UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Optimización de cálculo con OpenCL para sistemas de entrenamiento de redes neuronales

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Stabile, Eugenio Bernabé

Tutor: Flich Cardo, José

Curso 2019-2020

Resum

El camp relacionat amb la intel·ligència artificial ha suposat una revolució tant en la indústria com en la societat, millorant la qualitat de vida i la productivitat del treball. Tanmateix, existeixen certes àrees en les quals els dispositius que utilitzen aquesta tecnologia necessiten un ús eficient dels seus components per a fer possible l'ús de la intel·ligència artificial en entorns pràctics.

Llavors, el motiu d'aquest treball final de grau és l'optimització relativa al cost computacional de les funcions prèviament creades en OpenCL per a donar suport als dispositius GPUs en l'aplicació HELENNA, la qual realitza l'entrenament i la inferència de xarxes neuronals. Com a resultat obtenim una reducció considerable del temps d'execució de les diferents xarxes neuronals tan densament connectades com convolucionals.

Paraules clau: OpenCL, Xarxes Neuronals, Optimització, GPU, Sistemes de Computació Heterogenis, Multiplicació de matrius

Resumen

El campo relacionado con la inteligencia artificial ha supuesto una revolución tanto en la industria como en la sociedad, mejorando la calidad de vida y la productividad del trabajo. Aun así, existen ciertas áreas en las cuales los dispositivos que utilizan esta tecnología necesitan de un uso eficiente de sus componentes para hacer posible el uso de la inteligencia artificial en entornos prácticos.

Por esto, el motivo de este trabajo final de grado es la optimización relativa al coste computacional de las funciones previamente creadas en OpenCL para dar soporte a los dispositivos GPUs en la aplicación HELENNA, la cual realiza el entrenamiento e inferencia de redes neuronales. Como resultado obtenemos una reducción considerable del tiempo de ejecución en las distintas redes neuronales tanto densamente conectadas como convolucionales.

Palabras clave: OpenCL, Redes Neuronales, Optimización, GPU, Sistemas de Computación Heterogéneos, Multiplicación de Matrices

Abstract

Artificial intelligence has been a revolution in both industry and society, improving quality of life and work productivity. But, there are certain areas in which devices that use this technology require efficient use of their components to make possible the use of artificial intelligence in real-world environments.

So, the motivator for this work is the optimization of the computational cost of functions previously created in OpenCL to support GPU devices in the HELENNA application. The HELENNA application performs training and inference on neural networks. As a result of these optimizations, we obtain a considerable reduction in execution time in both fully connected and convolutional neural networks.

Key words: OpenCL, Neural Networks, Optimization, GPU, Heterogeneous Computing, Matrix Multiplication

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 . . . 4
1.4 Estructura de la memoria . . . 5
2 OpenCL y AMD 7
2.1 OpenCL . . . 7
2.1.1 Modelo de programación . . . 9
2.1.2 Gestión de memoria . . . 13
2.1.3 Sincronización . . . 16
2.2 AMD . . . 17
2.2.1 Microarquitectura . . . 17
2.2.2 Unidades de cómputo . . . 18
2.2.3 Núcleo SIMD . . . 19
2.2.4 Planificador de la unidad de cómputo y el concepto de *wavefront*. . 19
3 Aplicación para el entrenamiento de redes neuronales HELENNA 23
3.1 Descripción de la Aplicación . . . 23
3.2 Soporte a las GPUs con OpenCL . . . 25
3.3 Optimización de cálculo con OpenCL . . . 26
3.3.1 Función *matmul* . . . 26
3.3.2 Especificaciones del dispositivo . . . 28
3.3.3 Técnicas de optimización . . . 29
3.3.4 Optimización mediante la utilización de la memoria local . . . 30
3.3.5 Optimización mediante el incremento de elementos por hilo . . . 36
3.3.6 Funciones optimizadas . . . 40
3.3.7 Soporte de creación, lectura y escritura de *buffers* . . . 40
3.4 Evaluación . . . 40
4 Conclusiones 47
4.1 Relación del trabajo desarrollado con los estudios cursados . . . 47
5 Trabajo futuro 49
5.1 Bloques en memoria local de tamaño rectangular . . . 49
5.2 Optimizaciones a funciones . . . 49
5.3 Soporte a las capas de *batch normalization* y *dropout* . . . 49
Bibliografía 51

Apéndice
A Topologías utilizadas en el entrenamiento de redes neuronales 53

Índice de figuras

1.1 Diferencias de núcleos de la CPU y la GPU. [1] . . . 2
1.2 Estructura de las redes neuronales profundas [2]. . . . 2
1.3 Diagrama de una neurona artificial perteneciente a una red neuronal [3]. . 3
1.4 Imagen en donde el sistema YOLO ha detectado objetos [4] . . . 3
1.5 Tiempo de ejecución entre las distintas GPUs y una CPU en la topología ResNet-18 (datos obtenidos de [6]). . . . 4

2.1 Algunos de los dispositivos en donde se puede ejecutar OpenCL. . . . 7
2.2 Rendimiento SGEMM en las librerías cuBLAS, clBLAS y OpenCL (myGEMM) [14]. . . . 8
2.3 Modelo de Programación en OpenCL para dispositivos fabricados por AMD [16] . . . 9
2.4 *Command queue* [16]. . . . 11
2.5 Relación de conceptos entre los elementos utilizados por OpenCL y su distribución en un dispositivo físico [16]. . . . 12
2.6 Identificaciones locales y globales de un elemento de trabajo [19] . . . 13
2.7 Diagrama de distribución espacial de las diferentes memorias en un dispositivo GPU de AMD y sistema principal [16]. . . . 14
2.8 Conflictos en el banco de memoria [21]. . . . 15
2.9 Barrera a nivel de grupo de trabajo [22] . . . 16
2.10 Arquitectura de *Polaris* 10 [23]. . . . 18
2.11 Arquitectura Unidad de cómputo en *Polaris* 10 [23]. . . . 19
2.12 Diagrama de un núcleo SIMD [24] . . . 19
2.13 Unidad de cálculo en espera debido a dependencia de datos [16]. . . . 20
2.14 Ejecución de los distintos *wavefronts* en una unidad de cómputo [16]. . . . 21
2.15 Relación entre los conceptos de *wavefront* con los grupos y elementos de trabajos en una matriz tridimensional [16]. . . . 21

