo. La funcionalidad de este proceso es distribuir los casos de prueba entre los *kernels* creados. Para ello, la primera tarea que realiza el *master* es crear el conjunto de *kernels* y *servers* dinámicamente. Para esto, en primer lugar recibe por la línea de comandos el número de procesos que el usuario desea crear por cada tipo de procesos. El *master* será el responsable de crear los procesos en la máquina indicada. Para crear los procesos, se utiliza la función de MPICH *MPI_Comm_spawn_multiple*, explicada en el capítulo 2.3.3. El código implementado para realizar la creación de procesos se encuentra en el apéndice A.1.

Otra función necesaria del proceso *master* es la lectura de los datos de los casos de prueba necesarios para realizar las simulaciones. Esto se realiza mediante la carga de un fichero de entrada con una estructura conocida. De esta manera, el *master* realiza una lectura secuencial de todas las líneas del fichero, almacenando los datos en las regiones de memoria indicadas.

Con la lectura de fichero realizada, el *master* se encarga de enviar la información de cada caso de prueba para la realización de su simulación a cada *kernels* existente, de manera circular. Para esto, recorre un bucle y envía los datos desde el *kernel* con identificador 0 hasta el n-1, siendo n el número de *kernels* totales. Este envío se realiza mediante la comunicación básica basada en *MPI_Send* y *MPI_Recv*.

Cuando los *kernels* terminan las simulaciones envían el resultado al *master*. Este, que se mantiene a la espera de recibir los datos de cada simulación, recolecta los datos recibidos para su posterior escritura en un fichero de salida. Esta recepción se realiza mediante la comunicación básica basada en *MPI_Send* y *MPI_Recv*.

3.3.2. *Kernel*

En segundo lugar, nos encontramos los procesos *kernel*. Pueden existir tantos *kernels* como se desee. Sin embargo, el número ideal de estos procesos es igual a la cantidad de casos de prueba que se desea simular. Este proceso es el encargado de ejecutar el programa escrito en lenguaje Fortran para realizar las simulaciones sobre los casos de prueba, y ejecutará repetidamente la espera de recepción de datos y la ejecución de las simulaciones hasta que se le indique que debe finalizar. En la figura 3.3, se observan los pasos necesarios para realizar una simulación desde la perspectiva de este proceso.

En un primer lugar, se realiza la espera de la recepción de los datos necesarios para simular el paso de un tren por un puente a una determinada velocidad. El envío de estos datos, como se ha mencionado anteriormente, se realiza por parte del proceso *master*.

Seguidamente, con los datos preparados, este proceso lanza la aplicación de Fortran responsable de ejecutar la simulación completa. Esta aplicación ejecuta el cálculo complejo para realizar la simulación. Este cálculo, basado en una discretización por *Elementos Finitos* del puente, contiene numerosas multiplicaciones de matrices, donde la dimensión

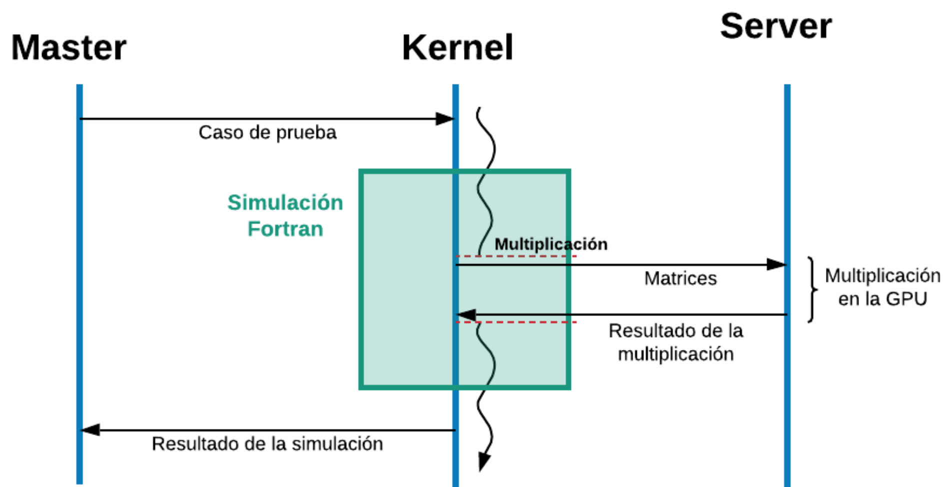

Figura 3.3: Ejecución de una simulación

de las multiplicaciones viene determinada por la complejidad de cada caso de prueba. Por esta razón, existen multiplicaciones de tamaños muy diversos y, por ese motivo, no todas ellas requieren el mismo método de cálculo.

A causa de estas razones, y después de analizar los resultados de un estudio que se detallará en la siguiente sección, cada vez que la simulación de Fortran necesita realizar una multiplicación de matrices se analizan los tamaños de las multiplicaciones a realizar y se decide si la multiplicación debe realizarse en la GPU con CUDA o en la CPU con Fortran.

Cuando se necesita realizar un cálculo en la GPU, el *kernel* es el encargado de realizar el envío de las multiplicaciones al *server* mediante el paso de mensajes de MPI. Para un mayor entendimiento, se ha implementado un protocolo para el control entre en intercambio de mensajes entre los procesos *kernel* y *server*, el cual se puede observar en la tabla 3.1.

VALOR	ETIQUETA	SIGNIFICADO
0	ACK	Acknowledgement
1	PETICION_CALCULO	Petición de cálculo
2	ENVIO_DIMENSIONES	Envío de los valores de las dimensiones de las matrices a multiplicar
3	ENVIO_MATRICES	Preparación para los envíos de las dos matrices a multiplicar
4	ENVIO_MATRIZ_A	Envío de la primera matriz
5	ENVIO_MATRIZ_B	Envío de la segunda matriz
6	ENVIO_RESULTADOS	Envío de los resultados de la multiplicación de matrices
7	CIERRE	Cierre del proceso

Tabla 3.1: Protocolo de envío de mensajes

Gracias a este protocolo, los dos tipos de proceso realizan un intercambio de mensajes mediante la comunicación básica de MPI basada en *send* y *receive*. El orden y la dirección de estos mensajes se pueden observar en la figura 3.4. Cuando el *kernel* obtiene los resul-

tados de la multiplicación realizada en la GPU, vuelve a la simulación de Fortran hasta que esta termina.

Finalmente, y como se puede ver en la figura 3.3, el *kernel* realiza el envío del resultado obtenido de la simulación lanzada al proceso *master* y queda a la espera de un nuevo caso de prueba si es el caso.

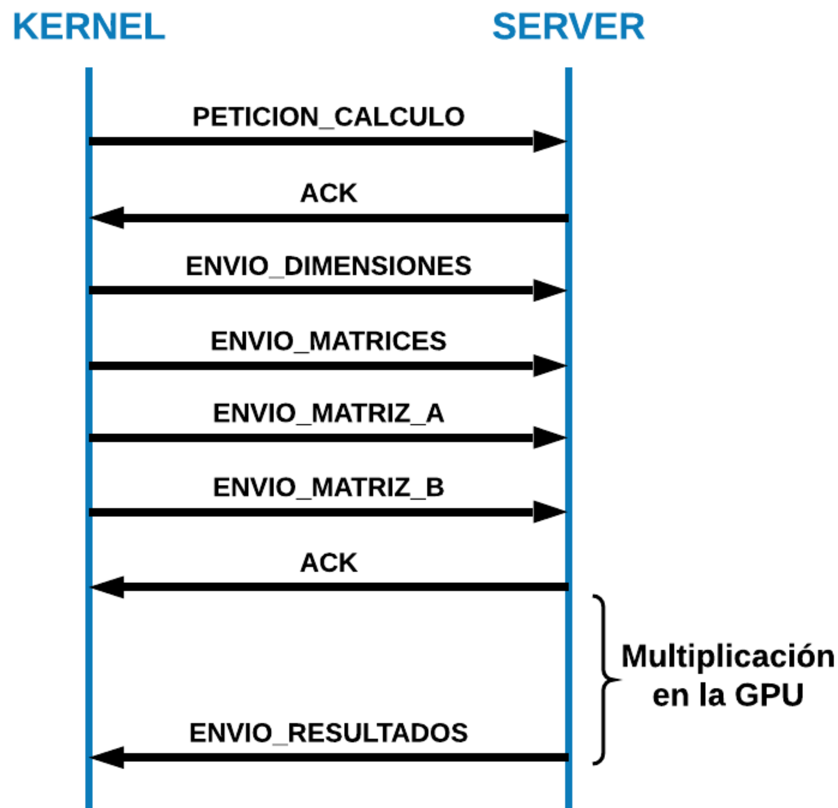

Figura 3.4: Mensajes de envío entre el *kernel* y el *server*

3.3.3. *Server*

Finalmente, el *server* es el encargado de realizar los cálculos en la GPU utilizando la función de multiplicaciones de matrices de cuBLAS. Por esta razón, es importante que los procesos *server* se instancien en máquinas con acceso a una GPU NVIDIA compatible con CUDA.

Este *server*, al igual que el *kernel*, se encuentra en constante espera para recibir los datos a multiplicar. Cuando termina una multiplicación, y después de devolverle el resultado al *kernel*, vuelve a la espera de un mensaje tipo PETICION_CALCULO.

Para poder recibir datos de todos los *kernel*, el *server* realiza la espera de mensajes sin hacer distinción del proceso emisor. Esto es posible al utilizar *MPI_ANY_SOURCE* en la recepción del primer mensaje. Para asegurarnos de que los siguientes mensajes provengan del mismo *kernel* que nos ha lanzado la petición en primer lugar, se guarda el identificador del *kernel* emisor mediante *status.MPI_SOURCE* y, en las siguientes recepciones, el *kernel* espera un mensaje de este kernel. Este funcionamiento queda plasmado en el código 3.1.

```
MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, PETICION_CALCULO, MPI_COMM_WORLD, &status);
int rank = status.MPI_SOURCE;
MPI_Send(NULL, 0, MPI_INT, rank, ACK, MPI_COMM_WORLD);
```

Listing 3.1: Identificación del *kernel* emisor.

3.4 Soporte de GPUs con cuBLAS

Como hemos mencionado antes, para ejecutar las multiplicaciones con cuBLAS se necesita una GPU compatible con CUDA. Además, este proyecto se ha implementado junto con otro alumno, el cual ha realizado la implementación de los cálculos con OpenCL para ejecutar la aplicación sobre GPUs de otros fabricantes. Para aportar flexibilidad a la aplicación se ha diseñado un sistema de soporte para estas dos opciones. De este modo, mediante la función *cudaGetDeviceCount*, obtenemos el número de GPUs compatibles con CUDA. En caso de ser mayor a cero, realizamos las operaciones con CUDA y, en caso contrario, con OpenCL.Esta función se puede observar en el anexo A.2 .

Las multiplicaciones dentro de la GPU se han implementado mediante la función de multiplicación de matrices de tipo de dato *double*, llamada *cublasDgemm()*, la cual corresponde a las funciones BLAS3. Dado que Fortran tiene un método de almacenamiento por columnas y esta función asume este método de almacenamiento, se ha implementado esta función con los parámetros indicados en la documentación.

Cabe aclarar, que para realizar una multiplicación de matrices en la GPU se necesita realizar una copia de los datos de estas matrices a la memoria de la GPU previamente. Además, como la función *cublasDgemm()* guarda el resultado del cálculo en la memoria de la GPU, también se debe realizar una copia de los datos desde la GPU hacia la CPU. Dado que estas transferencias consumen recursos, lo ideal es realizar las estrictamente necesarias. Sin embargo, la aplicación responsable de ejecutar las simulaciones se queda a la espera del resultado de las multiplicaciones, por lo que se necesita realizar esta transferencia lo antes posible.

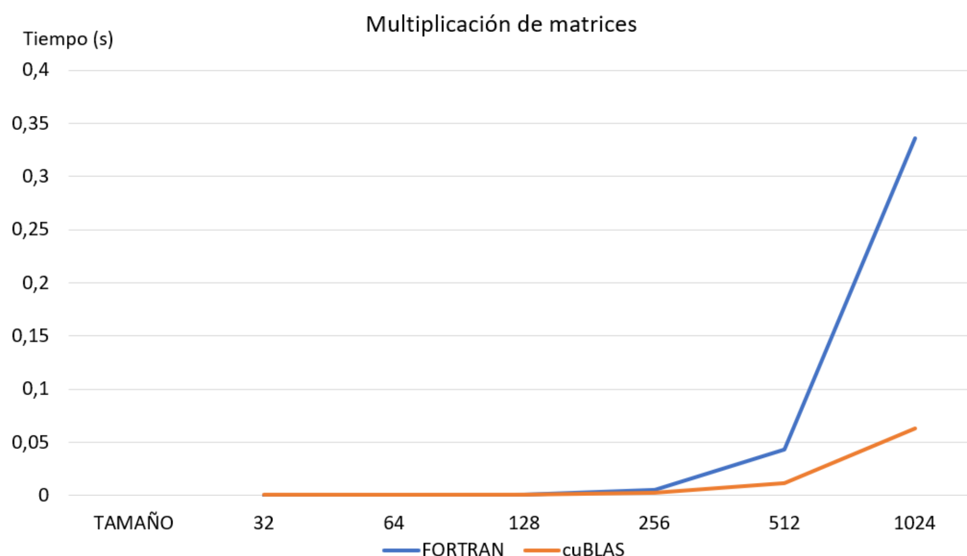

Figura 3.5: Multiplciaciones de matrices con Fortran y cuBLAS

Además, cabe destacar que el tamaño de las matrices influye en el factor de mejora de cuBLAS respecto a Fortran, por lo que las multiplicaciones de matrices con tamaño más reducido no utilizan cuBLAS. Esto puede observarse en la figura 3.5, donde se plasma el estudio de prestaciones sobre multiplicaciones de matrices cuadradas con diferentes

tamaños. Siguiendo este estudio, podemos indicar a la aplicación que las multiplicaciones con matrices mayores de 256 x 256 x 256, pueden ser calculadas en la GPU.

Sin embargo, debemos tener en cuenta otro factor para realizar en envío de las multiplicaciones a la GPU. Dado que, en caso de enviar una matriz esta se envía por red mediante MPI, el tamaño mínimo para obtener una optimización es más elevado. En la tabla 3.6 se plasman los resultados de este segundo estudio, donde se observa que los tamaños necesarios para obtener una optimización con el cálculo en la GPU han aumentado hasta 2048. Este estudio se ha realizado midiendo el tiempo (en segundos) que se tarda en enviar todos los mensajes necesarios, realizar la multiplicación y enviar el resultado. Cabe mencionar que este estudio se realizó entre computadoras conectadas a *Ethernet* por lo que podríamos obtener una aceleración mayor entre computadoras conectadas a *Infiniband*.

TAMAÑO	FORTRAN	CUDA	Aceleración	Opción
32	2,00E-05	2,23E-01	0,00	Fortran
64	1,03E-04	2,13E-01	0,00	Fortran
128	7,26E-04	4,18E-01	0,00	Fortran
256	5,40E-03	2,29E-01	0,02	Fortran
512	4,30E-02	2,76E-01	0,16	Fortran
1024	3,36E-01	8,97E-01	0,37	Fortran
2048	2,88E+00	1,84E+00	1,56	CUDA
4096	5,19E+01	5,76E+00	9,01	CUDA
8192	4,06E+02	2,94E+01	13,81	CUDA

Figura 3.6: Multiplicaciones de matrices incluyendo el tiempo de envío mediante MPI.

Para indicar qué multiplicaciones pueden realizarse en la GPU y qué otras se pueden llevar a cabo con la multiplicación de Fortran, se ha diseñado un módulo externo escrito en Fortran. Este módulo implementa una función con sobrecarga para permitir la multiplicación sobre matrices con diferentes dimensiones. Dentro de estas funciones comprueba si el tamaño de la multiplicación supera el indicado, y en caso afirmativo llama a la función de C del proceso *kernel* para empezar con el envío de datos al *server*.

```
module cuda_matmul
  implicit none
  integer, parameter :: ind1 = 2048, ind2 = 2048, ind3 = 2048
  ...
   interface ! declaracion de la funcion externa en C, para el compilador fortran
   subroutine matmulCUDA(c, a, b, af, ac, bc)
       integer*4 :: af, ac, bc
       real*8   :: a(af,ac), b(ac,bc), c(af,bc)
   end subroutine matmulCUDA
   end interface

   interface matmul_cuda ! polimorfismo en Fortran
     procedure matmul_cuda_m_m, matmul_cuda_m_vc,
  +            matmul_cuda_vf_m, matmul_cuda_vf_vc
   end interface matmul_cuda

   contains  ! rutinas propias de este modulo

   function matmul_cuda_m_m(a,b) result(c) ! variante para multipicar matriz por matriz
     real*8                            :: a(:,:), b(:,:)
     real*8, dimension(size(a,1),size(b,2)) :: c

     if (size(a,1)*size(a,2)*size(b,2).gt.(ind1*ind2*ind3)) then
       call matmulCUDA(c,a,b,size(a,1),size(a,2),size(b,2))
     else
       c = matmul(a,b)
```

```
    end if
  end function matmul_cuda_m_m
  ...
end module cuda_matmul
```

Listing 3.2: Módulo y función de decisión de Fortran.