3.1 Imágenes pertenecientes al conjunto MNIST [30] . . . 24
3.2 Imágenes pertenecientes al conjunto CIFAR10 [31] . . . 25
3.3 Coste en porcentaje de la ejecución de la clasificación MNIST en HELENNA 27
3.4 Ilustración de la función *matmul* [14] . . . 29
3.5 Diagrama ejecución *matmul* con memoria local [14] . . . 31
3.6 Diagrama de la obtención de un elemento en las submatrices [14] . . . 34
3.7 Tiempo de ejecución con diferentes valores de tamaño de bloque . . . 34
3.8 Comparación tiempo de ejecución . . . 35
3.9 Tiempo de ejecución con diferentes valores de trabajo por hilo . . . 39
3.10 Comparación tiempo de ejecución entre las optimizaciones con memoria local e incremento de trabajo por hilo . . . 39
3.11 Comparación tiempo de ejecución entre la versión en OpenCL y la versión final de su optimización . . . 41
3.12 Comparación del porcentaje de precisión entre la versión en OpenCL y la versión final de su optimización . . . 42
3.13 Aceleración obtenida entre las versiones base y optimizada de OpenCL . . 43

3.14 Comparación del tiempo de ejecución entre las funciones optimizadas y sus versiones base en la topología *MNIST-Big* 44
3.15 Comparación del tiempo de ejecución entre las funciones optimizadas y sus versiones base en la topología *MNIST-Conv* 44
3.16 Comparación del tiempo de ejecución entre las optimización a mano y clBLAS en las topologías que utilizan el conjunto de datos MNIST 45
3.17 Comparación del tiempo de ejecución entre las optimización a mano y clBLAS en las topologías que utilizan el conjunto de datos CIFAR 45
3.18 Comparación de la precisión entre las optimización a mano y clBLAS en las topologías anteriormente ejecutadas 46

A.1 Configuración utilizada en las topologías 53
A.2 Topología MNIST-Medium 54
A.3 Topología MNIST-Big 54
A.4 Topología MNIST-Large 54
A.5 Topología CIFAR10-VGG1 55
A.6 Topología CIFAR10-VGG2 56
A.7 Topología CIFAR10-VGG3 57

Índice de tablas

3.1 Funciones implementadas (parte I). 25
3.2 Funciones implementadas (parte II). 26
3.3 Valores con los que se ejecutan los parámetros para el entrenamiento de redes neuronales en HELLENA. 27
3.4 Especificaciones AMD RX 480. 29
3.5 Especificaciones AMD RX 480 con la herramienta clinfo. 30
3.6 Funciones optimizadas e implementadas. 40
3.7 Funciones implementadas relacionadas con el soporte de *buffers*. 41

CAPÍTULO 1
Introducción

1.1 Sistemas de computación heterogéneos

Hoy en día, el crecimiento de tecnologías como el *deep learning* nos da la posibilidad de utilizar diferentes tipos de datos como textos, imágenes, el habla, entre otros. Esta variedad de información implica una fuente de entrada de datos heterogénea, la cual necesitamos procesar utilizando los diferentes dispositivos que empleen formas de computo que se adaptan de una forma eficiente tanto en rendimiento como de forma energética. Los sistemas integrados por dos o más de estos componentes tales como CPU (*Central Processing Unit*), FPGA (*Field Programmable Gate Array*), GPU (*Graphical Processing Unit*), ASIC (*Application-Specific Integrated Circuit*), entre otros, se les da el nombre de sistemas de computación heterogéneos. Esta forma de computación separa y reparte los datos a los dispositivos de cálculo teniendo en cuenta el tipo de la información y la capacidad de la unidad para procesarlo eficientemente.

Uno de los sistemas heterogéneos actualmente más utilizados son los compuestos por los dispositivos CPU y GPU, debido a que realizan un conjunto de tareas diferentes proporcionando así un mayor rendimiento computacional y eficiencia energética. Esta agrupación permite usar eficientemente numerosas aplicaciones de cálculo intensivo relacionado al entrenamiento de redes redes neuronales.

La mejora de rendimiento y eficiencia en este sistema se debe al tipo de trabajo en el cual se enfocan estas unidades. Por un lado tenemos las GPUs, creadas para el procesamiento de gráficos en tres dimensiones y cálculos matemáticos con un elevado grado de paralelismo gracias a que posee una cantidad de miles de núcleos, ver figura 1.1. Sin embargo la frecuencia de reloj, la simplicidad de su conjunto de instrucciones y el tamaño de memoria *cache* utilizados en este dispositivo, hacen que estrategias como la fuerza bruta para el procesamiento de datos en paralelo sean claves para su rendimiento. De esta forma, podemos asignar a esta unidad la realización de cálculos matemáticos con un conjunto de datos considerablemente grande para aprovechar sus características de la forma más eficiente posible.

Por otro lado tenemos las CPUs, que a diferencia de las GPUs, estas poseen solamente decenas de núcleos que cuentan con un juego de instrucciones más complejo, mayor tamaño de memoria *cache* y una frecuencia mayor. De esta manera, esta unidad es destinada a realizar acciones en los que se procesan instrucciones complejas como los controles de flujo, cálculos matemáticos complejos como las funciones trascendentales y aplicaciones donde se emplea el mayor tiempo de la ejecución en transferencias de datos.