Debe destacarse que la programación mixta en Fortran y C, necesaria para el empleo de cuBLAS, obliga a definir interfaces explícitas en Fortran cuyo empleo no resulta trivial, y para el que hubo que recurrir al asesoramiento de programadores expertos en la materia. Además, la implementación sugerida inicialmente por estos, basada en el módulo ISO-C-BINDING, no se reveló más eficaz que el módulo en Fortran anteriormente citado en el código 3.2, obtenido tras varias semanas de pruebas y tras consultar bibliografía *ad-hoc* [40]. Dicho módulo solventó definitivamente la conexión entre ambos lenguajes, permitiendo el envío de matrices y vectores de cualquier tamaño para su multiplicación.

El lenguaje Fortran, por otra parte, ha evolucionado desde sus orígenes incorporando diferentes mejoras de modo no siempre homogéneo, con lo cual existen diferentes procedimientos para lograr objetivos similares que dificultan una llegada clara a soluciones óptimas. En particular, los diferentes tipos de matrices, de dimensiones asumidas, diferidas, allocatables o punteros, en ocasiones permiten realizar tareas que parecen equivalentes pero que posteriormente pueden generar inconvenientes en el enlace con C. La propia complejidad del módulo ISO-C-BINDING es buena prueba de los inconvenientes aquí descritos y, aún así, no resultó suficiente hasta que no se creó el módulo completo de Fortran antes indicado. Profundizar en dicho lenguaje para poder unir del modo más eficaz el núcleo de simulación en Fortran con las funciones de cuBLAS ha representado un reto no menor.

3.5 Evaluación

En la figura 3.7 se pasman los resultados obtenidos en la ejecución de 2.000 casos donde los tamaños de las matrices y, por lo tanto, sus respectivos cálculos, son simples. Esto significa que las multiplicaciones de matrices no son lo suficientemente grandes para ser ejecutadas en la GPU, como hemos visto anteriormente, por lo que se han realizado mediante la multiplicación de Fortran. En este caso, se han lanzado los 2.000 casos sobre un conjunto diferente de *kernels*. Nótese que estos *kernels* han sido creados en la misma máquina, la cual posee 4 núcleos. A sabiendas de que los resultados pueden escalarse, en esta figura ya observamos una optimización clara, por lo que podemos decir que la implementación mediante MPI dada ha sido favorable.

Por otra parte, en la figura 3.8 se observa la ejecución de un caso de prueba más extenso al anterior, donde las multiplicaciones de matrices tienen un tamaño aceptable para poder ser enviadas a la GPU. Todos los procesos de esta ejecución han sido creados en la misma máquina, por lo que el tiempo de envío de matrices por la red no ha afectado al cálculo. Cabe mencionar, que la aplicación original realiza más cálculos complejos de gran peso además de las multiplicaciones de matrices, los cuales corresponden a rutinas *dsysv* [41], entre otras funciones. Estas rutinas, al no estar optimizadas, se siguen realizando en la CPU.

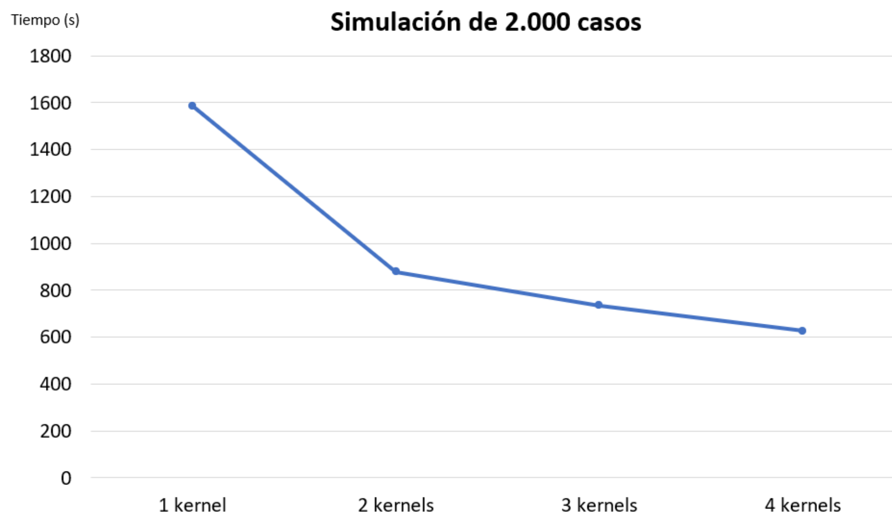

Figura 3.7: Simulaciones con varios *kernels*.

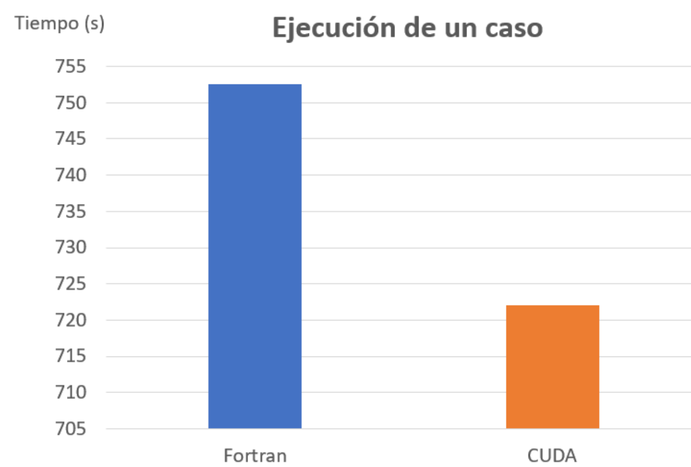

Figura 3.8: Ejecución de una simulación con Fortran y cuBLAS.

CAPÍTULO 4

Aplicación de entrenamiento de redes neuronales

En este capítulo se describe HELENNA, una aplicación de entrenamiento de redes neuronales objetivo de mejora. Originalmente concebido como un simulador de redes neuronales, HELENNA es una aplicación que permite obtener modelos de redes neuronales a partir de una base de datos de entrenamiento y una topología de red neuronal. Esta aplicación se ha desarrollado en el grupo de arquitecturas paralelas de la Universitat Politècnica de València.

4.1 Descripción de la Aplicación

HELENNA (HEterogeneous LEarning Neural Network Application) es una aplicación centrada en el entrenamiento e inferencia de redes neuronales. El ámbito de uso y aplicación es en docencia y en investigación ligado con proyectos Europeos actualmente en ejecución (H2020 RECIPE [42]). HELENNA es un proyecto que se ha iniciado en el Departamento de Informática de Sistemas y Computadores (DISCA) de la UPV por parte de investigadores de dicho departamento, todos ellos en el ámbito del grupo de investigación GAP.

Esta aplicación se caracteriza por su especialización a diferentes arquitecturas de cálculo como CPUs, GPUs y FPGAs, pero tiene como objetivo adaptarse a nuevas arquitecturas más heterogéneas como las TPU de Google [43] o las plataformas embarcadas de NVIDIA como JETSON XAVIER [44] y similares. Estas últimas arquitecturas no están soportadas por el momento. HELENNA tiene como objetivo principal el uso en procesos de inferencia de bajo consumo y con necesidades de tiempo real.

HELENNA permite al usuario seleccionar el dispositivo a utilizar en el proceso de cálculo de la red neuronal. En la tabla 4.1 podemos ver los diferentes dispositivos actualmente soportados, incluido el soporte realizado en este TFG.

HELENNA también contiene soporte para realizar entrenamientos distribuidos de redes neuronales en un *cluster*. Esto es posible mediante el modelo de paralelismo de datos donde los datos de entrenamiento se distribuyen entre los diferentes nodos que componen el *cluster* y estos se sincronizan periódicamente. Para el soporte distribuido se utilizan primitivas de sincronización y transferencia por medio de MPI.

HELENNA se ha desarrollado completamente en el lenguaje de programación C, haciendo uso de una estrategia de llamadas de funciones que permita una programación sencilla y a la vez eficiente. En concreto, en un proceso de entrenamiento o inferencia cada función de cálculo necesaria se ha codificado como un multiplexor el cual selecciona la

Dispositivo	Estado
CPU	Soportado
CPU-AVX	Soportado
CPU-AVX512	En desarrollo
CPU-MKL	Soportado
GPU-clBLAS	Desarrollado en otro TFG
GPU-cuBLAS	Desarrollado en este TFG
FPGA-OpenCL	En desarrollo

Tabla 4.1: Dispositivos actualmente soportados.

Capa	Soporte en MKL	Soporte en GPU/CUDA	Soporte en GPU/OpenCL
Fully Connected (Dense)	Sí	En este proyecto	En otro proyecto
Batch Normalization	Sí	En este proyecto	En otro proyecto
Dropout	Sí	En este proyecto	En otro proyecto
Convolucional	Sí	En este proyecto	En otro proyecto
MaxPooling	Si	En este proyecto	En otro proyecto
AvgPooling	Si	No	No
PerfectShuffle (experimental)	Si	No	No
Permutation (experimental)	Si	No	No
Padding	Si	No	No
Reshape	Si	No	No
SoftMax	Si	En este proyecto	En otro proyecto
Concatenate	Si	En este proyecto	En otro proyecto
Add	Si	En este proyecto	En otro proyecto
Upsampling	Si	No	No
Funciones de activación	Sí	En este proyecto	En otro proyecto
Funciones de coste	Si	En este proyecto	En otro proyecto
Funciones de métricas	Si	En este proyecto	En otro proyecto

Tabla 4.2: Capas soportadas en HELENNA y contribución de este proyecto.

función correspondiente asociada al dispositivo indicado previamente. Esto nos permite que toda la implementación necesaria para soportar un dispositivo se concentre en un único fichero en lenguaje C. Siguiendo la estrategia de diseño de módulos en HELENNA en este TFG se ha implementado todo un módulo de soporte para CUDA en GPUS y se ha enlazado con HELENNA.

Otra cualidad de HELENNA es la generación de estadísticas de tiempos de cálculo para cada una de las operaciones más importantes realizadas durante el proceso de entrenamiento de una red neuronal. Estas estadísticas permiten conocer el impacto de cada una de ellas sobre las diferentes arquitecturas, dispositivos, e implementaciones existentes en esta aplicación.

Asimismo, HELENNA permite una expansión en el soporte de nuevas capas de redes neuronales. Para ello, las diferentes capas se implementan en ficheros en C diferentes y tienen una interfaz definida e uniforme. Las capas actuales que soporta HELENNA se muestran en la tabla 4.2.

Es importante comentar que una nueva capa puede conllevar el soporte a una nueva función aritmética, que por consiguiente requerirá su correcta codificación en los diferentes dispositivos soportados por HELENNA. En este proyecto se ha implementado soporte para dispositivos GPU con CUDA/cuBLAS para las capas actualmente indicadas en la tabla anterior. No obstante, describimos con detalle solamente un subconjunto ellas.

Por último, en todo proceso de entrenamiento se utiliza memoria para crear *buffers* de datos donde se almacenan temporalmente los datos del entrenamiento, los parámetros

de las capas, y otros datos temporales como los gradientes del proceso de entrenamiento. Para cada dispositivo se necesitan las correspondientes funciones de lectura y escritura de *buffers* de memoria, lo cual resulta crucial para las prestaciones de HELENNA dado que debemos minimizar la transferencia de datos en el propio proceso de entrenamiento. En este TFG se indica la estrategia de memoria utilizada así como su implementación.

4.2 Soporte de GPU con cuBLAS

Al comienzo del presente trabajo HELENNA únicamente incluía soporte para CPUs y la librería MKL. En este proyecto se ha desarrollado todo el soporte de los cálculos sobre las GPUs de NVIDIA a través de CUDA y su combinación con la librería cuBLAS. Las funciones específicas con soporte en cuBLAS se han codificado por medio de esta librería. Sin embargo, para las funciones que no presentan soporte de esta librería se han implementado *kernels* en CUDA específicos. De esta manera, se ha dado soporte para las capas anteriormente mencionadas con *kernels* específicos en la mayoría de los casos. Las tablas 4.3, 4.4 y 4.5 muestra todas las funciones implementadas en este proyecto, indicando para cada una de ellas una breve descripción y la estrategia seguida de implementación.

Función	**Descripción**	**Soporte cuBLAS**	**kernel CUDA <fn>_cublas_kernel**
vect_trunc	Para todos los elementos, si el valor es menor a un límite, cambia el valor a cero	No	Sí
vec_add	Realiza la suma de dos vectores	No	SÍ
copy	Copia el vector de entrada en un vector de salida indicando los elementos de separación entre los elementos de los vectores	cublasScopy	No
vec_copy	Copia el vector de entrada en un vector de salida	cublasScopy	No
vec_copy_with _stride_from _cpumem	Realiza la copia de dos vectores indicando los elementos de separación entre los elementos donde el vector de entrada está almacenado en la memoria de la CPU	cublasScopy	No
zero_vec	Guarda en todos los elementos del vector, el valor cero	No	Sí
set_vec	Guarda en todos los elementos del vector, el valor deseado	No	Sí
im2col	Función im2com	No	Sí
col2im	Función col2im	No	Sí
matset_random_ones	Guarda en todos los elementos de la matriz los valores cero y uno de manera aleatoria.	No	No
write_value	Guarda en el elemento indicado el valor deseado	No	Sí

Tabla 4.3: Funciones implementadas.

A continuación se describe con más detalle el soporte dado a las capas convolucional y *dropout* dado que estas presentan un mayor detalle y complejidad y son más representativas del trabajo realizado.

Función	Descripción	Soporte cuBLAS	kernel CUDA <fn>_cublas_kernel
maxpooling	Función de maxpooling	No	Sí
demaxpooling	Función de demaxpooling		
batch_normalization	Función de *batch normalization*	Varias	Varias
batch_normalization _compute_gradients	Función de *batch compute gradients*	Varias	Varias
batch_normalization _backward	Función de *batch normalization backward*	Varias	Varias
vect_step	Para todos los elementos de un vector dado, escribe en un vector los valores cero o uno según si el valor de entrada es negativo o positivo respectivamente	No	Sí
vect_mult	Realiza las multiplicaciones de los elementos de dos vectores	No	Sí
vect_mult_add	Multiplicación de dos vectores	cublasSdot	No
vect_scalar _product	Multiplicación de todos los elementos de un vector por un número escalar	cublasSscal	No
vect_max	Guarda los elementos de mayor valor dados dos vectores	No	Sí
vect_mult_add _with_stride	Multiplicación de dos vectores indicando los elementos de separación de los vectores	cublasSdot	No
vect_V2subV1xK	Realiza, sobre dos vectores, la operación V2= V2 - (k * V1), donde k corresponde a un escalar	cublasSaxpy	No
matmul	Multiplicación de dos matrices	cublasSgemm	No
matmul_bt	Multiplicación de dos matrices donde la segunda matriz está transpuesta	cublasSgemm	No
matmul_at	Multiplicación de dos matrices donde la primera matriz está transpuesta	cublasSgemm	No
matadd_col	Suma cada elemento de cada columna con el elemento del vector de entrada correspondiente	cublasSger	No
matadd_col_alpha	Suma cada elemento de cada columna con el elemento del vector de entrada correspondiente y un valor escalar	cublasSger	No
matadd_row	Suma cada elemento de cada fila con el elemento del vector de entrada correspondiente	cublasSger	No
matrix_transpose	Realiza la transpuesta de una matriz	No	Sí

Tabla 4.4: Funciones implementadas.

Función	Descripción	Soporte cuBLAS	kernel CUDA <fn>_cublas_kernel
matmul_elwise	Multiplicación de dos matrices en forma de vector	No	Sí
vector_to_matrix	Realiza la copia del vector completo en una matriz por filas tantas veces como elementos tenga el vector	cublasSgemm	No
matsub_cublas	Resta cada par de elementos de dos matrices	cublasSaxpy cublasSgeam	No
matadd_cublas	Suma cada par de elementos de dos matrices	cublasSaxpy cublasSgeam	No
mat_A_plus_b_L1	Realiza la operación D = A + beta*sign(B)	No	Sí
mat_copy	Por cada fila de la matriz suma todos los elementos y los almacena en el vector de salida	cublasScopy	No
mat_set	Copia el valor dado para todos los elementos de la matriz.	No	Sí
mat_reduce_rows	Por cada fila de la matriz suma todos los elementos y los almacena en el vector de salida	cublasSgemm	No
vec_axpy	Realiza la operación Y = a * X + Y.	cublasSaxpy	No
matrix_relu	Para todos los elementos de un vector dado, escribe en un vector los valores cero o el valor de entrada según si el valor de entrada es negativo o positivo respectivamente.	No	Sí
matrix_relu_der	Para todos los elementos de un vector dado, escribe en un vector los valores TRUE o FALSE según si el valor de entrada es negativo o positivo respectivamente	No	Sí
sigmoid	Función de *sigmoid*	No	Sí
sigmoid_der	Realiza la derivada de la función *sigmoid*	No	Sí
vect_limit	Normaliza todos los elementos del vector entre un mínimo y un máximo	No	Sí

Tabla 4.5: Funciones implementadas.