Como podemos ver, la diferencia entre ambas unidades son notables tanto a nivel de arquitectura como de enfoque de trabajo. Es por ello que la utilización de estos disposi-

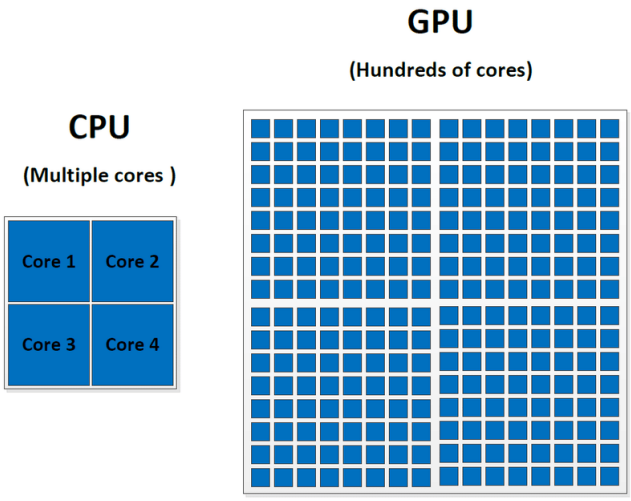

Figura 1.1: Diferencias de núcleos de la CPU y la GPU. [1]

tivos en un sistema heterogéneo, nos proporciona las mejores características de la CPU y GPU obteniendo mejoras de eficiencia tanto en tiempo como en coste energético que si tuviéramos que utilizar solo un dispositivo.

Como hemos visto anteriormente, la utilización de los sistemas heterogéneos para el desarrollo de las redes neuronales profundas (DNN del inglés *Deep Neural Network*) (figura 1.2) resultan ser eficientes debido a la diversidad de los conjuntos de datos empleados en su entrenamiento. Debemos aclarar que estas redes son un tipo de red neuronal artificial, las cuales están formadas por múltiples capas interconectadas en donde se hallan dichas neuronas (ver figura 1.3) para la consecución de resolver los problemas del mundo real.

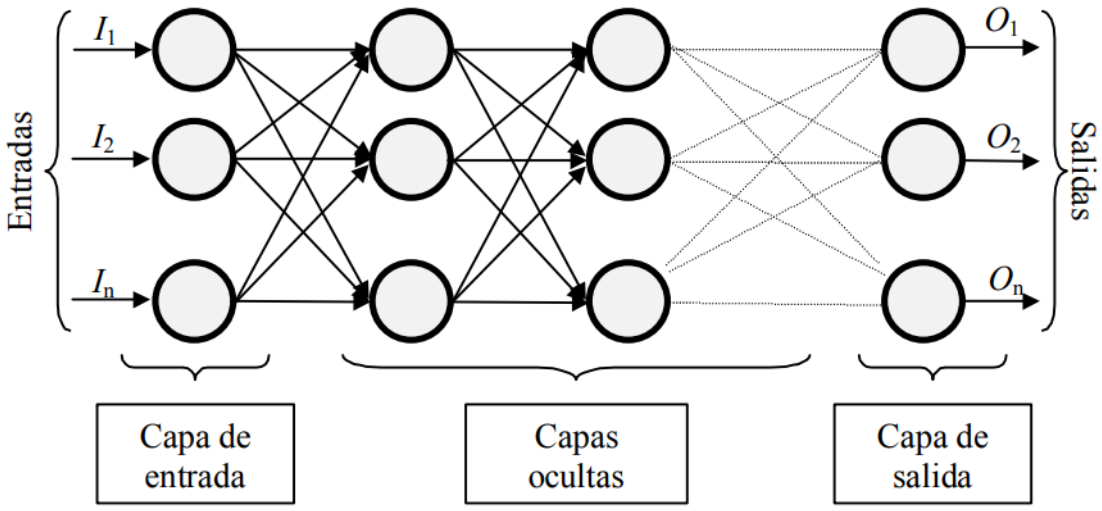

Figura 1.2: Estructura de las redes neuronales profundas [2].

La utilización de la GPU en la mejora de rendimiento en el entrenamiento de las redes neuronales se basa en la operación de grandes cantidades de información proveniente de los datos de entrada a la red. Mientras que las funciones de transferencia, dependiendo de su complejidad, pueden ser ejecutadas tanto en CPU (funciones trascendentales), como en GPU. Por último debemos destacar que en el presente trabajo realizaremos la

implementación y optimización en GPU del cálculo para el entrenamiento de redes neuronales.

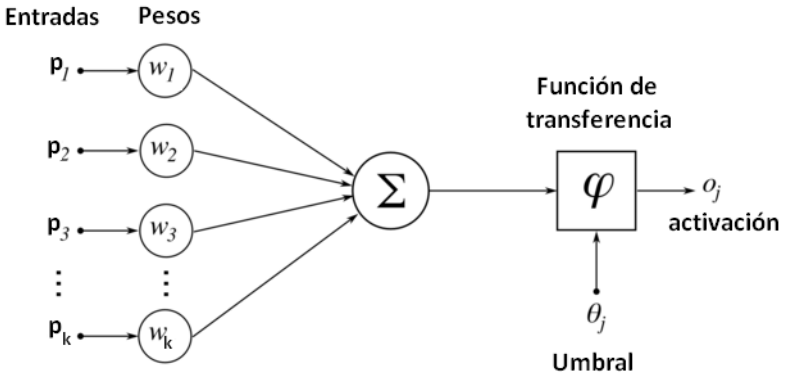

Figura 1.3: Diagrama de una neurona artificial perteneciente a una red neuronal [3].

Ejemplificaremos a continuación, el valor que tienen las redes neuronales para la sociedad como es el uso de la detección de objetos en tiempo real. Esta técnica en los últimos años fue adquiriendo importancia con la incorporación de las redes neuronales convolucionales (CNN) y el aprendizaje profundo. Sistemas como YOLO (*You Only Look Once*), (ver imagen 1.4) o R-CNN (*Region-Based Convolutional Neural Networks*) ayudan a la realización de tareas como la detección de objetos en vehículos autónomos o sistemas de vigilancia en tiempo real.