4.2.1. Capa convolucional

La función de convolución se utiliza de forma masiva en el procesamiento de imágenes. Al aplicarlo a redes neuronales, la capa convolucional (que implementa el proceso de convolución) tiene la finalidad de detectar patrones relevantes en las imágenes de entrada (en el proceso de entrenamiento) aplicando para ello convoluciones. De hecho, se suelen utilizar diferentes capas convolucionales conectadas de forma secuencial para maximizar y optimizar la detección de patrones y su posterior interpretación en un proceso de segmentación o de clasificación. Mediante la agrupación de un conjunto de estas capas, se puede conseguir identificar estructuras más complejas y abstractas (figura 4.1). Generalmente, estas capas operan sobre un conjunto de imágenes de entrada, donde cada imagen tiene una altura, una anchura y una profundidad representado por el número de canales utilizados (*channels* o *depth*[45]). Así mismo, en un proceso de entrenamiento los datos de entrada se agrupan en *batches* por lo que el proceso de convolución se aplica a un conjunto de imágenes cada vez.

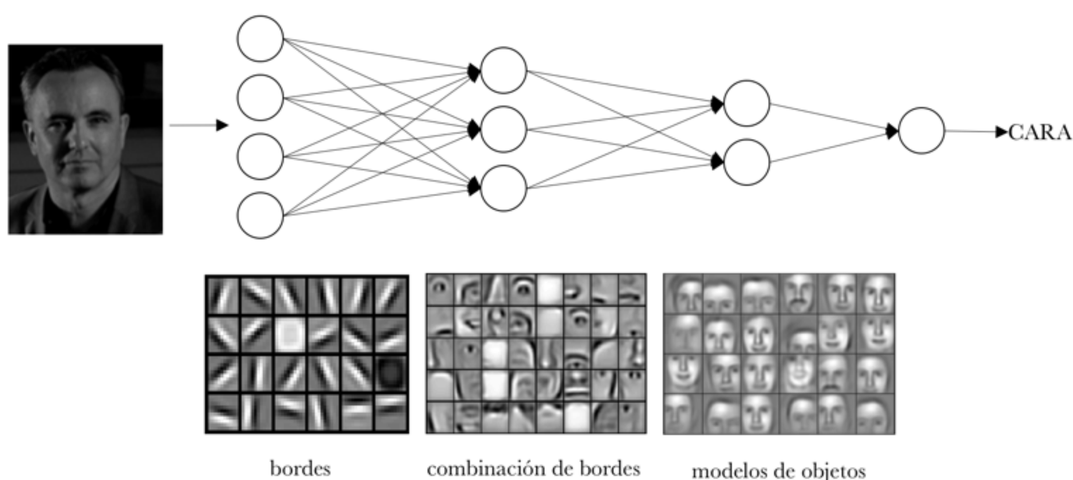

Figura 4.1: Red Neuronal Convolucional [45]

Cuando utilizamos dispositivos como GPUs con un elevado grado de paralelismo, la función de convolución suele transformarse y adaptarse a un proceso de multiplicación de matrices. Esto es debido a que el proceso de multiplicar matrices en GPU está muy optimizado y suele ser la mejor opción. Para ello, en la figura 4.2 se puede observar los pasos básicos necesarios para realizar una convolución por medio de una multiplicación de matrices. Primero, se necesita convertir las imágenes de entrada en una matriz, lo que se conoce como *lowering*. En este trabajo, se ha implementado esta función en HELENNA mediante un *kernel* de CUDA para poder ser ejecutada en la GPU. Esta función es conocida como *im2col* y su implementación se detalla más adelante.

Las imágenes de entrada tienen un formato de cuatro dimensiones ($HI \times WI \times BS \times IC$) al ser un conjunto de BS imágenes, cada una de ellas de $WI \times HI$ pixels de ancho por alto, y cada una de IC canales de entrada (por ejemplo, 3 en imágenes RGB). Por medio de la función *im2col* las imágenes de entrada se convierten a una matriz de tamaño $BS \times WO \times HO$ columnas y $IC \times KW \times KH$ filas. WO y HO es el tamaño de las imágenes de salida del proceso de la convolución, mientras que KW y KH es el tamaño del *kernel* del proceso de convolución que incluye los coeficientes que se entrenarán en el proceso (se denominan también filtros de la convolución). Existe un *kernel* por cada par de canal de entrada y canal de salida del proceso de convolución.

Con la imagen convertida mediante la función *im2col*, se multiplica la matriz resultante con la matriz de filtros (*kernel*). El tamaño del filtro es de $KW \times KH \times IC$ columnas

y de OC filas, donde OC es el número de canales de salida. Al multiplicar el resultado de *im2col* con el *kernel* obtenemos una matriz de salida en formato $BS \times WO \times HO$ columnas y OC filas. Por tanto, corresponde con la salida esperada de BS imágenes de tamaño $WO \times HO$ y cada una de ellas con OC canales. Esta multiplicación se lleva a cabo mediante la función de multiplicación de matrices *cublas<t>gemm()* de cuBLAS.

Adicionalmente, se suma el sesgo (*bias*) que es un coeficiente adicional por cada canal de salida. Todo este proceso forma parte del procesado *forward* que se realiza en la capa. Ahora bien, en el entrenamiento de redes neuronales también debemos implementar el proceso de *backward*, donde se actualizan los filtros y sesgos y se propaga el error en sentido contrario a la topología de la red neuronal. Para ello, debemos implementar una función inversa *col2im* que nos permitirá transformar el error de entrada de la capa convolucional al error de salida que se propagará hacia atrás en la topología de la red neuronal.

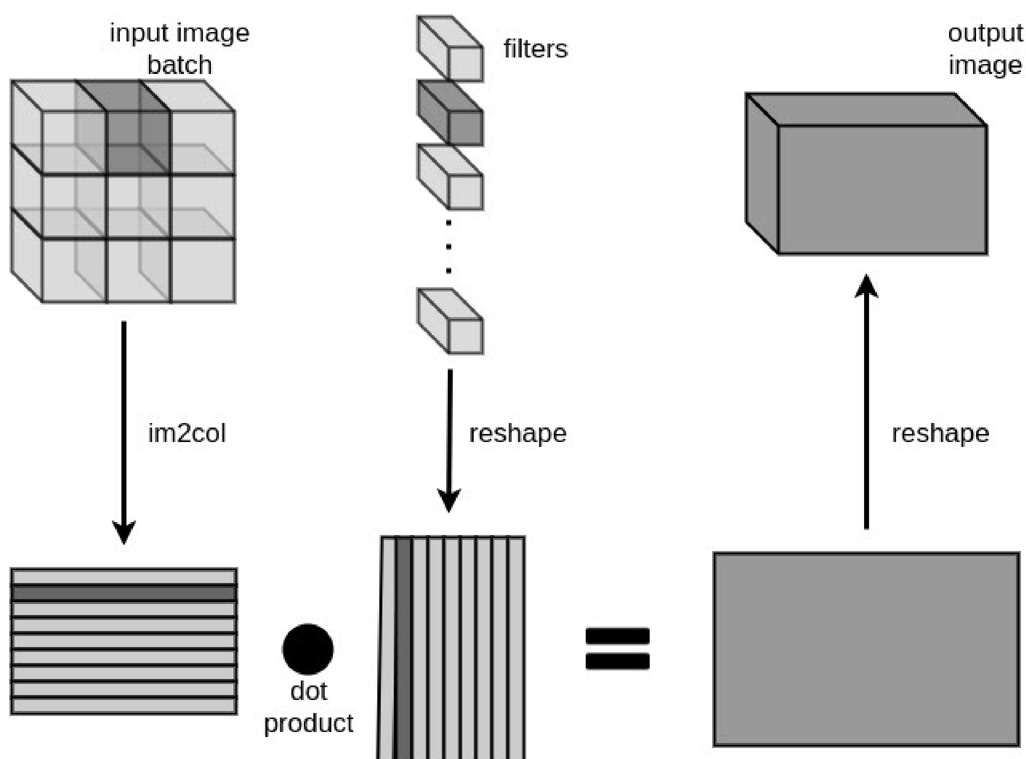

Figura 4.2: Convolución [46]

Pasamos a describir las funciones *im2col* y *col2im*.

Función im2col

Como hemos mencionado anteriormente, el objetivo de esta función es transformar las imágenes de entrada en una matriz adaptada para su posterior multiplicación por la matriz de filtros (figura 4.3). Los cálculos de esta función están compuestos, principalmente, por estas variables:

- BS: tamaño del *batch* (número de imágenes o muestras).
- IC: número de canales de entrada.
- HI: alto de los canales de entrada.
- WI: ancho del canal de entrada.

- HO: alto de los canales de salida.
- WO: ancho del canal de salida.
- KH: alto del *kernel*.
- KW: ancho del *kernel*.
- SH: número de elementos en los que se desplaza la ventana *kernel* verticalmente (*stride* vertical).
- SW: número de elementos en los que se desplaza la ventana *kernel* horizontalmente (*stride* horizontal).
- PH: *padding* horizontal aplicado a la imagen de entrada.
- PW: *padding* vertical aplicado a la imagen de entrada.

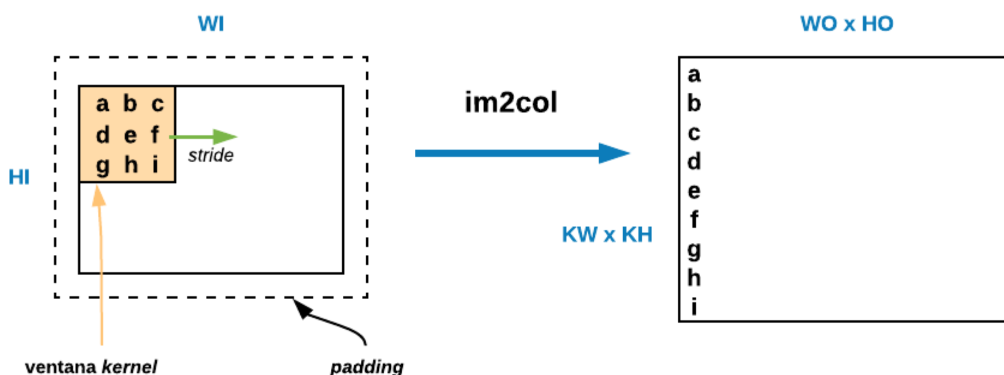

Figura 4.3: Proceso im2col por una sola imagen con un solo canal.

Las dimensiones de la imagen origen es $BS \times IC \times HI \times WI$, mientras que las dimensiones de la matriz resultante son $BS \times CI \times HO \times WO \times KW \times KH$. Si abstraemos el problema a un solo canal de una sola imagen podemos ver en la figura 4.3 el formato de las imágenes de entrada y el resultado a nivel de un solo canal. En concreto, la ventana del *kernel* de tamaño $KW \times KH$ se desplaza por encima de la imagen de entrada y sus componentes de la imagen (píxeles a a i) se disponen en la matriz resultado en una columna. Todos estos píxeles se utilizarán para multiplicarse por el kernel y producirán un solo píxel de salida. La ventana se desplaza horizontalmente y verticalmente recorriendo toda la imagen en función del *stride* indicado en la operación, rellenando columnas sucesivas en la matriz de salida de la operación. Adicionalmente se puede definir un *padding* de píxeles a 0 en la imagen de entrada.

En HELENNA, la operación de *im2col* debe poder lanzarse por columnas. Esto quiere decir que una operación *im2col* debe poder ser ejecutada solamente para un fragmento de la matriz de salida. El motivo de esta funcionalidad radica en el excesivo tamaño que puede tener una matriz completa *im2col*.

El algoritmo implementado en CUDA se puede ver en el código 4.1. Para implementar el algoritmo en CUDA, tomamos la aproximación de maximizar el paralelismo de la GPU, y por tanto, lanzamos tantos hilos como elementos de la matriz resultante ($IC \times WO \times HO \times KW \times KH \times OC$), evitando así el uso de bucles. Ahora bien, cada hilo se encarga del mismo píxel en todas las imágenes de entrada (BS). Cada hilo calcula la

posición correspondiente en las dos matrices (matriz de entrada y matriz resultante) respecto al identificador de cada hilo. En caso de que la posición calculada corresponda a un elemento de la matriz de entrada, se accede a este elemento y se copia en la matriz resultante en la posición indicada. En caso contrario, se guarda el valor cero en el elemento de la matriz destino, lo cual se conoce como *padding*.

```
__global__ void fn_im2col_cublas_device(int I, int WI, int HI, int B, int KW, int KH, int
    WO, int HO, int PW, int PH, int SW, int SH, float *in_ptr, float *out_ptr, int
    first_row, int last_row, int first_col, int last_col)
{
    //identificador del hilo
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    //solo ejecutan la funcion los hilos indicados
    if (tid < I * KH * KW * HO * WO * B)
    {
        //identificacion de las variables
        int i  = tid / B / WO / HO / KW / KH;
        int kh = tid / B / WO / HO / KW % KH;
        int kw = tid / B / WO / HO % KW;
        int h  = tid / B / WO % HO;
        int w  = tid / B % WO;
        int b  = tid % B;
        int hi = h * SH - PH + kh;
        int wi = w * SW - PW + kw;

        //soporte para la ejecucion fragmentada
       int num_cols = last_col - first_col + 1;
       int row_size = num_cols;
       int row = kw + (kh * KW) + (i * KW * KH);
       int col = b + (w * B) + (h * WO * B);

       if ((row >= first_row) && (row <= last_row)
           && (col >= first_col) && (col <= last_col))
      {
           //calculo de las posiciones de los elementos
           size_t out_addr = ((row - first_row) * row_size) + (col - first_col);
            size_t in_addr1 = (size_t)i * (size_t)B * (size_t)WI * (size_t)HI;
            size_t in_addr2 = in_addr1 + (size_t)hi * (size_t)B * (size_t)WI;
            int in_addr = (in_addr2 + (wi * B) )+ b;

            int force_padding = (wi < 0) || (wi >= WI) || (hi < 0) || (hi >= HI);
            //si el elemento NO corresponde a la matriz de entrada se realiza padding
            if (force_padding)
            {
                out_ptr[out_addr] = 0;
            }
            else
            {
                //si el elemento corresponde a la matriz de entrada
                //se copia el elemento en la matriz de salida
                out_ptr[out_addr ] = in_ptr[in_addr ];
            }
       }
    }
}
```

Listing 4.1: Código de la función im2col

Es importante recalcar la importancia de los valores de stride y *kernel* a la hora de calcular esta posición referente. La matriz resultante, se puede ver como una composición de submatrices de tamaño $KW \times KH$. Los valores de *stride* (SW y SH) indican el número de elementos en los que se mueve la ventana *kernel*. Todo esto está plasmado en la figura

4.4, donde observamos como los diferentes *kernels*, de tamaño 3×3 y representados con diferentes colores, provienen de la matriz de entrada con un determinado *stride* 2×2.

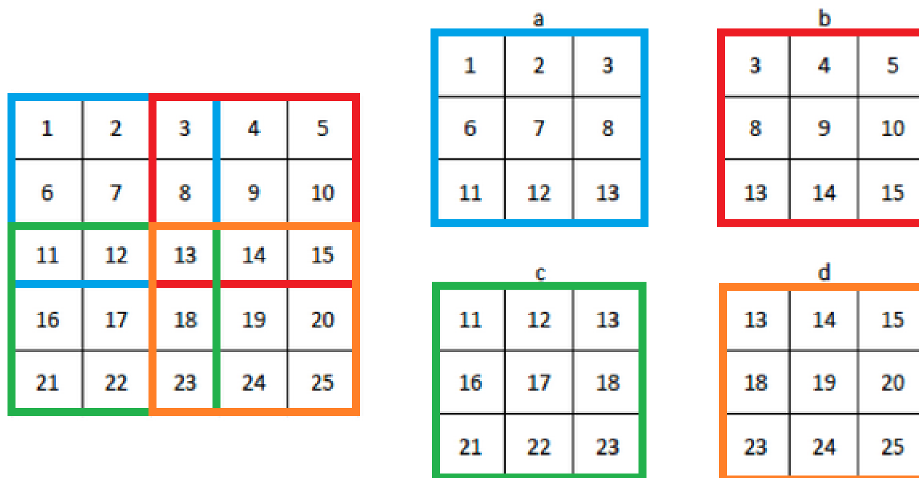

Figura 4.4: Funcionamiento del *stride*.

En la figura 4.5 se representa la mejora obtenida con la implementación de la función *im2col* mediante CUDA. Nótese que las redes neuronales utilizadas en este proyecto están descritas en el apéndice B. Se puede observar una clara mejora en todas las topologías ejecutadas, donde el factor de mejora en todas ellas es de más de cuatro. Hay que hacer constar que en este algoritmo no existe cálculo de coma flotante. Es más, la complejidad en el cálculo es mínima ya que solamente estamos copiando posiciones de memoria de la matriz de entrada en posiciones de memoria de la matriz de salida. El hecho de poder lanzar múltiples hilos en la GPU nos garantiza unas prestaciones muy superiores a el uso de MKL con CPUs.

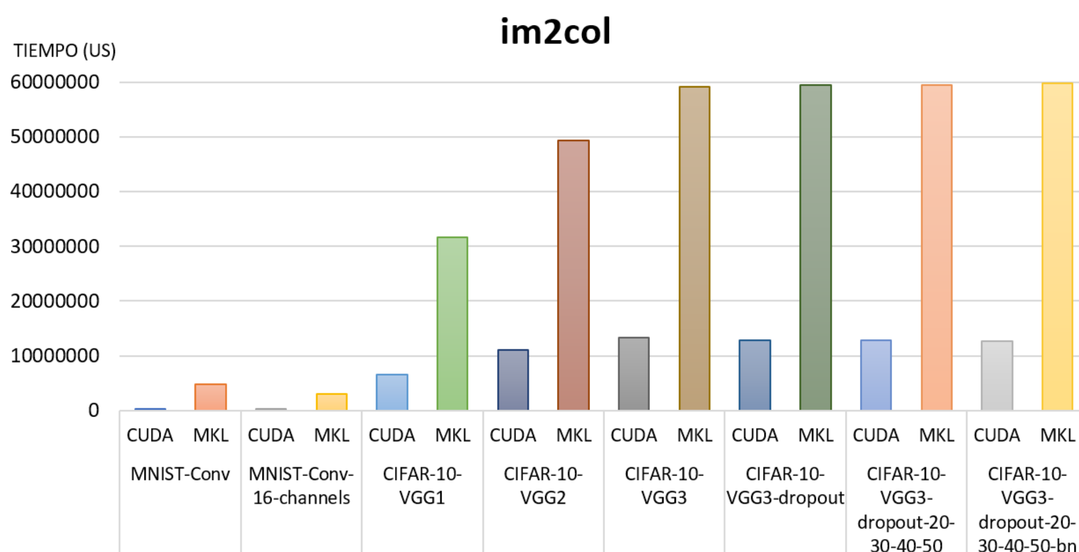

Figura 4.5: Resultados acumulados de la ejecución de *im2col* en las diferentes topologías de entrenamiento.

Función *col2im*

En el proceso de aprendizaje, los *kernels* (coeficientes) de la capa convolucional deben modificarse y adaptarse al error obtenido. El error viaja a través de la red neuronal en sentido inverso. Por tanto, en el caso de nuestra capa convolucional, el error tiene el formato de una matriz de entrada de OC filas y $BS \times WO \times HO$ columnas. En el proceso de propagación se multiplica el error de entrada por el *kernel* que recordemos tiene una geometría de OC filas y $KW \times KH \times IC$ columnas. Al multiplicar el kernel (en formato traspuesto) con el error de entrada obtenemos una matriz resultado de $KW \times KH \times IC$ filas y $B \times WO \times HO$ columnas. Si comparamos con la función anterior (*im2col*) veremos que tiene el mismo formato. Es decir, ahora necesitamos realizar la función inversa (*col2im*) para obtener el error a la salida (en el sentido contrario de la topología) que se propaga hacia atrás. Por tanto, la función *col2im* tiene la propiedad de transformar la matriz de entrada en una o diversas imágenes (figura 4.6). Los cálculos de esta función están compuestos, principalmente, por las mismas variables existentes en la función anterior, incluido el *padding* y el *stride*.

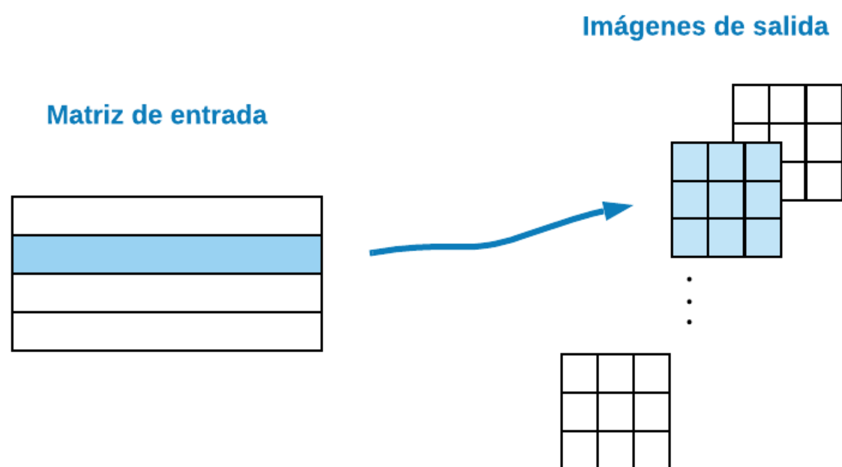

Figura 4.6: Funcionamiento col2im.

Al igual que en la función anterior, se lanzan tantos hilos como elementos de la matriz de entrada existen. En este caso, cada hilo también calcula la posición correspondiente en las dos matrices respecto al identificador de cada hilo. Sin embargo, a diferencia del caso anterior, solo se suma el elemento desde la matriz de entrada a la imagen destino en caso de que el elemento no se encuentre fuera de rango de la imagen destino. En caso contrario, simplemente no se realiza ningún paso. La implementación de este algoritmo se detalla en el código 4.2. Cabe mencionar que esta implementación, actualmente, está soportada para tamaños de SW y SH iguales a KW y KH. En caso contrario, se pueden producir condiciones de carrera, ya que mismos hilos estarían modificando las mismas regiones de memoria sin control ninguno. En la sección 6.2.2 se contempla este aspecto desfavorable y se indican su posible futura mejora.

```
__global__ void fn_col2im_cublas_device(int I, int WI, int HI, int B, int KW, int KH, int
    WO, int HO, int PW, int PH, int SW, int SH, float *in_ptr, float *out_ptr, int
    first_col, int last_col) {

   //identificador del hilo
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   //solo ejecutan la funcion los hilos indicados
```

```
    if (tid < I * KH * KW * HO * WO * B)
    {
        //identificacion de las variables
        int i = tid / WO / B / HO / KW / KH;
        int kh = tid / WO / B / HO / KW % KH;
        int kw = tid / WO / B / HO % KW;
        int h = tid / WO / B % HO;
        int w = tid / B % WO;
        int b = tid % B;
        int hi = h * SH - PH + kh;

        //soporte para la ejecucion fragmentada
        int num_cols = last_col - first_col + 1;
        int row_size = num_cols;
        int row = kw + (kh * KW) + (i * KW * KH);
        int col = b + (w * B) + (h * WO * B);

        if ((col >= first_col) && (col <= last_col)) {
            //calculo de las posiciones de los elementos
           size_t out_addr = (row * row_size) + col;
            size_t in_addr1 = (size_t)i * (size_t)B * (size_t)WI * (size_t)HI;
            size_t in_addr2 = in_addr1 + (size_t)hi * (size_t)B * (size_t)WI;
            int wi = w * SW - PW + kw;
            int force_padding = (wi < 0) || (wi >= WI) || (hi < 0) || (hi >= HI);
            if (!force_padding) {
                //si el elemento corresponde a la matriz de entrada
                //se suma el elemento en la imagen de salida
                int in_addr = in_addr2 + (wi * B) + b;
              in_ptr[in_addr] += out_ptr[out_addr];
            }
       }
    }
}
```

Listing 4.2: Código de la función im2col

En la figura 4.7 se plasman los resultados de la ejecución de esta función en las diferentes topologías. En este caso, la aceleración mínima conseguida en todas las topologías es de más de 11. Podemos observar que esta aceleración es más alta que la que se daba en la función anterior. Esto ocurre porque la cantidad de veces que se realizan accesos a memoria es considerablemente menor que en el algoritmo anterior.

4.2.2. Capa *dropout*

En las Redes Neuronales el sobreajuste (o *overfitting*) es un problema frecuente. Esto ocurre cuando se sobreentrena la red neuronal y considera como válidos los defectos de los datos (lo que se conoce como ruido) [45]. Este efecto queda visualmente representado en la figura 4.8, donde podemos observar tres gráficas separando, mediante una línea distinta, las dos posibles clases: la primera clase corresponde a los puntos y la segunda a las cruces.

En la primera gráfica queda reflejado el sobreajuste. Podemos observar como, al definir la línea entre las dos clases, los elementos que sobresalen de su clase tienen el mismo impacto que los elementos correctos. Esta situación provoca que la línea no sea óptima y presenta un efecto negativo, por lo que es importante evitar este sobreajuste. Para reducir este sobreajuste se pueden utilizar las capas *dropout*.

La capa *dropout* desactiva un conjunto aleatorio de neuronas, tanto ocultas como visibles, junto con todas sus conexiones (figura 4.9). Este método reduce el sobreajuste ya que, al desactivar algunas neuronas, se evita que las neuronas activadas dependan exce-

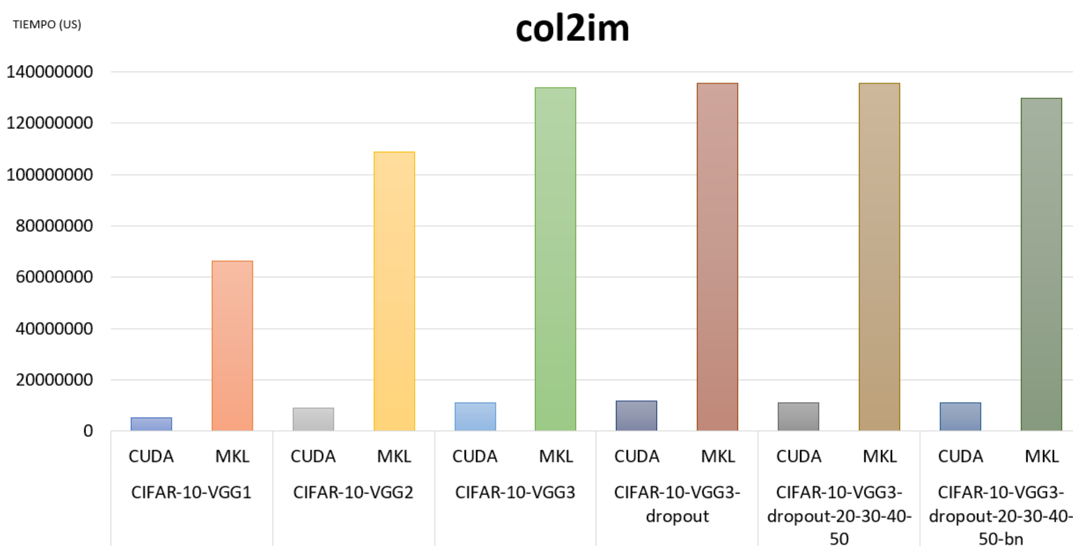

Figura 4.7: Resultados acumulados de la ejecución de *col2im* en las diferentes topologías en el proceso de entrenamiento.

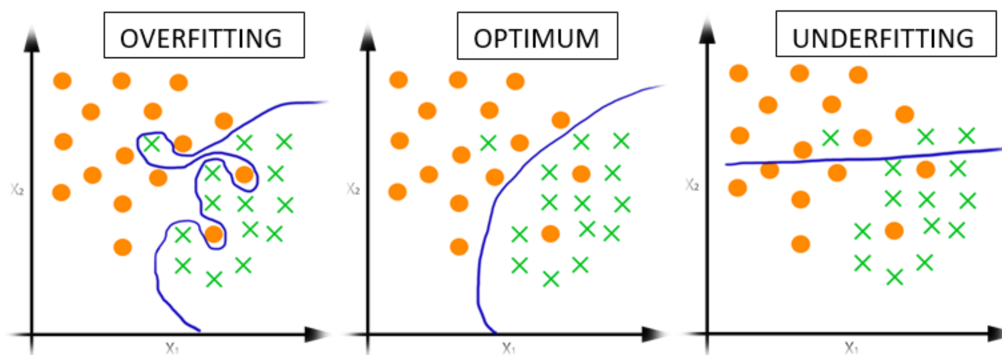

Figura 4.8: Ejemplo de *overfitting* [47]

sivamente de la salida de las neuronas vecinas, y la fuerza a trabajar, en mayor medida, solitariamente [48]. Dado que el *dropout* desactiva un porcentaje de neuronas, es necesario llevar un control para evitar el *underfitting*. Además, el valor de la tasa de aprendizaje (*learning rate*) puede ser mayor que en una red neuronal sin *dropout*.

En cada iteración, la capa *dropout* debe desactivar un conjunto aleatorio y diferente de neuronas. Para decidir qué neuronas se desactivan se debe crear una máscara de ceros y unos, donde el cero indica que la neurona se desactiva y el uno que la neurona sigue activada. La implementación de la función generadora de la máscara se explica más adelante. Una capa de *dropout* simplemente realiza una multiplicación de la matriz de entrada por la máscara generada, por tanto, filtrando una serie de neuronas. En la fase de aprendizaje la capa *dropout* debe propagar el error solo a través de aquellas neuronas que habían sido activadas en el proceso de *forward*. Por este motivo, la máscara se vuelve a utilizar en la fase de *back propagation*.

Creación de la máscara

Para crear la máscara, en primer lugar, se ha implementado una función que genera números aleatorios. Para poder ejecutar esta función en la GPU, se ha implementado una función de la librería cuRAND (*CUDA random number generation* library), la cual genera

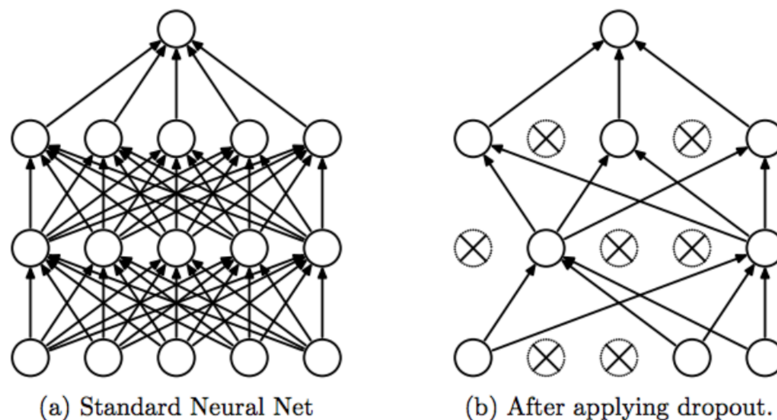

Figura 4.9: Método *dropout*. [49]

números aleatorios en un vector. Esta función es conocida como *curandGenerateUniform* y genera números aleatorios entre cero (incluido) y uno (no incluido).

Sin embargo, el vector resultante no es suficiente para decidir la aleatoriedad con la que las neuronas se desactivan. Se necesita llevar un control de la tasa de neuronas desactivadas. Para esto, se ha creado una segunda función mediante un *kernel* de CUDA, la cual nos permite indicar el porcentaje de neuronas que necesitamos desactivar. Esto es posible mediante el uso de un argumento que nos indique la probabilidad de que una neurona sea desactivada. Esta función filtra los números aleatorios generados en la función anterior para que solo se desactiven aquellas que han recibido un valor menor a la probabilidad. En caso de que el valor sea menor desactivan la neurona. La estructura de esta función es la siguiente:

```
__global__ void fn_matset_random_ones_cublas_device( float* ptr_m, int rows, int cols,
    float prob) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < rows * cols) {
        ptr_m[idx] = ptr_m[idx] <= prob;
    }
}
```

Listing 4.3: Desactivación de neuronas

En la figura 4.10 podemos ver los tiempos de ejecución de la función implemente en la ejecución topología VGG3-dropout-20-30-40 sobre la clasificación CIFAR-10, la cual contiene cuatro capas *dropout*. Podemos comprobar que la función de CUDA es óptima, ya que presenta una aceleración de 120 veces respecto MKL.

4.2.3. Soporte de creación, lectura y escritura de *buffers*

Como hemos mencionado anteriormente, para los cálculos en la GPU mediante CUDA se necesita que los datos estén almacenados en la memoria de la GPU, lo cual difiere de la necesidad de MKL ya que esta última necesita que los datos estén almacenados en la CPU. Para proporcionar un soporte para todos los dispositivos existe un array de *buffers*, el cual contiene todos los punteros de la GPU donde se almacenan los datos necesarios para realizar el entrenamiento. Esta estructura de *buffers* no ha sido desarrollada en este proyecto. Sin embargo, en este proyecto se han implementado las funciones necesarias para llevar un control sobre la memoria de la GPU mediante CUDA las cuales se pueden observar en la tabla 4.6.