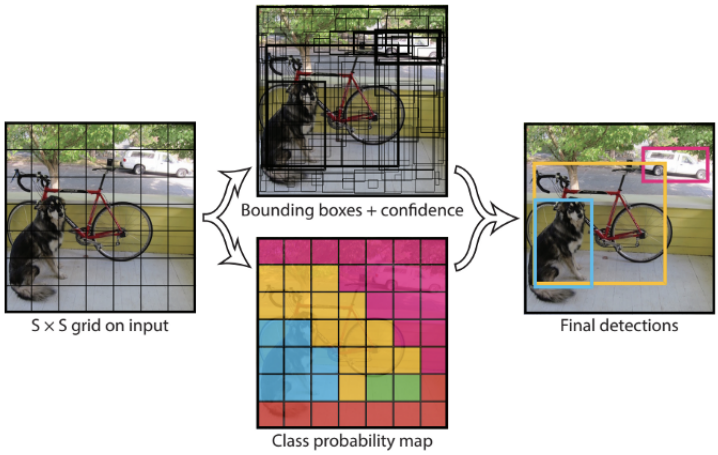

Figura 1.4: Imagen en donde el sistema YOLO ha detectado objetos [4]

1.2 Motivación

En la última década, el incremento de aplicaciones que utilizan las redes neuronales ha sido notorio. Esto se debe en parte a los avances en este ámbito de nuevas arquitecturas de redes neuronales, como por ejemplo Alexnet [5], que han ido obteniendo una mejora en la tasa de acierto sobre la detección de objetos en imágenes. Otro de los motivos que han hecho posible la masificación de la inteligencia artificial ha sido el uso a la GPU como dispositivo para realizar el entrenamiento de las redes neuronales (ver figura 1.5), ya que anteriormente se utilizaban las CPU para este fin. Este hecho a su vez estuvo influenciado por los avances en la potencia de cálculo a lo largo de los años en donde un incremento del limite teórico de GFLOPS por parte de las GPUs y la implementación de

librerías y *software* como TensorFlow, Keras, cuDNN, clBLAS, cuBLAS, entre otros, han permitido sacar el máximo rendimiento de estos dispositivos reduciendo considerablemente el tiempo de ejecución.

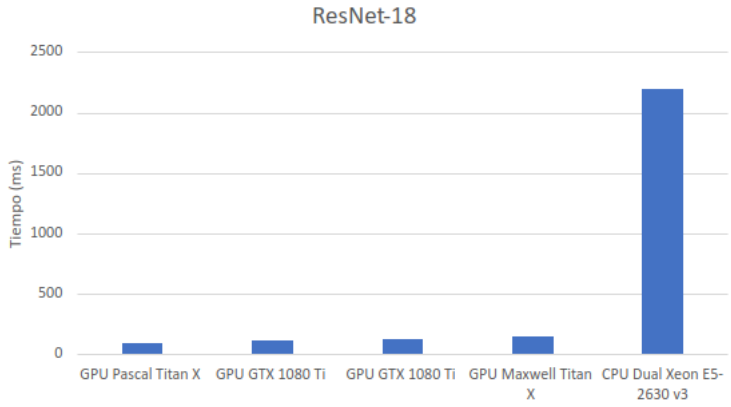

Figura 1.5: Tiempo de ejecución entre las distintas GPUs y una CPU en la topología ResNet-18 (datos obtenidos de [6]).

Desarrollar nuevas herramientas para el uso de la inteligencia artificial, ya sean para ámbitos con fines educativos o de investigación, requiere de una integración de estas tecnologías y arquitecturas. Este es el caso de la aplicación HELENNA, desarrollada para el entrenamiento e inferencia de redes neuronales en arquitecturas heterogéneas para servir como herramienta en trabajos relacionados con la docencia, así como también, en investigación.

1.3 Objetivos

El siguiente trabajo tenemos como objetivo dar soporte al uso de arquitecturas GPUs en la aplicación HELENNA en el modelo de programación OpenCL, así como también de la optimización de las funciones de cálculo presentes en el programa para el entrenamiento de redes neuronales en el dispositivo a emplear. Cabe destacar que en el enfoque de este proyecto está en las técnicas de optimización y en el ajuste de la aplicación a la arquitectura que se vaya a utilizar para su implementación en dispositivos cuyas características impidan el uso adecuado de las librerías de cálculo de álgebra lineal como clBLAS, así como también de aquellos que no puedan utilizar el modelo de programación CUDA, es decir productos del fabricante NVIDIA. Siendo de esta manera, las técnicas de optimización una solución heterogénea a los diversos dispositivos que cumplen con lo detallado previamente y el ajuste de la aplicación a la arquitectura un ajuste adherente a la unidad a emplear.

Mientras que en el contexto de la aplicación, tendremos que tener en cuenta aspectos como el diseño de funciones para dispositivos que realizan cómputo de forma paralela o la optimización de estas funciones mediante técnicas que usen de forma eficiente los recursos del dispositivo utilizado, teniendo en consideración su arquitectura y modelo de ejecución. En resumen, los objetivos del presente trabajo lo podemos listar en los subjetivos siguientes:

- Soporte del modelo de programación OpenCL para las arquitecturas GPUs no pertenecientes al fabricante NVIDIA en HELENNA.
- Soporte a las capas convolucionales en la aplicación HELENNA con OpenCL para las GPUs.

- Soporte a las capas densamente conectadas en la aplicación HELENNA con OpenCL para las GPUs.
- Optimización de las funciones de mayor carga computacional utilizadas en las capas anteriormente mencionadas en HELENNA.