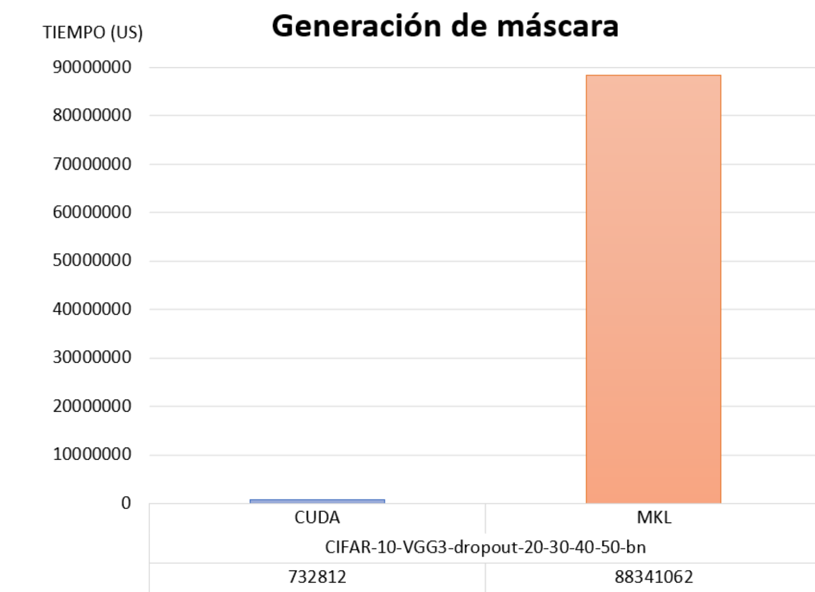

Figura 4.10: Ejecución de la creación de la máscara

Función	Descripción
fn_allocate_buffer_cublas	Realiza la asignación dinámica de memoria en la GPU
fn_deallocate_buffer_cublas	Libera la región de memoria de la GPU asociada al puntero
fn_read_buffer_cublas	Copia desde los datos desde la GPU hacia la CPU
fn_value_cublas	Copia un elemento de la matriz indicada desde la GPU hacia la CPU
fn_write_buffer_cublas	Copia los datos desde la CPU hacia la GPU

Tabla 4.6: Funciones implementadas relacionadas con el soporte de *buffers*.

4.2.4. Sincronización

En este proyecto se ha implementado una función para realizar la sincronización explícita entre la GPU y la CPU. Esta sincronización puede ser invocada desde la línea de parámetros añadiendo el argumento *–force_sync* y se ejecutará durante toda la simulación. De esta manera, todas las funciones implementadas llaman a la función encargada de realizar la sincronización independientemente de si se ha indicado la sincronización explícita o no. Esta última función comprueba si se ha invocado la sincronización explicita en la línea de comandos y, en caso de afirmativo, ejecuta la función *cudaDeviceSynchronize*, responsable de realizar la sincronización, lo cual se puede observar en el siguiente código:

```
void fn_synchronize(){
    if (force_sync)
    {
          cudaDeviceSynchronize();
    }
}
```

Esta sincronización forzada es útil para el estudio de las prestaciones de las funciones de CUDA y cuBLAS ya que, como hemos mencionado anteriormente, las funciones

son asíncronas y, en caso de no realizar una sincronización explícita, los resultados no representarían la ejecución real de función que se desea estudiar.

4.3 Evaluación

El objetivo principal de esta aplicación es ejecutar una topología de Red Neuronal para poder realizar un entrenamiento y, posteriormente, clasificar un conjunto de muestras. Para hacer esto posible, la aplicación realiza predicciones y, según el acierto, va modificando los parámetros necesarios automáticamente. Por lo tanto, podríamos decir que para saber si nuestra aplicación realiza las predicciones correctamente debemos visualizar el porcentaje de acierto. Cabe decir, que en este porcentaje entran numerosas variables como tipo de clasificación, el número de épocas (*epochs*) a ejecutar o la tasa de aprendizaje (*learning rate*), entre otras.

En la figura 4.11 se refleja el porcentaje de acierto de las clasificaciones MNIST y CIFAR-10 según la ejecución de las diferentes topologías con el uso de CUDA y MKL. Todas las ejecuciones se han lanzado con los mismos parámetros. En esta figura, podemos observar como el acierto de las ejecuciones de MNIST es mayor al acierto de CIFAR-10, incluso con las mismas épocas y tasa de aprendizaje. Esto es debido a que las imágenes de MNIST son más fáciles de clasificar que las imágenes de CIFAR-10, dado que MNIST clasifica número escritos a mano y CIFAR-10 imágenes sobre objetos tridimensionales, los cuales pueden ser vistos desde diferentes perspectivas.

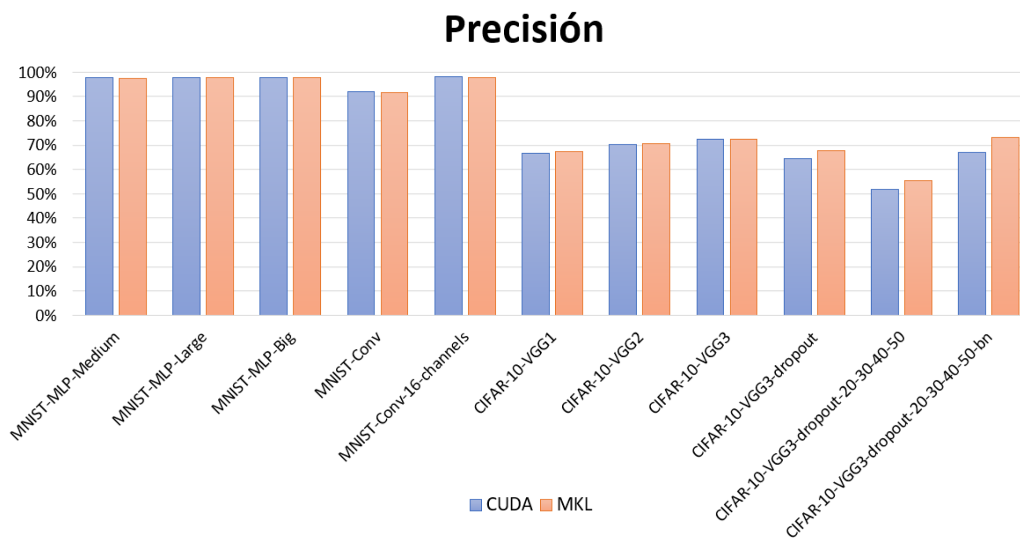

Figura 4.11: Precisión de la ejecución de las topologías con CUDA y MKL

Además, en la figura anterior, podemos observar como la precisión de CUDA y MKL es prácticamente la misma, pero con un tiempo de cálculo menor en el caso de CUDA, como veremos a continuación. Esto significa que obtenemos una optimización sin perder en precisión.

En cuanto a los resultados sobre la contribución aportada a la aplicación de HELENNA, podemos ver que dan lugar a una clara optimización. Para ver más detalladamente la paralelización conseguida a continuación se comparan las ejecuciones realizadas de diferentes topologías con diferentes clasificaciones. De esta manera, comparamos los resultados obtenidos con la implementación de CUDA y CUBLAS con el uso de la librería MKL para los cálculos. En todos los escenarios, se ha decidido utilizar los mismos pará-

metros de ejecución, por lo que las operaciones a realizar y el tamaño del problema son idénticos en los dos casos.

Primeramente, se compara la clasificación de la base de datos MNIST con diferentes topologías. Esta base de datos está formada por un conjunto de imágenes que corresponden a números escritos a mano. El objetivo de este entrenamiento es clasificar las imágenes en una de las diez clases existentes. El tamaño de esta clasificación es, en comparación a la clasificación CIFAR-10, pequeño, dado que cada imagen está formada por 28 x 28 píxeles y, dado que las imágenes se encuentran en blanco y negro, solo hay un canal. Sin embargo, la cantidad de cálculos a realizar también viene determinado por la topología y el número de épocas, entre otros factores.

Los resultados de las ejecuciones de las diferentes topologías de MNIST están plasmados en la figura 4.12. Como se esperaba, se puede observar que en todas las ejecuciones cuBLAS tiene un mejor rendimiento. También era de esperar que en las topologías más simples, la diferencia de prestaciones sea menor que en aquellas que implican un cálculo computacional mayor. Esto queda reflejado en la figura 4.16, donde se calcula la aceleración de cuBLAS respecto MKL. Además, podemos observar como la implementación de cuBLAS soporta mejor el aumento de la magnitud del problema ya que, en MKL aumenta hasta un factor de 31, mientras que en el caso de cuBLAS solo aumenta en un factor de 14.

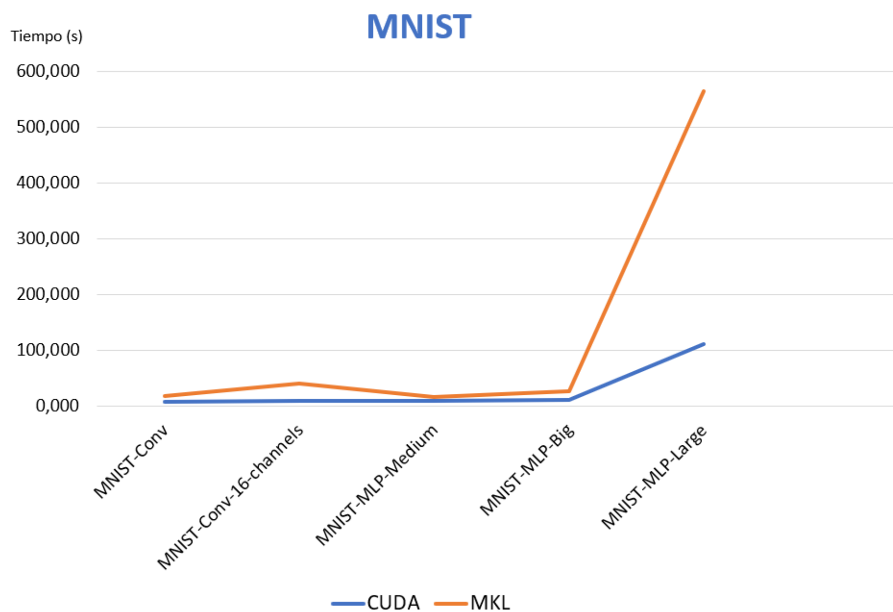

Figura 4.12: Comparación de la ejecución de la clasificación de MNIST.

Si indagamos más exhaustivamente dentro de los resultados de la ejecución de esta clasificación con la topología MLP-Large, podemos comparar el coste que representan las ejecuciones de las funciones que componen esta topología. Estos costes están representados en la figura 4.13, donde, como cabe esperar, podemos observar que una de las funciones más costosa en MKL es la multiplicación de matrices, la cual está representada como matmul en la figura. Podemos observar como, gracias a la optimización de esta función, se reduce drásticamente el tiempo de cómputo. Esto se debe a que esta función ha sido implementada mediante la función de multiplicación de cuBLAS.

Sin embargo, si comparamos las ejecuciones de las funciones de CUDA, observamos que en este caso, la función más costosa es la multiplicación de matrices. En cambio, las funciones que consumen más tiempo de cómputo son las funciones encargadas de

realizar *batch_normalization*. Esto es debido a que, en estas funciones, ejecuta diferentes *kernels* de CUDA implementados en este proyecto.

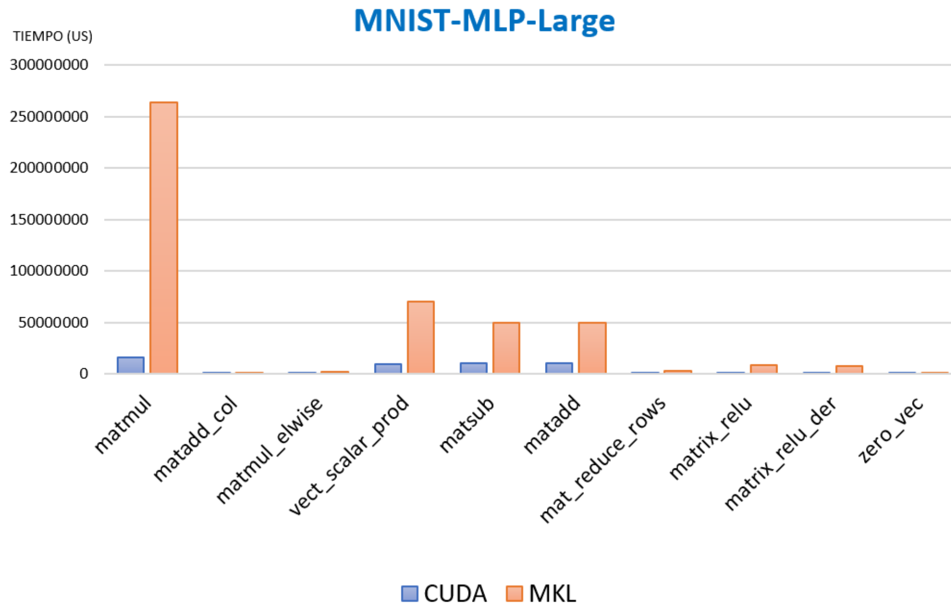

Figura 4.13: Comparación del coste de la ejecución de las funciones.

En un segundo lugar, también se ha realizado un estudio de los resultados de clasificación de la base de datos CIFAR-10. Esta base de datos está formada por un conjunto de imágenes en las que se observan diferentes tipos de animales y métodos de transporte. El objetivo de este entrenamiento, al igual que en la clasificación anterior, es clasificar las imágenes en una de las diez clases existentes. Sin embargo, esta base de datos aumenta el tamaño del problema, dado que cada imagen está formada por 32×32 píxeles, lo cual incrementa el número de entradas. Además, al tratarse de imágenes en color, el tamaño de canales es de tres (canales RGB).

En las ejecuciones de la clasificación de CIFAR-10 (figura 4.14), se puede observar más claramente la mejora que se ha obtenido con cuBLAS, llegando a ser hasta un factor de más de 12 en comparación a MKL (figura 4.16).

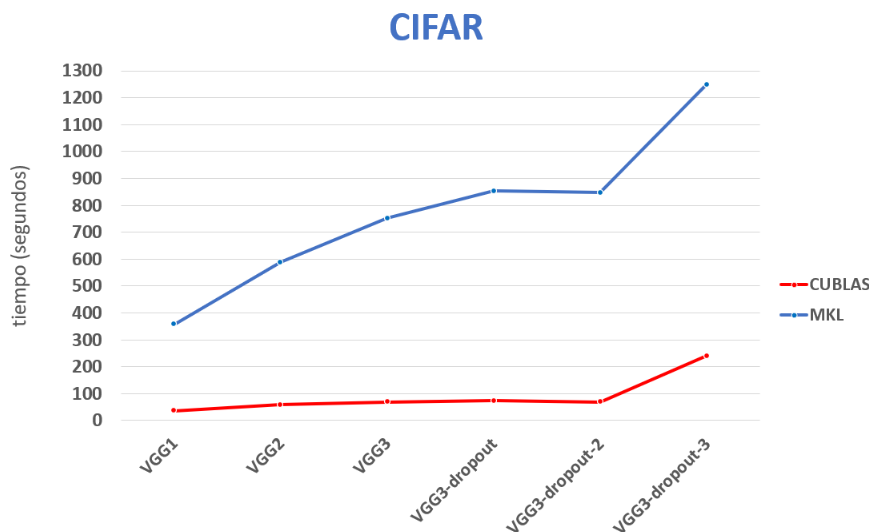

Figura 4.14: Comparación de la ejecución de la clasificación CIFAR.

Una vez más, podemos indagar más exhaustivamente dentro de los resultados de la ejecución de esta clasificación con la topología más costosa para comparar el coste que representan las ejecuciones de las funciones que componen esta topología (figura 4.15). Volvemos a observar como la función más costosa en MKL es la multiplicación de matrices, la cual está representada como matmul en la figura, lo cual era lo esperado.

Otro aspecto a destacar es la diferencia de mejoras entre la función de multiplicación y la función *back normalization backward*, la cual está representada en la figura como b_n_backward. Esta última presenta una aceleración significativamente menor si la comparamos con la aceleración que presenta la función de multiplicación. Esto viene dado por la implementación ya que la función de matmul, como hemos mencionado antes, corresponde a una función de la librería cuBLAS. Sin embargo, la función b_n_backward ha sido implementada mediante un *kernel* de CUDA.

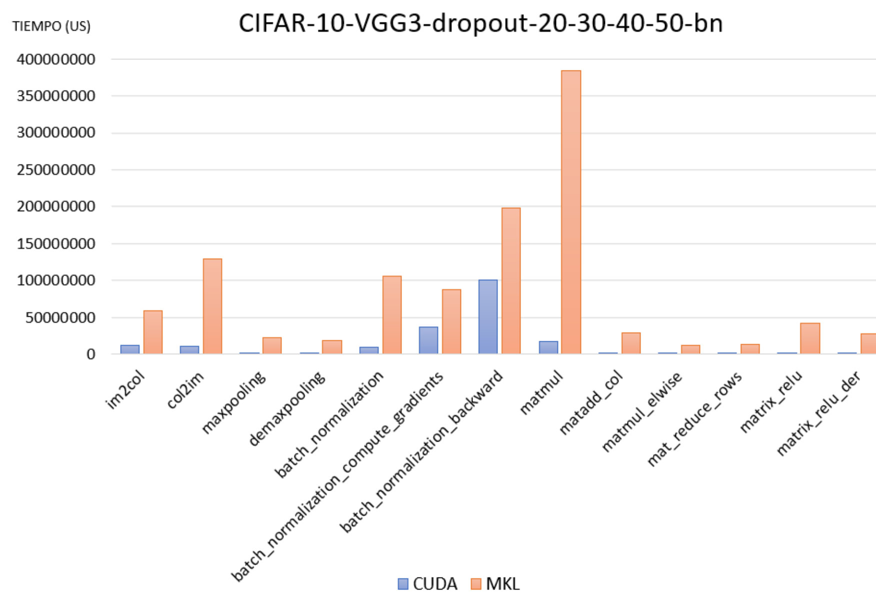

Figura 4.15: Comparación del coste de la ejecución de las funciones.

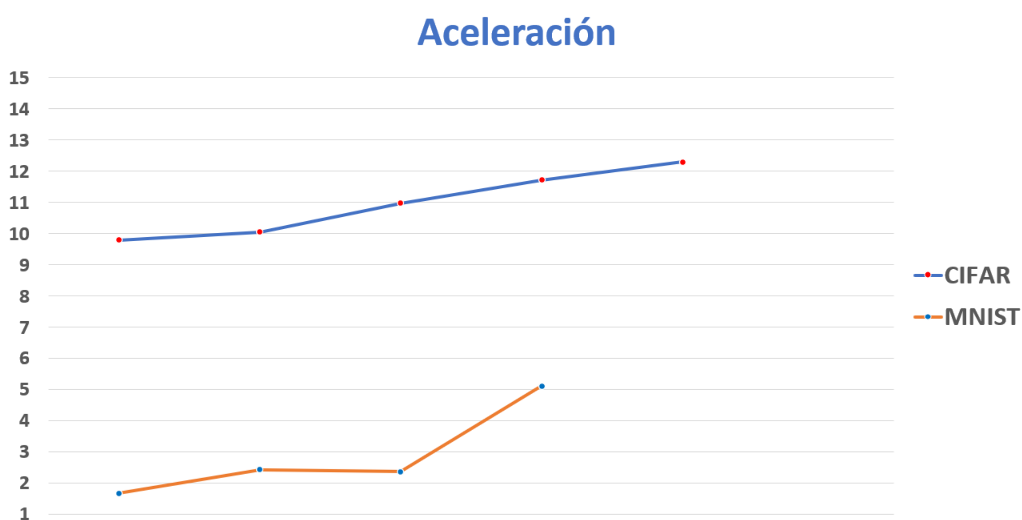

Figura 4.16: Aceleración de cuBLAS respecto MKL de las ejecuciones de las dos clasificaciones.

CAPÍTULO 5
Conclusiones

En este capítulo se presentan las conclusiones obtenidas durante el desarrollo del trabajo. En un primer lugar se expondrán aquellas conclusiones relacionadas con la aplicación de modelados de puentes ferroviarios y, en segundo lugar, las conclusiones obtenidas del soporte dado de la librería cuBLAS en la aplicación HELENNA.

En referente a la aplicación de simulaciones sobre puentes ferroviarios, podemos obtener las siguientes conclusiones:

- Se ha realizado un estudio para analizar las necesidades y problemas actuales de las líneas de ferrocarril y la conservación de puentes ferroviarios.
- Se ha integrado la aplicación original escrita en lenguaje C con la aplicación creada en este trabajo, teniendo que hacer frente a los problemas surgidos por la programación mixta entre los lenguajes C y Fortran tales como los diferentes métodos de ordenación de ambos.
- Se ha conseguido implementar un sistema distribuido escalable mediante la comunicación entre procesos remotos utilizando el paso de mensajes de MPI. Además, también observamos que el funcionamiento de la arquitectura de procesos proporciona una clara optimización frente a la aplicación original, debido a que se ha instaurado la posibilidad de ejecutar un conjunto de simulaciones simultáneamente.
- Se ha implementado la lectura de ficheros externos para permitir la entrada flexible de datos sobre los casos de ejecución.
- Se ha conseguido la optimización del cálculo matricial mediante el uso de la librería cuBLAS.
- Se ha creado un sistema de identificación de GPUs para reconocer su compatibilidad con CUDA y poder otorgar flexibilidad a la aplicación con OpenCL.

Por todo esto, se puede decir que los objetivos relacionados con esta parte, han sido satisfactoriamente alcanzados. Sin embargo, realizando un estudio más exhaustivo, existen varias futuras mejoras que se deben tener en cuenta, las cuales están expuestas en el siguiente capítulo.

Cabe mencionar, que el trabajo realizado sobre esta aplicación ha sido presentado en el congreso *First International Symposium on Risk Analysis and Safety of Complex Structures and Components* realizado en Portugal en 2019 [35].

En segundo lugar, en referencia a la aplicación HELENNA, se pueden obtener las siguientes conclusiones:

- Se ha realizado un estudio de las características principales de las Redes Neuronales, así como de las capas utilizadas en este proyecto. Además, se han analizado e implementado las operaciones complejas necesarias para el entrenamiento de estas Redes Neuronales.
- Se ha conseguido dar soporte de la librería cuBLAS en HELENNA para realizar las operaciones complejas sobre la GPU y conseguir la ejecución esta aplicación sobre sistemas heterogéneos.
- Se ha conseguido dar soporte de la librería cuBLAS en HELENNA a las capas convolucionales, *dropout* y *batch normalization*.

Como resultado, mediante este trabajo se conseguido una notable optimización del cálculo sobre la CPU y, por lo tanto, de la ejecución de las diferentes topologías sobre las clasificaciones MNIST y CIFAR-10, donde podemos ver que el factor de aceleración global llega hasta más de 12. Además, gracias al soporte dado, se ha conseguido la ejecución de la aplicación HELENNA sobre sistemas heterogéneos de manera eficiente.

5.1 Relación del trabajo desarrollado con los estudios cursados

Este trabajo está altamente relacionado con los estudios cursados. Concretamente, se puede observar una fuerte relación con los conceptos dados en la asignatura de Computación Paralela, ya que se ha realizado una implementación de un sistema distribuido mediante MPI. Además, este trabajo también está relacionado con la asignatura de Arquitectura e Ingeniería de Computadores, ya que se ha realizado un estudio de las de unidades CPU y GPU, así como de la arquitectura de la GPU. Finalmente, se puede apreciar una relación con la asignatura de Sistemas Inteligentes, dado que se han explicado conceptos, problemas y resoluciones de las Redes Neuronales.

CAPÍTULO 6
Trabajo futuro

En este capítulo se presentan las futuras mejoras que se pueden implementar sobre las soluciones desarrollas en las aplicaciones presentadas en este trabajo.

6.1 Aplicación para Cálculo Dinámico por Elementos Finitos de puentes ferroviarios

6.1.1. Envío de datos circular

El primer problema que nos encontramos con la implementación realizada está relacionado con el envío de los datos de cada simulación que se realiza desde el *master* hacia los *kernels*. Como se ha mencionado anteriormente, la configuración elegida ha sido circular, esto significa que el *master* envía los datos de las simulaciones a los Kernels según su identificador de manera ascendente. En caso de que haya más casos que *kernels*, el *master* volvería a enviar el siguiente caso al primer *kernel* y seguiría, otra vez, de manera ascendente.

Dado que la configuración elegida ha sido circular, esto claramente puede llevar a un problema de suspensión de *kernels*, puesto que no nos aseguramos de que el acceso a la GPU se dé por *kernels* secuenciales. Además, también nos encontramos ante una situación en la que el tamaño de las simulaciones puede ser desigual, independientemente de en qué lugar va a ser lanzada dicha simulación. Esto podría dar lugar a que los *kernels* que han empezado los cálculos primeramente no sean los primeros en finalizar. Al ocurrir, los *kernels* que han terminado la simulación se quedan a la espera hasta la recepción de otra porción de datos para poder realizar la siguiente simulación. Dicha recepción no ocurrirá hasta que el Kernel anterior a él termine su simulación y reciba los siguientes datos, dado que los envíos del *master* son síncronos.

Una posible solución para este problema sería organizar otro tipo de envío de forma que el *master* espere un mensaje de terminación por parte de los *kernels* para proceder a enviar el siguiente caso de prueba para la simulación.

6.1.2. Almacenamiento persistente de matrices en la memoria de la GPU

Como hemos mencionado anteriormente, realizar copias entre la memoria de la CPU y la GPU provoca que el tiempo de las multiplicaciones en la GPU sea mayor a lo necesario. Además, nos encontramos delante de un escenario donde numerosas multiplicaciones se realizan con matrices iguales. Por lo que, al borrarse las matrices almacenadas

después la multiplicación, se da el caso en el que las matrices son borradas, pero en la siguiente multiplicación una de ellas se vuelve a copiar a la GPU.

Mejorar el cálculo, se puede disminuir el número de copias entre la memoria de las dos unidades. Esto sería posible almacenando en la memoria de la GPU aquellas matrices que se utilizan reiteradas veces en los cálculos de multiplicaciones, lo cual es posible si se analiza el núcleo interno del simulador en Fortran. De esta manera, cada vez que la simulación Fortran necesite el resultado de una multiplicación, este le indicaría si las matrices de la multiplicación necesitan ser guardadas, borradas o si, por lo contrario, ya se encuentran almacenadas en la memoria de la GPU. Además, esta gestión implicaría llevar un control de la memoria de la GPU utilizada.

6.1.3. Implementación de un sistema de colas en el *server*

Actualmente, los procesos *server* reciben todos los datos necesarios mediante los mensajes de MPI y, posteriormente, realizan la multiplicación en la GPU, secuencialmente. Por esta razón, cuando el *server* está realizando el paso de mensajes, la GPU queda inactiva. De la misma manera, cuando la GPU está realizando el cálculo la CPU también queda inactiva. Podemos observar una posible mejora a nuestra implementación, ya que se podrían usar hilos en el proceso *server* para permitir el uso simultáneamente las dos unidades.

Esto puede ser posible mediante la implementación de hilos y colas en el *server*. De esta manera, un hilo se encargaría de almacenar las peticiones provenientes del proceso *kernel* en la cola, y otro hilo se encargaría se desencolar las peticiones, realizar la multiplicación y enviar el resultado.

6.1.4. Soporte para otras rutinas

La aplicación responsable de calcular las simulaciones llama a un conjunto de rutinas de cálculo complejo las cuales están orientadas a resolver ecuaciones lineales y no han sido implementadas en CUDA. Dado que existe un protocolo de comunicación, existe la posibilidad de indicarle al *server* el tipo de operación que se necesita realizar. De esta manera, con el fin de aportar un mayor grado de optimización, se pueden implementar estas funciones para ejecutarse en la GPU.

6.2 Aplicación HELENNA

6.2.1. Implementación de *streams* de CUDA

Otra optimización que se podría implementar es el uso de los *streams* de CUDA para conseguir una ocupación total de la GPU. Actualmente, cuando se necesita una copia entre los datos de las dos unidades CPU y GPU se llama a la función *cudaMemcpy*, la cual, como hemos mencionado anteriormente, es una función síncrona, por lo que la GPU y la CPU se quedan bloqueadas hasta que la transferencia termine. Además, algunas funciones lanzadas a la GPU tienen muy poca ocupación dentro de la misma, por lo que se podrían lanzar un conjunto de estas funciones para aumentar la capacidad de ocupación dentro de la GPU.

6.2.2. Mejora de la función implementada col2im de la capa convolucional

Como hemos mencionado anteriormente la función col2im implementa en la capa convolucional solo acepta dimensiones SW y SH iguales a KW y KH. Una clara optimización sería eliminar esta restricción implementando una función flexible. Para esto, se debe eliminar la posibilidad de que varios hilos puedan acceder al mismo elemento. Esto se puede lograr añadiendo un bucle, de manera que solo un hilo sea el responsable de realizar la suma de los elementos de entrada.

Bibliografía

[1] Nvidia. «CUDA C++ PROGRAMMING GUIDE». En: (2020).

[2] Fernando Sancho Caparrini. *Entrenamiento de Redes Neuronales: mejorando el Gradiente Descendiente*. URL: http://www.cs.us.es/~fsancho/?e=165. (acceso: 13.06.2020).

[3] Albert Soto i Serrano. «YOLO Object Detector for Onboard DrivingImages». En: (2017).

[4] Jonathan Hui. *Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3*. URL: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088. (acceso: 13.06.2020).

[5] Intel. *Intel® MPI Library*. URL: https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html. (acceso: 13.06.2020).

[6] V. Codreanu y col. «Evaluating automatically parallelized versions of the support vector machine». En: *Concurrency and Computation* (2014).

[7] Rasa Remenyte-Prescott Matteo Vagnoli y John Andrews. «Railway bridge structural health monitoring and fault detection: State-of-the-art methods and future challenges». En: *Structural Health Monitoring* 17.4 (2018), págs. 971-1007.

[8] *Plataforma de contratación del sectur público*. URL: https://contrataciondelestado.es/wps/portal/!ut/p/b0/04_Sj9CPykssy0xPLMnMz0vMAfIjU1JTC3Iy87KtUlJLEn%20%5CNyUuNzMpMzSxKTgQr0w_Wj9KMyU1zLcvQjDTPyDb%20%5CV9cksKI5KDS9OLSxy%5Cd3ItMHW1t9Qtycx0B2p_8Yg!!/. (acceso: 29.6.2020).

[9] NVIDIA. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone. (acceso: 03.03.2020).

[10] NVIDIA. *NVIDIA*. URL: https://www.nvidia.com/es-es/. (acceso: 03.03.2020).

[11] Khronos Group. *OpenCL Overview*. URL: https://www.khronos.org/opencl/. (acceso: 12.05.2020).

[12] Cedric Nugteren. *Tutorial: OpenCL SGEMM tuning for Kepler*. URL: https://cnugteren.github.io/tutorial/pages/page11.html. (acceso: 12.0.2020).

[13] NVIDIA. «cuBLAS: The API Reference guide for cuBLAS, the CUDA Basic Linear Algebra Subroutine library.» En: (2020).

[14] NVIDIA. *CUDA-MEMCHECK*. URL: https://developer.nvidia.com/cuda-memcheck. (acceso: 13.06.2020).

[15] NVIDIA. *NVIDIA Tesla P100*. URL: https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/. (acceso: 04.05.2020).

[16] NVIDIA. *Arquitectura NVIDIA Ampere*. URL: https://www.nvidia.com/es-es/data-center/nvidia-ampere-gpu-architecture/. (acceso: 30.06.2020).

[17] NVIDIA. *NVIDIA Tensor Cores*. URL: https://www.nvidia.com/en-us/data-center/tensor-cores/. (acceso: 30.06.2020).

[18] *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional., 2013. ISBN: 978-0-321-80946-9.

[19] NVIDIA. *Using Shared Memory in CUDA C/C++*. URL: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/. (acceso: 06.05.2020).

[20] Justin Luitjens. «CUDA Streams: Best Practices and Common Pitfalls». En: *GPU Technology Conference* (2014).

[21] NVIDIA. *cuBLAS*. URL: https://developer.nvidia.com/cublas. (acceso: 12.05.2020).

[22] Intel. *Intel® Math Kernel Library*. URL: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html. (acceso: 12.05.2020).

[23] Intel. *Developer Reference for Intel® Math Kernel Library 2020: cblas?gemm*. URL: https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines/blas-routines/blas-level-3-routines/cblas-gemm.html. (acceso: 12.05.2020).

[24] NVIDIA. *CUDA 6.5 Performance Report*. URL: http://developer.download.nvidia.com/compute/cuda/6_5/rel/docs/CUDA_6.5_Performance_Report.pdf. (acceso: 14.06.2020).

[25] SURFsara Cedric Nugteren. *Tutorial: OpenCL SGEMM tuning for Kepler*. URL: https://cnugteren.github.io/tutorial/pages/page1.html. (acceso: 12.05.2020).

[26] OpenMPI. *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/. (acceso: 13.06.2020).

[27] MPICH. *High-Performance Portable MPI*. URL: https://www.mpich.org/. (acceso: 13.06.2020).

[28] Message Passing Interface Forum. «MPI: A Message-Passing Interface Standard-Version 3.1». En: (2015).