1.4 Estructura de la memoria

Este trabajo final de grado se compone de cinco capítulos:

- Introducción: Explicamos en este apartado la situación actual de los sistemas de computación heterogéneos y las redes neuronales, así como también la motivación para realizar este trabajo y los objetivos a conseguir y la motivación para llevarlo a cabo.
- OpenCL y AMD: en este apartado nos centramos en las tecnologías utilizadas en este trabajo. Vemos las características principales de OpenCL, sus diferencias con los demás *frameworks* para entornos que empleen computación en paralelo, su modelo de programación, sincronización y gestión de memoria. Mientras que en la parte de AMD, explicamos con detalle el dispositivo empleado en este proyecto, una tarjeta gráfica AMD RX 480 de la cual se detalla la microarquitectura que utiliza y el modelo de ejecución empleado por este componente.
- Aplicación para el entrenamiento de redes neuronales (HELENNA): en el tercer capítulo describimos la aplicación HELENNA, la aportación realizada tanto en el soporte como en la optimización posterior y evaluamos los resultados conseguidos.
- Conclusiones: en este apartado explicamos los resultados conseguidos en este proyecto. Además se expone la relación del trabajo desarrollado con respecto a los estudios cursados.
- Trabajo futuro: en este último capitulo, se dan a conocer las mejoras a futuro que podemos desarrollar en la aplicación HELENNA.

CAPÍTULO 2
OpenCL y AMD

Para realizar el siguiente trabajo, se ha necesitado del lenguaje de programación e interfaz de programación de aplicaciones OpenCL, el cual en la siguiente sección se describe su funcionamiento y las razones que han llevado a optar por este modelo de programación. Adicionalmente, para este proyecto se ha utilizado la tarjeta gráfica AMD RX 480 la cual detallaremos posteriormente su arquitectura y modelo de ejecución.

2.1 OpenCL

OpenCL (*Open Computing Language*) fue creado en un trabajo conjunto entre Apple, AMD, IBM, Intel y NVIDIA [7] y es un entorno de trabajo para el desarrollo de aplicaciones con paralelismo a nivel de tareas y de datos. Además, consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Este *framework* está enfocado para ser ejecutado en unidades centrales de procesamiento (CPU), matriz de puertas lógicas programable en campo (FPGA), unidades de procesamiento gráfico (GPU) así como también para sistemas embebidos entre otros. Esto nos permite desarrollar para diferentes plataformas heterogéneas, en donde la arquitectura y/o el dispositivo a programar tengan el soporte de esta herramienta a pesar de las características propias de cada componente. Actualmente es un estándar abierto y libre de derechos gestionado por el Grupo Khronos [8].

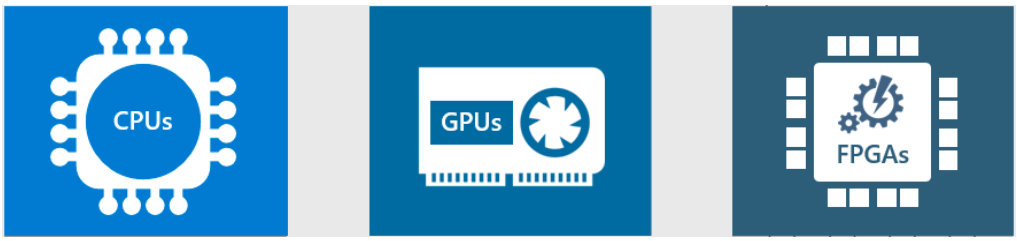

Figura 2.1: Algunos de los dispositivos en donde se puede ejecutar OpenCL.

Al ser OpenCL un estándar abierto, cada fabricante implementa su propia versión de los controladores para el dispositivo, así como también aplicaciones para el desarrollo en este modelo de programación. Por ejemplo, AMD ofrece la plataforma de desarrollo ROCm, dedicada a la computación de alto rendimiento (HPC) [9]. En ella podemos encontrar los controladores para sus dispositivos que brindan soporte a OpenCL, así como documentación, librerías y herramientas para el desarrollo en este *framework*. AMD ROCm Profiler [10] para el análisis de rendimiento en GPU, AMD ROCm Debugger [11] para la depuración a bajo nivel en los dispositivos GPU de la compañía, entre otros, son ejemplos claros de estas herramientas. Intel por su parte, ofrece soporte para OpenCL de

la misma forma que AMD con su plataforma *Intel Developer Zone* [12], con la diferencia de que incluye herramientas para el desarrollo en FPGAs.

Debemos mencionar que OpenCL no es la única plataforma de programación para sistemas dedicados a la computación de alto rendimiento. A continuación, se describen las diferentes alternativas y se explican la elección de OpenCL para este trabajo.

CUDA (*Compute Unified Device Architecture*), es el principal competidor en este ámbito. Creada por NVIDIA, esta plataforma se centra en aprovechar las características multinúcleo y altamente paralelizable de las GPUs desarrolladas por esta misma compañía para acelerar el cómputo de funciones de cálculo. En términos de herramientas y recursos para el desarrollo en su plataforma, NVIDIA proporciona en su página dedicada a CUDA documentación al respecto del lenguaje así como también de sus librerías [13]. Además, posee aplicaciones como CUDA-GDB, un depurador multihilo capaz de administrar miles de subprocesos que se ejecutan simultáneamente en cada GPU del sistema o como Nsight, un entorno de desarrollo integrado en Microsoft Visual Studio para CUDA y aplicaciones gráficas que se utilizan en las GPUs de NVIDIA.

En cuanto al rendimiento comparado con OpenCL, CUDA ofrece un mayor desempeño, prueba de esto se puede visualizar en la figura 2.2 perteneciente al tutorial *OpenCL SGEMM tuning for Kepler* [14]. En ella podemos ver como se obtienen unos resultados claramente superiores para la plataforma creada por NVIDIA. Debemos mencionar que para toma de muestras de esta figura se ha utilizado la tarjeta gráfica NVIDIA Tesla K40M y se han escogido las librerías para el cálculo de álgebra lineal cuBLAS y clBLAS correspondientes a CUDA y OpenCL respectivamente. Mientras que la función que se ha empleado para obtener el resultado corresponde a una multiplicación de matrices con un tamaño variable cuyos elementos son del tipo coma flotante de simple precisión (SGEMM). Por último, tanto los datos ofrecidos por clBlas *tuned* (modificación de clBlas) y myGEMM (optimización a mano sin emplear librerías) han sido desarrollas en OpenCL, con lo cual, podemos decir que ninguna de las opciones basadas en este *framework* alcanzan en rendimiento a la solución desarrollada en CUDA.