[29] William Gropp, Ewing Lusk y Anthony Skjellum. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. ProQuest Ebook Central, 2014. ISBN: 978-0262571326.

[30] *BLAS (Basic Linear Algebra Subprograms)*. URL: http://www.netlib.org/blas/. (acceso: 12.06.2020).

[31] *LAPACK — Linear Algebra PACKage*. URL: http://www.netlib.org/lapack/. (acceso: 12.06.2020).

[32] Intel. *Intel® Math Kernel Library Developer Reference*. URL: https://software.intel.com/content/www/us/en/develop/articles/mkl-reference-manual.html?language=en. (acceso: 12.05.2020).

[33] Avinash Sodani Jim Jeffers James Reinders. *Intel Xeon Phi Processor High Performance Programming*. Knights Landing Edition, 2016. ISBN: 978-0-12-809194-4.

[34] Intel. *Improving Performance of Math Functions with Intel® Math Kernel Library*. URL: https://software.intel.com/content/www/us/en/develop/articles/improving-performance-of-math-functions-with-intel-math-kernel-library.html. (acceso: 12.05.2020).

[35] P. Museros L. Medina M. Gavilán, R. Palma y J. Flich. «Updated parallel computing strategies for nondeterministic dynamic analysis of railway bridges». En: *First International Symposium on Risk Analysis and Safety of Complex Structures and Componen* (2019), págs. 199-200.

[36] Ministerio de Fomento. *ITPF-05: Instrucción sobre las inspecciones técnicas en los puentes de ferrocarril*. 2005.

[37] João M. Santos Pereira da Rocha. «Probabilistic methodologies for the safety assessment of short span railway bridges for high-speed traffic. Tesis Doctoral.» En: *Universidad de Oporto* (2015).

[38] *openMP*. URL: https://www.openmp.org/. (acceso: 15.06.2020).

[39] Lawrence Livermore National Laboratory Blaise Barney. *POSIX Threads Programming*. URL: https://www.mpich.org/static/docs/latest/www3/MPI_Comm_get_parent.htmlhttps://computing.llnl.gov/tutorials/pthreads/. (acceso: 15.06.2020).

[40] Arjen J. Markus. *Modern Fortran in practice*. Cambridge University Press, 2012.

[41] *Subroutine DSYSV*. URL: http://www.netlib.org/lapack/lapack-3.1.1/html/dsysv.f.html. (acceso: 1.7.2020).

[42] *H2020 RECIPE*. URL: http://www.recipe-project.eu/. (acceso: 29.6.2020).

[43] *Cloud Tensor Processing Unit (TPU)*. URL: https://cloud.google.com/tpu/docs/tpus?hl=es-419. (acceso: 29.6.2020).

[44] *Jetson AGX Xavier*. URL: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/. (acceso: 29.6.2020).

[45] Jordi Torres. *Deep Learning: Introducción práctica con Keras*. Lulu Press, Inc., 2018. ISBN: 978-0-244-07895-9.

[46] *CNN Implementation*. URL: https://kunicom.blogspot.com/2017/08/28-cnn-implementation.html. (acceso: 20.6.2020).

[47] Shams S. *Overfitting and Regularization*. URL: https://machinelearningmedium.com/2017/09/08/overfitting-and-regularization/. (acceso: 20.6.2020).

[48] Peter Sadowsk Pierre Baldi. «The dropout learning algorithm». En: *Artificial Intelligenc* 210 (2014), págs. 78-122.

[49] Nitish Srivastava y col. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». En: *Machine Learning Resear* 15 (2014), págs. 1929-19.

APÉNDICE A

Código de la implementación de la aplicación para el Cálculo Dinámico por Elementos Finitos de puentes ferroviarios

A.1 Creación de procesos

```
MPI_Comm spawnProcesses(int number_kernels, int number_servers)
{
   MPI_Info info[2];
   //se indican los identificadores de las maquinas en las que se desean crear
   //los procesos y el numero de procesos a crear en cada maquina
   char kernelsHost[100] = "IP1:number_kernels";
   char serversHost[100] = "IP2:number_servers";

    //creacion del objeto MPI_Info
   MPI_Info_create(&info[0]);
   MPI_Info_set(info[0], "host", kernelsHost);
   MPI_Info_create(&info[1]);
   MPI_Info_set(info[1], "host", serversHost);

    //identificacion de los programas a ejecutar por los nuevos procesos creados
   char *cmds[2] = {"./kernel",
               "./server"};
   int np[2] = {kernels, servers};
   int errcodes[2];

    //llamada a la funcion responsable de crear los procesos
   MPI_Comm intercomm;
   MPI_Comm_spawn_multiple(2, cmds, MPI_ARGVS_NULL, np, info, 0, MPI_COMM_WORLD,
                 &intercomm, errcodes);

   return intercomm;
}
```

Nótese que para ejecutar correctamente este código se debe cambiar IP por el identificador de la máquina.

A.2 Identificación de GPU

```
void get_device() {
   //Comprobamos si la targeta tiene una GPU compatible con CUDA
   int gpusCUDAcompatibles = 0;
   cudaGetDeviceCount (&gpusCUDAcompatibles);
   if(gpusCUDAcompatibles > 0)
      cudaDEVICE = 1;
   if(cudaDEVICE)
      initCUDA();
   else
      initOPENCL();
}
```

Listing A.1: Identificación de la GPU.

APÉNDICE B
Configuración de las pruebas de evaluación

En este apéndice mostramos las configuraciones utilizadas para las pruebas de evaluación en la contribución de redes neuronales. En concreto mostramos la configuración de todas las redes neuronales utilizadas. En cada una de ellas se tabula las diferentes capas neuronales, su tipología, y su geometría en término de número de neuronas y de configuración, como por ejemplo la configuración de las capas convolucionales (número de canales, tamaños de imágenes, ...) y capas dropout (porcentaje de neuronas desactivadas). Las configuraciones mostradas en la figura B.1 son iguales para las ejecuciones de todas las topologías.

```
Configuration (training):
  Gradient descend method               : minibatch
  Minibatch size                        : 64
  Learning rate                         : 0.0100 (constant)
  Momentum                              : 0.9000
  Gradient value clipping               : 0.0000 not used
  Loss function                         : cross entropy (ce)
  L1 regularization                     : no
  L2 regularization                     : no
  Lambda                                : 0.0000
  Block pruning lambda                  : 0.0000
  Block pruning row size                : 1
  Block pruning col size                : 1
  Truncation threshold                  : 0.0000
  Synthetic optimization function       : None
  Training dataset size                 : 49984
  Test dataset size                     : 9984
  Number of epochs                      : 5
  Dataset directory                     : datasets/cifar-10
  Plot                                  : None
  Fused im2col                          : no
  Load model                            : --
  Save model                            : --
  Generate confusion matrix image       : no
  Fit decoupled                         : yes
  Preload images                        : no
  Broadcast images                      : no
  Disjoint training set                 : no
  Chunk size (in items)                 : 49984
--------------------------------------------------------------------------------------------------|
|devices                                                                                          |
|-------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga  | opencl-gpu   | cublas       | clblas  |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|             |             |             |             |             | yes         |             |
|-------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                      |
| MPI: no - 1 processes                                                                           |
| Force synchronization (for cublas and clblas): yes                                              |
|-------------------------------------------------------------------------------------------------|
```

Figura B.1: Configuración de las topologías

```
|------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                               |
|------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type                | neurons | params    | in_layer | configuration                   | memory     |
|-----|-----------|--------------------------|---------|-----------|----------|---------------------------------|------------|
|    0|input      |input layer               |      784|          0|          |inputs 784, outputs 784          |     0.00 GB|
|    1|fc1_0      |fully connected           |      784|     615440|input_0   |inputs 784, outputs 784          |     0.01 GB|
|    2|relu1_0    |relu                      |      784|          0|fc1_0     |                                 |     0.00 GB|
|    3|fc2_0      |fully connected           |      256|     200960|relu1_0   |inputs 784, outputs 256          |     0.00 GB|
|    4|relu2_0    |relu                      |      256|          0|fc2_0     |                                 |     0.00 GB|
|    5|fc3_0      |fully connected           |       10|       2570|relu2_0   |inputs 256, outputs 10           |     0.00 GB|
|    6|softmax_0  |softmax                   |       10|          0|fc3_0     |                                 |     0.00 GB|
|------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                     |     2100|     818970|          |Memory (no layers):    0.00 GB   |     0.02 GB|
|------------------------------------------------------------------------------------------------------------------------|
```

Figura B.2: Topología MLP-Medium.

```
-----------------------------------------------------------------------------------------------------|
|devices                                                                                             |
|----------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga  | opencl-gpu   | cublas       | clblas       |
|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
|              |              |              |              |              | yes          |              |
|----------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                         |
| MPI: no - 1 processes                                                                              |
| Force synchronization (for cublas and clblas): yes                                                  |
|----------------------------------------------------------------------------------------------------|

|-----------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type                | neurons | params    | in_layer | configuration                  | memory     |
|-----|-----------|--------------------------|---------|-----------|----------|--------------------------------|------------|
|    0|input      |input layer               |      784|          0|          |inputs 784, outputs 784         |    0.00 GB|
|    1|fc1_0      |fully connected           |     1000|     785000|input_0   |inputs 784, outputs 1000        |    0.02 GB|
|    2|relu1_0    |relu                      |     1000|          0|fc1_0     |                                |    0.00 GB|
|    3|fc2_0      |fully connected           |     1000|    1001000|relu1_0   |inputs 1000, outputs 1000       |    0.02 GB|
|    4|relu2_0    |relu                      |     1000|          0|fc2_0     |                                |    0.00 GB|
|    5|fc3_0      |fully connected           |       10|      10010|relu2_0   |inputs 1000, outputs 10         |    0.00 GB|
|    6|smax_0     |softmax                   |       10|          0|fc3_0     |                                |    0.00 GB|
|-----------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                     |     4020|    1796010|          |Memory (no layers):    0.00 GB  |    0.04 GB|
|-----------------------------------------------------------------------------------------------------------------------|
```

Figura B.3: Topología MLP-Big.

```
-----------------------------------------------------------------------------------------------------|
|devices                                                                                             |
|----------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga  | opencl-gpu   | cublas       | clblas       |
|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
|              |              |              |              |              | yes          |              |
|----------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                         |
| MPI: no - 1 processes                                                                              |
| Force synchronization (for cublas and clblas): yes                                                  |
|----------------------------------------------------------------------------------------------------|

|----------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                             |
|----------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type                | neurons | params    | in_layer | configuration                 | memory     |
|-----|-----------|--------------------------|---------|-----------|----------|-------------------------------|------------|
|    0|input      |input layer               |      784|          0|          |inputs 784, outputs 784        |    0.00 GB|
|    1|fc1_0      |fully connected           |     2000|    1570000|input_0   |inputs 784, outputs 2000       |    0.03 GB|
|    2|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|    3|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|    4|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|    5|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|    6|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|    7|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|    8|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|    9|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   10|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|   11|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   12|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|   13|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   14|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|   15|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   16|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|   17|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   18|relu1_0    |relu                      |     2000|          0|fc1_0     |                               |    0.00 GB|
|   19|fc1_0      |fully connected           |     2000|    4002000|relu1_0   |inputs 2000, outputs 2000      |    0.08 GB|
|   20|output_0   |softmax                   |     2000|          0|fc1_0     |                               |    0.00 GB|
|----------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                     |    40000|   37588000|          |Memory (no layers):    0.00 GB |    0.73 GB|
|----------------------------------------------------------------------------------------------------------------------|
```

Figura B.4: Topología MLP-Large.

```
--------------------------------------------------------------------------------------------------|
|devices                                                                                           |
|--------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga  | opencl-gpu   | cublas       | clblas       |
|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
|              |              |              |              |              | yes          |              |
|--------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                       |
| MPI: no - 1 processes                                                                            |
| Force synchronization (for cublas and clblas): yes                                                |
|--------------------------------------------------------------------------------------------------|

|------------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                             |
|------------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type              | neurons | params    | in_layer | configuration                                                | memory      |
|-----|-----------|------------------------|---------|-----------|----------|--------------------------------------------------------------|-------------|
|    0|input      |input layer             |      784|          0|          |inputs 784, outputs 784                                       |      0.00 GB|
|    1|conv1_0    |convolutional           |      784|         10|input_0   |IN: 1x28x28 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 1x28x2( |      0.00 GB|
|    2|relu1_0    |relu                    |      784|          0|conv1_0   |                                                              |      0.00 GB|
|    3|maxp1_0    |maxpooling              |      196|          0|relu1_0   |IN: 1x28x28 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 1x14x14    |      0.00 GB|
|    4|fc1_0      |fully connected         |       10|       1970|maxp1_0   |inputs 196, outputs 10                                        |      0.00 GB|
|    5|softmax_0  |softmax                 |       10|          0|fc1_0     |                                                              |      0.00 GB|
|------------------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                   |     1784|       1980|          |Memory (no layers):   0.00 GB                                 |      0.01 GB|
|------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.5: Topología convolucional.

```
-----------------------------------------------------------------------------------------------|
|devices                                                                                       |
|----------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas       | clblas       |
|--------------|--------------|--------------|-------------|-------------|--------------|--------------|
|              |              |              |             |             | yes          |              |
|----------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                   |
| MPI: no - 1 processes                                                                        |
| Force synchronization (for cublas and clblas): yes                                            |
|----------------------------------------------------------------------------------------------|

|------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type              | neurons | params    | in_layer | configuration                                                 | memory      |
|-----|-----------|------------------------|---------|-----------|----------|---------------------------------------------------------------|-------------|
|    0|input      |input layer             |      784|          0|          |inputs 784, outputs 784                                        |    0.00 GB|
|    1|conv1_0    |convolutional           |    12544|        160|input_0   |IN: 1x28x28 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 16x28x28 |    0.01 GB|
|    2|relu1_0    |relu                    |    12544|          0|conv1_0   |                                                               |    0.01 GB|
|    3|maxp1_0    |maxpooling              |     3136|          0|relu1_0   |IN: 16x28x28 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 16x14x14  |    0.00 GB|
|    4|fc2_0      |fully connected         |       10|      31370|maxp1_0   |inputs 3136, outputs 10                                        |    0.00 GB|
|    5|softmax_0  |softmax                 |       10|          0|fc2_0     |                                                               |    0.00 GB|
|------------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                   |    28244|      31530|          |Memory (no layers):   0.00 GB                                  |    0.03 GB|
|------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.6: Topología convolucional-16 canalaes.

```
---------------------------------------------------------------------------------------------|
|devices                                                                                     |
|--------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas      | clblas      |
|--------------|--------------|--------------|-------------|-------------|-------------|-------------|
|              |              |              |             |             | yes         |             |
|--------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                 |
| MPI: no - 1 processes                                                                      |
| Force synchronization (for cublas and clblas): yes                                          |
|--------------------------------------------------------------------------------------------|

|---------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type             | neurons | params    | in_layer | configuration                                               | memory      |
|-----|-----------|-----------------------|---------|-----------|----------|-------------------------------------------------------------|-------------|
|    0|input      |input layer            |     3072|          0|          |inputs 3072, outputs 3072                                    |     0.00 GB|
|    1|conv1_0    |convolutional          |    32768|        896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32 |     0.04 GB|
|    2|relu1_0    |relu                   |    32768|          0|conv1_0   |                                                             |     0.02 GB|
|    3|conv2_0    |convolutional          |    32768|       9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32 |     0.16 GB|
|    4|relu2_0    |relu                   |    32768|          0|conv2_0   |                                                             |     0.02 GB|
|    5|maxp1_0    |maxpooling             |     8192|          0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16 |     0.01 GB|
|    6|fc1_0      |fully connected        |      128|    1048704|maxp1_0   |inputs 8192, outputs 128                                     |     0.02 GB|
|    7|relu3_0    |relu                   |      128|          0|fc1_0     |                                                             |     0.00 GB|
|    8|fc2_0      |fully connected        |       10|       1290|relu3_0   |inputs 128, outputs 10                                       |     0.00 GB|
|    9|sfmax1_0   |softmax                |       10|          0|fc2_0     |                                                             |     0.00 GB|
|---------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                  |   139540|    1060138|          |Memory (no layers):   0.00 GB                                |     0.28 GB|
|---------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.7: Topología VGG1.