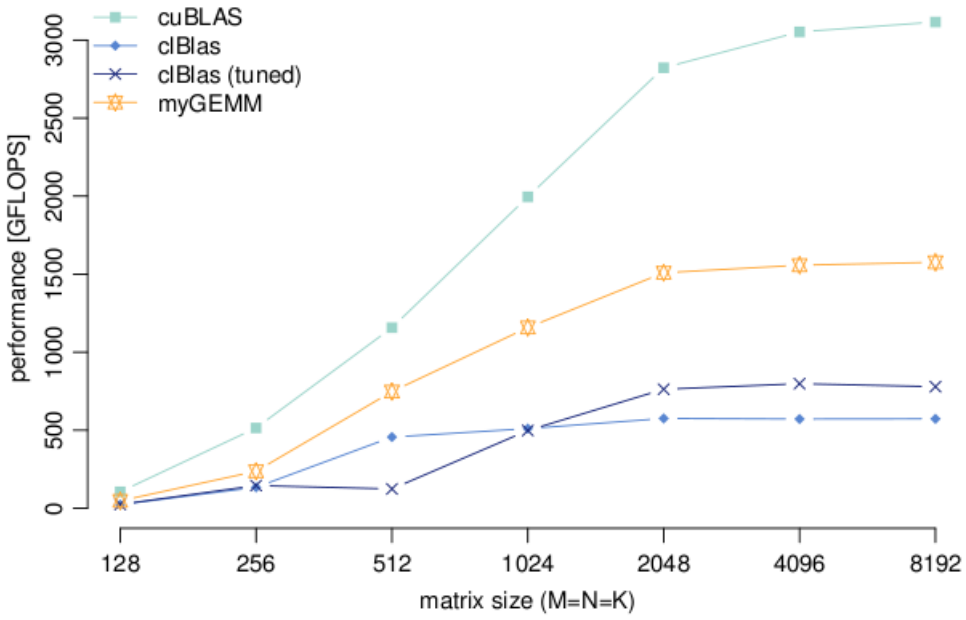

Figura 2.2: Rendimiento SGEMM en las librerías cuBLAS, clBLAS y OpenCL (myGEMM) [14].

A pesar de ello, la principal desventaja de CUDA es que solo se puede utilizar en las GPUs fabricadas por la compañía NVIDIA. Por lo tanto, si queremos una aplicación

que pueda ser ejecutada en la mayor cantidad de dispositivos posibles, debemos elegir OpenCL para llevar a cabo el trabajo.

Como hemos podido ver en la figura 2.2, existen librerías que ofrecen implementaciones de cálculo de álgebra lineal en OpenCL, utilizadas para el entrenamiento de redes neuronales, podemos destacar a clBLAS [15] por ser una de las más conocidas. Esta librería se crea para facilitar a los desarrolladores el uso de una manera sencilla y eficiente de las funciones de cálculo de álgebra lineal en OpenCL sin necesidad de escribir y optimizar el código *kernel* a mano. Otorgando de esta manera un rendimiento mayor que una implementación sencilla de estas funciones.

Sin embargo, clBLAS ha sido diseñada y desarrollada teniendo en mente las arquitecturas de las GPUs. De esta forma, su uso para en los diversos sistemas heterogéneos está determinada por el fabricante del dispositivo en cuestión, llegando a casos en los que su implementación pueda no ser óptima o incluso no tenga soporte, como en varios sistemas embebidos y FPGAs. Debido a que buscamos optimizar el cálculo para sistemas de entrenamiento de redes neuronales en una aplicación enfocada a diferentes arquitecturas heterogéneas, elegiremos la optimización a mano mediante técnicas que puedan ser empleadas en los diversos sistemas heterogéneos.

2.1.1. Modelo de programación

Para desarrollar código en OpenCL debemos tener en cuenta que es un estándar abierto para sistemas heterogéneos. Esto implica que se ha de especificar en que dispositivo trabajaremos y configurarlo antes de programar debido a que cada fabricante tiene su propia implementación particular de este *framework*.

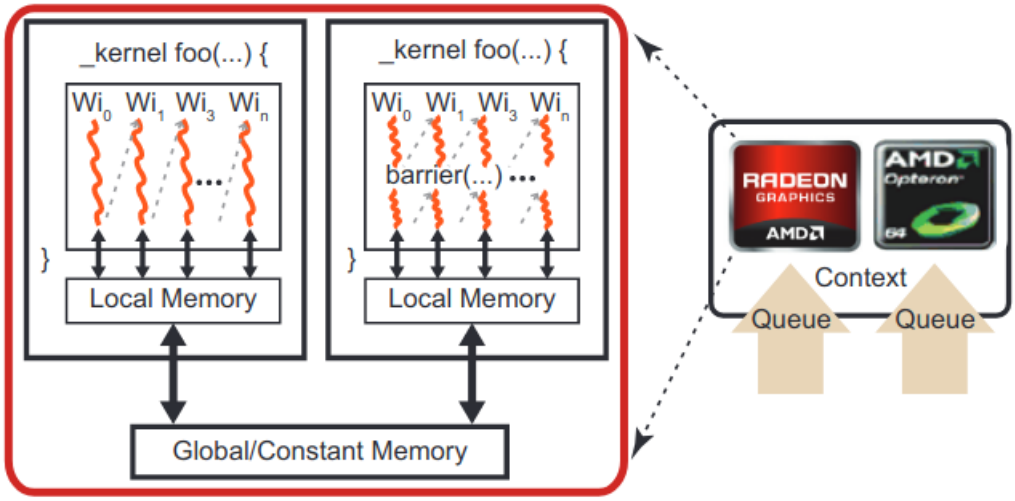

Figura 2.3: Modelo de Programación en OpenCL para dispositivos fabricados por AMD [16]

A continuación, introducimos los principales conceptos en OpenCL para entender posteriormente el desarrollo del problema.