```
|-----------------------------------------------------------------------------------------------|
|devices                                                                                        |
|-----------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas       | clblas       |
|--------------|--------------|--------------|-------------|-------------|--------------|--------------|
|              |              |              |             |             | yes          |              |
|-----------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                     |
| MPI: no - 1 processes                                                                          |
| Force synchronization (for cublas and clblas): yes                                              |
|-----------------------------------------------------------------------------------------------|

|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name      |layer type           | neurons | params   | in_layer | configuration                                                       | memory      |
|-----|----------|---------------------|---------|----------|----------|---------------------------------------------------------------------|-------------|
|    0|input     |input layer          |     3072|         0|          |inputs 3072, outputs 3072                                            |      0.00 GB|
|    1|conv1_0   |convolutional        |    32768|       896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32       |      0.04 GB|
|    2|relu1_0   |relu                 |    32768|         0|conv1_0   |                                                                     |      0.02 GB|
|    3|conv2_0   |convolutional        |    32768|      9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32      |      0.16 GB|
|    4|relu2_0   |relu                 |    32768|         0|conv2_0   |                                                                     |      0.02 GB|
|    5|maxp1_0   |maxpooling           |     8192|         0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16        |      0.01 GB|
|    6|conv4_0   |convolutional        |    16384|     18496|maxp1_0   |IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16      |      0.05 GB|
|    7|relu3_0   |relu                 |    16384|         0|conv4_0   |                                                                     |      0.01 GB|
|    8|conv4_0   |convolutional        |    16384|     36928|relu3_0   |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16      |      0.08 GB|
|    9|relu4_0   |relu                 |    16384|         0|conv4_0   |                                                                     |      0.01 GB|
|   10|maxp2_0   |maxpooling           |     4096|         0|relu4_0   |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8          |      0.00 GB|
|   11|conv5_0   |convolutional        |     8192|     73856|maxp2_0   |IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8         |      0.02 GB|
|   12|relu5_0   |relu                 |     8192|         0|conv5_0   |                                                                     |      0.01 GB|
|   13|conv6_0   |convolutional        |     8192|    147584|relu5_0   |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8        |      0.04 GB|
|   14|relu6_0   |relu                 |     8192|         0|conv6_0   |                                                                     |      0.01 GB|
|   15|maxp3_0   |maxpooling           |     2048|         0|relu6_0   |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4          |      0.00 GB|
|   16|fc1_0     |fully connected      |      128|    262272|maxp3_0   |inputs 2048, outputs 128                                             |      0.00 GB|
|   17|relu7_0   |relu                 |      128|         0|fc1_0     |                                                                     |      0.00 GB|
|   18|fc2_0     |fully connected      |       10|      1290|relu7_0   |inputs 128, outputs 10                                               |      0.00 GB|
|   19|softmax_0 |softmax              |       10|         0|fc2_0     |                                                                     |      0.00 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     |          |TOTAL                |   243988|    550570|          |Memory (no layers):   0.00 GB                                        |      0.50 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.8: Topología VGG2.

```
|----------------------------------------------------------------------------------------------------|
|devices                                                                                             |
|----------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas       | clblas       |
|--------------|--------------|--------------|-------------|-------------|--------------|--------------|
|              |              |              |             |             | yes          |              |
|----------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                         |
| MPI: no - 1 processes                                                                              |
| Force synchronization (for cublas and clblas): yes                                                  |
|----------------------------------------------------------------------------------------------------|

|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                                    |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name      |layer type             | neurons | params   | in_layer | configuration                                                     | memory       |
|-----|----------|-----------------------|---------|----------|----------|-------------------------------------------------------------------|--------------|
|    0|input     |input layer            |     3072|         0|          |inputs 3072, outputs 3072                                          |       0.00 GB|
|    1|conv1_0   |convolutional          |    32768|       896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32     |       0.04 GB|
|    2|relu1_0   |relu                   |    32768|         0|conv1_0   |                                                                   |       0.02 GB|
|    3|conv2_0   |convolutional          |    32768|      9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32    |       0.16 GB|
|    4|relu2_0   |relu                   |    32768|         0|conv2_0   |                                                                   |       0.02 GB|
|    5|maxp1_0   |maxpooling             |     8192|         0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16      |       0.01 GB|
|    6|conv4_0   |convolutional          |    16384|     18496|maxp1_0   |IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16    |       0.05 GB|
|    7|relu3_0   |relu                   |    16384|         0|conv4_0   |                                                                   |       0.01 GB|
|    8|conv4_0   |convolutional          |    16384|     36928|relu3_0   |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16    |       0.08 GB|
|    9|relu4_0   |relu                   |    16384|         0|conv4_0   |                                                                   |       0.01 GB|
|   10|maxp2_0   |maxpooling             |     4096|         0|relu4_0   |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8        |       0.00 GB|
|   11|conv5_0   |convolutional          |     8192|     73856|maxp2_0   |IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8       |       0.02 GB|
|   12|relu5_0   |relu                   |     8192|         0|conv5_0   |                                                                   |       0.01 GB|
|   13|conv6_0   |convolutional          |     8192|    147584|relu5_0   |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8      |       0.04 GB|
|   14|relu6_0   |relu                   |     8192|         0|conv6_0   |                                                                   |       0.01 GB|
|   15|maxp3_0   |maxpooling             |     2048|         0|relu6_0   |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4        |       0.00 GB|
|   16|fc1_0     |fully connected        |      128|    262272|maxp3_0   |inputs 2048, outputs 128                                           |       0.00 GB|
|   17|relu7_0   |relu                   |      128|         0|fc1_0     |                                                                   |       0.00 GB|
|   18|fc2_0     |fully connected        |       10|      1290|relu7_0   |inputs 128, outputs 10                                             |       0.00 GB|
|   19|softmax_0 |softmax                |       10|         0|fc2_0     |                                                                   |       0.00 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     |          |TOTAL                  |   243988|    550570|          |Memory (no layers):   0.00 GB                                      |       0.50 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.9: Topología VGG3.

```
------------------------------------------------------------------------------------------------|
|devices                                                                                        |
|-----------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu   | cublas       | clblas      |
|--------------|--------------|--------------|-------------|--------------|--------------|-------------|
|              |              |              |             |              | yes          |             |
|-----------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                    |
| MPI: no - 1 processes                                                                         |
| Force synchronization (for cublas and clblas): yes                                            |
|-----------------------------------------------------------------------------------------------|

|---------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type           | neurons | params    | in_layer | configuration                                                    | memory     |
|-----|-----------|---------------------|---------|-----------|----------|------------------------------------------------------------------|------------|
|    0|input      |input layer          |     3072|          0|          |inputs 3072, outputs 3072                                         |     0.00 GB|
|    1|conv1_0    |convolutional        |    32768|        896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32    |     0.04 GB|
|    2|relu1_0    |relu                 |    32768|          0|conv1_0   |                                                                  |     0.02 GB|
|    3|conv2_0    |convolutional        |    32768|       9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32   |     0.16 GB|
|    4|relu2_0    |relu                 |    32768|          0|conv2_0   |                                                                  |     0.02 GB|
|    5|maxp1_0    |maxpooling           |     8192|          0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16     |     0.01 GB|
|    6|dropout1_0 |dropout              |     8192|          0|input_0   |RATE: 0.200000                                                    |     0.01 GB|
|    7|conv4_0    |convolutional        |    16384|      18496|dropout1_0|IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16   |     0.05 GB|
|    8|relu3_0    |relu                 |    16384|          0|conv4_0   |                                                                  |     0.01 GB|
|    9|conv4_0    |convolutional        |    16384|      36928|relu3_0   |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16   |     0.08 GB|
|   10|relu4_0    |relu                 |    16384|          0|conv4_0   |                                                                  |     0.01 GB|
|   11|maxp2_0    |maxpooling           |     4096|          0|relu4_0   |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8       |     0.00 GB|
|   12|dropout2_0 |dropout              |     4096|          0|input_0   |RATE: 0.200000                                                    |     0.00 GB|
|   13|conv5_0    |convolutional        |     8192|      73856|dropout2_0|IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8      |     0.02 GB|
|   14|relu5_0    |relu                 |     8192|          0|conv5_0   |                                                                  |     0.01 GB|
|   15|conv6_0    |convolutional        |     8192|     147584|relu5_0   |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8     |     0.04 GB|
|   16|relu6_0    |relu                 |     8192|          0|conv6_0   |                                                                  |     0.01 GB|
|   17|maxp3_0    |maxpooling           |     2048|          0|relu6_0   |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4       |     0.00 GB|
|   18|dropout3_0 |dropout              |     2048|          0|input_0   |RATE: 0.200000                                                    |     0.00 GB|
|   19|fc1_0      |fully connected      |      128|     262272|dropout3_0|inputs 2048, outputs 128                                          |     0.00 GB|
|   20|relu7_0    |relu                 |      128|          0|fc1_0     |                                                                  |     0.00 GB|
|   21|dropout4_0 |dropout              |      128|          0|input_0   |RATE: 0.200000                                                    |     0.00 GB|
|   22|fc2_0      |fully connected      |       10|       1290|dropout4_0|inputs 128, outputs 10                                            |     0.00 GB|
|   23|softmax_0  |softmax              |       10|          0|fc2_0     |                                                                  |     0.00 GB|
|---------------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                |   258452|     550570|          |Memory (no layers):    0.00 GB                                    |     0.52 GB|
|---------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.10: Topología VGG3-dropout.

```
|-------------------------------------------------------------------------------------------------------|
|devices                                                                                                |
|-------------------------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas       | clblas       |
|--------------|--------------|--------------|-------------|-------------|--------------|--------------|
|              |              |              |             |             | yes          |              |
|-------------------------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                                            |
| MPI: no - 1 processes                                                                                 |
| Force synchronization (for cublas and clblas): yes                                                     |
|-------------------------------------------------------------------------------------------------------|

|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name       |layer type              | neurons | params    | in_layer | configuration                                                         | memory       |
|-----|-----------|------------------------|---------|-----------|----------|-----------------------------------------------------------------------|--------------|
|    0|input      |input layer             |     3072|          0|          |inputs 3072, outputs 3072                                              |       0.00 GB|
|    1|conv1_0    |convolutional           |    32768|        896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32        |       0.04 GB|
|    2|relu1_0    |relu                    |    32768|          0|conv1_0   |                                                                       |       0.02 GB|
|    3|conv2_0    |convolutional           |    32768|       9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32       |       0.16 GB|
|    4|relu2_0    |relu                    |    32768|          0|conv2_0   |                                                                       |       0.02 GB|
|    5|maxp1_0    |maxpooling              |     8192|          0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16         |       0.01 GB|
|    6|dropout1_0|dropout                  |     8192|          0|input_0   |RATE: 0.200000                                                         |       0.01 GB|
|    7|conv4_0    |convolutional           |    16384|      18496|dropout1_0|IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16       |       0.05 GB|
|    8|relu3_0    |relu                    |    16384|          0|conv4_0   |                                                                       |       0.01 GB|
|    9|conv4_0    |convolutional           |    16384|      36928|relu3_0   |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16       |       0.08 GB|
|   10|relu4_0    |relu                    |    16384|          0|conv4_0   |                                                                       |       0.01 GB|
|   11|maxp2_0    |maxpooling              |     4096|          0|relu4_0   |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8           |       0.00 GB|
|   12|dropout2_0|dropout                  |     4096|          0|input_0   |RATE: 0.300000                                                         |       0.00 GB|
|   13|conv5_0    |convolutional           |     8192|      73856|dropout2_0|IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8          |       0.02 GB|
|   14|relu5_0    |relu                    |     8192|          0|conv5_0   |                                                                       |       0.01 GB|
|   15|conv6_0    |convolutional           |     8192|     147584|relu5_0   |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8         |       0.04 GB|
|   16|relu6_0    |relu                    |     8192|          0|conv6_0   |                                                                       |       0.01 GB|
|   17|maxp3_0    |maxpooling              |     2048|          0|relu6_0   |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4           |       0.00 GB|
|   18|dropout3_0|dropout                  |     2048|          0|input_0   |RATE: 0.400000                                                         |       0.00 GB|
|   19|fc1_0      |fully connected         |      128|     262272|dropout3_0|inputs 2048, outputs 128                                               |       0.00 GB|
|   20|relu7_0    |relu                    |      128|          0|fc1_0     |                                                                       |       0.00 GB|
|   21|dropout4_0|dropout                  |      128|          0|input_0   |RATE: 0.500000                                                         |       0.00 GB|
|   22|fc2_0      |fully connected         |       10|       1290|dropout4_0|inputs 128, outputs 10                                                 |       0.00 GB|
|   23|softmax_0  |softmax                 |       10|          0|fc2_0     |                                                                       |       0.00 GB|
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                   |   258452|     550570|          |Memory (no layers):   0.00 GB                                          |       0.52 GB|
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.11: Topología VGG3-dropout-20-30-40-50.

```
|devices                                                                                 |
|----------------------------------------------------------------------------------------|
| mkl          | avx          | avx512       | opencl-fpga | opencl-gpu  | cublas       | clblas       |
|--------------|--------------|--------------|-------------|-------------|--------------|--------------|
|              |              |              |             |             | yes          |              |
|----------------------------------------------------------------------------------------|
| OMP: CPUs  13, dynamic   0                                                              |
| MPI: no - 1 processes                                                                   |
| Force synchronization (for cublas and clblas): yes                                      |
|----------------------------------------------------------------------------------------|

|------------------------------------------------------------------------------------------------------------------------------------------|
|network and model summary                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------|
|layer|name        |layer type             | neurons | params    | in_layer | configuration                                                   | memory      |
|-----|------------|-----------------------|---------|-----------|----------|-----------------------------------------------------------------|-------------|
|    0|input       |input layer            |     3072|          0|          |inputs 3072, outputs 3072                                        |     0.00 GB|
|    1|conv1_0     |convolutional          |    32768|        896|input_0   |IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32   |     0.04 GB|
|    2|relu1_0     |relu                   |    32768|          0|conv1_0   |                                                                 |     0.02 GB|
|    3|bn1_0       |batch normalization    |    32768|         64|relu1_0   |                                                                 |     0.10 GB|
|    4|conv2_0     |convolutional          |    32768|       9248|bn1_0     |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32  |     0.16 GB|
|    5|relu2_0     |relu                   |    32768|          0|conv2_0   |                                                                 |     0.02 GB|
|    6|bn2_0       |batch normalization    |    32768|         64|relu2_0   |                                                                 |     0.10 GB|
|    7|maxp1_0     |maxpooling             |     8192|          0|bn2_0     |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16    |     0.01 GB|
|    8|dropout1_0|dropout                  |     8192|          0|input_0   |RATE: 0.200000                                                   |     0.01 GB|
|    9|conv4 0     |convolutional          |    16384|      18496|dropout1 0|IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16 |     0.05 GB|
|   10|relu3_0     |relu                   |    16384|          0|conv4_0   |                                                                 |     0.01 GB|
|   11|bn3_0       |batch normalization    |    16384|        128|relu3_0   |                                                                 |     0.05 GB|
|   12|conv4_0     |convolutional          |    16384|      36928|bn3_0     |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16  |     0.08 GB|
|   13|relu4_0     |relu                   |    16384|          0|conv4_0   |                                                                 |     0.01 GB|
|   14|bn4_0       |batch normalization    |    16384|        128|relu4_0   |                                                                 |     0.05 GB|
|   15|maxp2_0     |maxpooling             |     4096|          0|bn4_0     |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8      |     0.00 GB|
|   16|dropout2_0|dropout                  |     4096|          0|input_0   |RATE: 0.300000                                                   |     0.00 GB|
|   17|conv5 0     |convolutional          |     8192|      73856|dropout2 0|IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8    |     0.02 GB|
|   18|relu5_0     |relu                   |     8192|          0|conv5_0   |                                                                 |     0.01 GB|
|   19|bn5_0       |batch normalization    |     8192|        256|relu5_0   |                                                                 |     0.03 GB|
|   20|conv6_0     |convolutional          |     8192|     147584|bn5_0     |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8    |     0.04 GB|
|   21|relu6_0     |relu                   |     8192|          0|conv6_0   |                                                                 |     0.01 GB|
|   22|bn6_0       |batch normalization    |     8192|        256|relu6_0   |                                                                 |     0.03 GB|
|   23|maxp3_0     |maxpooling             |     2048|          0|bn6_0     |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4      |     0.00 GB|
|   24|dropout3_0|dropout                  |     2048|          0|input_0   |RATE: 0.400000                                                   |     0.00 GB|
|   25|fc1 0       |fully connected        |      128|     262272|dropout3 0|inputs 2048, outputs 128                                        |     0.00 GB|
|   26|relu7_0     |relu                   |      128|          0|fc1_0     |                                                                 |     0.00 GB|
|   27|bn1_0       |batch normalization    |      128|        256|relu7_0   |                                                                 |     0.00 GB|
|   28|dropout4_0|dropout                  |      128|          0|input_0   |RATE: 0.500000                                                   |     0.00 GB|
|   29|fc2_0       |fully connected        |       10|       1290|dropout4_0|inputs 128, outputs 10                                          |     0.00 GB|
|   30|softmax_0 |softmax                  |       10|          0|fc2_0     |                                                                 |     0.00 GB|
|------------------------------------------------------------------------------------------------------------------------------------------|
|     |            |TOTAL                  |   373268|     551722|          |Memory (no layers):   0.00 GB                                    |     0.88 GB|
|------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura B.12: Topología VGG3-dropout-20-30-40-50-bn.