Plataforma

Una plataforma OpenCL es un concepto relacionado con los dispositivos, ya que cada fabricante realiza una implementación específica de OpenCL. Un programa OpenCL puede funcionar con múltiples dispositivos en diferentes plataformas. Para ello primero

consulta y selecciona una plataforma, una vez elegida, se crea un contexto y luego se utiliza para crear *kernels*, *command queues* y se almacenan en búferes del dispositivo objetos de memoria OpenCL.

Contexto

Un contexto oculta los detalles de bajo nivel de los diferentes dispositivos de cálculo y proporciona una interfaz consistente para que el programa OpenCL interactúe con los diferentes dispositivos. Un contexto se puede crear para uno o más dispositivos de cálculo en una plataforma específica, una vez que lo tengamos inicializado, se puede realizar lo siguiente:

- Crear una o más *Command queue*.
- Crear programas OpenCL para ejecutarse en uno o más dispositivos asociados.
- Crear *kernels* dentro de esos programas.
- Asignar búferes de memoria a los dispositivos.
- Escribir datos en el dispositivo.
- Enviar el *kernel* (con los argumentos adecuados) a la cola de comandos para su ejecución.
- Leer los datos del dispositivo al *host*.

Command queue

Para operar con objetos de memoria u otros elementos que estén dentro de un programa o *kernel* de OpenCL debemos utilizar un *command queue*. Desde el sistema, se envían comandos como la ejecución del *kernel* o la transferencia de datos a través de la cola para calcular estos objetos en el dispositivo asociado. Las instrucciones de un *command queue* son ejecutadas por el dispositivo dependiendo del modo establecido durante la creación del comando, ya sea en el mismo orden en que entran en la cola, es decir empleando un método FIFO [17], o fuera de orden ejecutando según estén disponibles los recursos [18]. Un *command queue* solo puede asociarse a un contexto y a un dispositivo de cálculo, pero este último puede tener múltiples colas, una para la transferencia de datos y otra para la ejecución del *kernel*. Por último, podemos clasificar los diferentes comandos que integran *command queue* en tres categorías:

- Comandos pertenecientes al *kernel* como por ejemplo *clEnqueueNDRangeKernel()*.
- Comandos de memoria como *clEnqueueWriteBuffer()*, entre otros.
- Comandos para los eventos, por ejemplo *clWaitForEvents()*.

Como se ilustra en la Figura 2.4, una aplicación puede generar múltiples *command queue* que se añaden posteriormente a una cola del dispositivo seleccionado. Como podemos ver, los *command queues* 1 y 3 se fusionan dentro de la cola de la CPU (flechas negras), mientras que el *command queue* 2 se añade en la cola correspondiente a la GPU (flecha de color rojo). Una vez realizada esta acción, la cola de cada dispositivo programa el trabajo repartiéndolos a los recursos de cómputo que dispone.

Para finalizar, en la figura 2.4 las siguientes letras representan a los siguientes comandos:

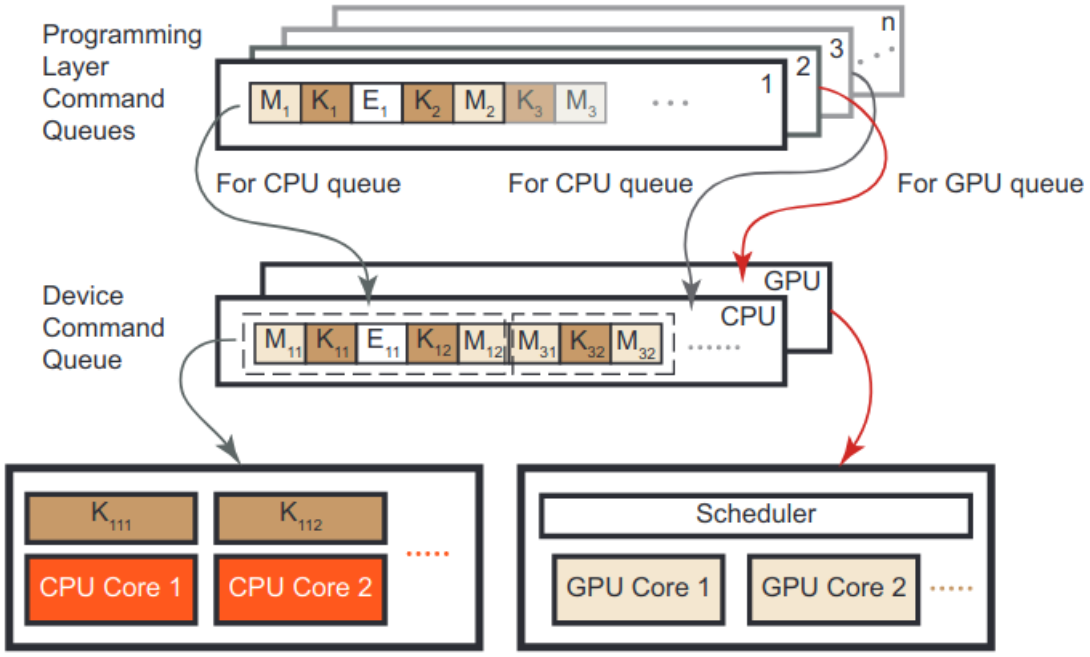

Figura 2.4: *Command queue* [16].

- *K* = comandos del *kernel*.
- *M* = comandos de memoria.
- *E* = comandos de eventos.

Objeto del programa

Un objeto del programa, en OpenCL, es una colección de una o más funciones del *kernel*. Este objeto está construido para un contexto y un dispositivo específicos y puede ser creado de dos formas. La primera es escribiendo su código fuente en C el cual posteriormente será pasado como un búfer de texto para crear el programa objeto mediante la llamada a función de *clCreateProgramWithSource* en tiempo de ejecución. La segunda es mediante archivos binarios y posteriormente en tiempo de ejecución realizando la llamada a *clCreateProgramWithBinary*.

Kernel

Un *kernel* es una pequeña unidad de ejecución que realiza una función claramente definida y que puede ser ejecutada en paralelo. Se identifica con la etiqueta *__kernel* y para ejecutarse necesita, en primer lugar, estar creada dentro de un objeto del programa y en segundo lugar le ha de asignar la cola de comandos la dimensión del problema a ejecutar así como también el tamaño de este.

NDRange

NDRange es un espacio de índice de N dimensiones al que se asignan a los elementos de trabajo un número de índice. El *kernel* indica al dispositivo cuantos elementos de tra-

bajo utilizará así como también la dimensión en la que se va a ejecutar teniendo en cuenta el máximo de dimensiones soportado por el *hardware*.

Grupo de trabajo

Un *Work group* o grupo de trabajo es una agrupación de elementos de trabajo y se asocia, cada grupo, con una unidad de cálculo solamente (ver imagen 2.5). Aunque, una unidad puede contener múltiples grupos de trabajo. En cuanto al tamaño del grupo de trabajo, esta se establece a la hora de ejecutar el *kernel*, la agrupación se puede establecer de manera explícita, insertando el número total de elementos de trabajo y el tamaño del grupo de trabajo. O se puede especificar de manera implícita, dejando que OpenCL los divida en grupos de trabajo.

Para establecer el tamaño del *work group*, hemos de tener en cuenta la memoria local de la unidad de cómputo, así como también el número máximo de elementos que puede operar el dispositivo asociado. Una vez asignado el grupo a una unidad de cómputo, un elemento de esta agrupación puede escribir en cualquier lugar de la memoria local de la unidad. Los datos sólo pueden compartirse entre los elementos del grupo de trabajo perteneciente.

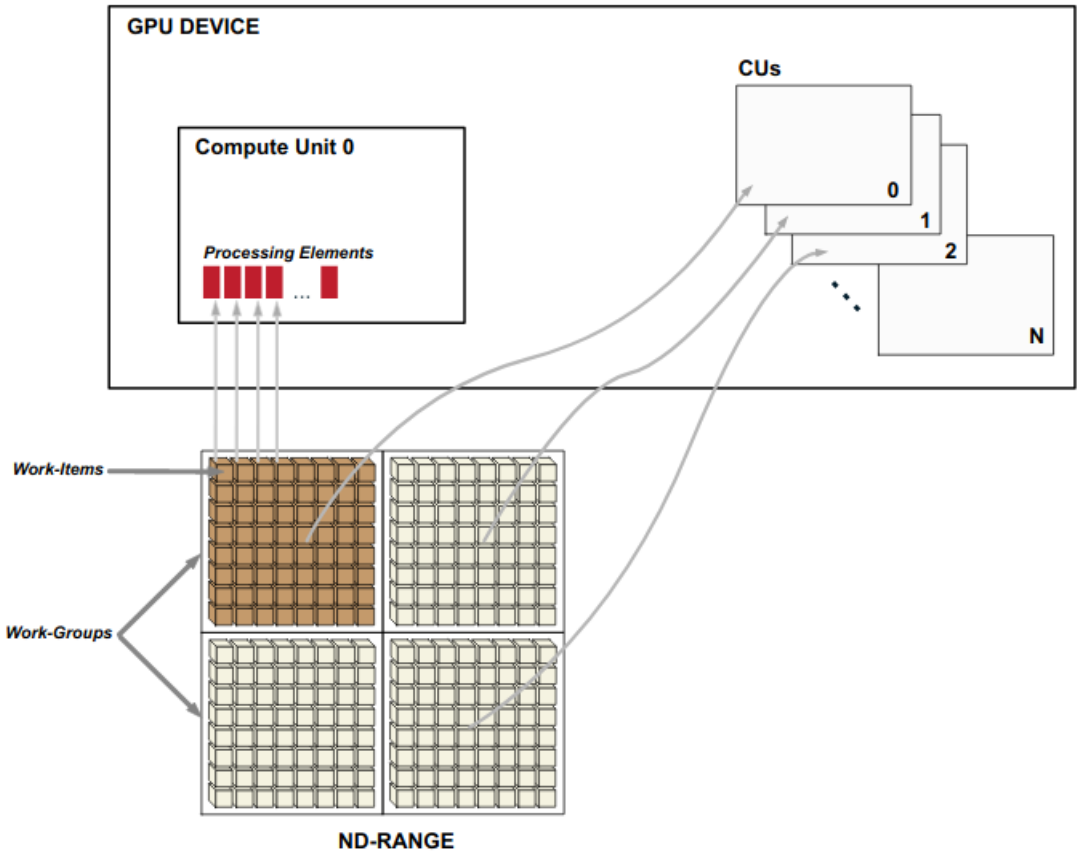

Figura 2.5: Relación de conceptos entre los elementos utilizados por OpenCL y su distribución en un dispositivo físico [16].

Elemento de trabajo

Un *work item* es un elemento perteneciente a una colección de ejecuciones paralelas en un dispositivo que ha sido invocado a través de un comando. Este elemento de trabajo

es ejecutado por uno o más elementos de procesamiento como parte de un grupo de trabajo asignado en una unidad de cómputo. Para distinguirse de otras colecciones de ejecución, cada elemento posee su identificador global y su identificador local creadas por el *NDRange* en las diferentes dimensiones en la cual ha sido instanciada, como podemos ver en la imagen 2.6.

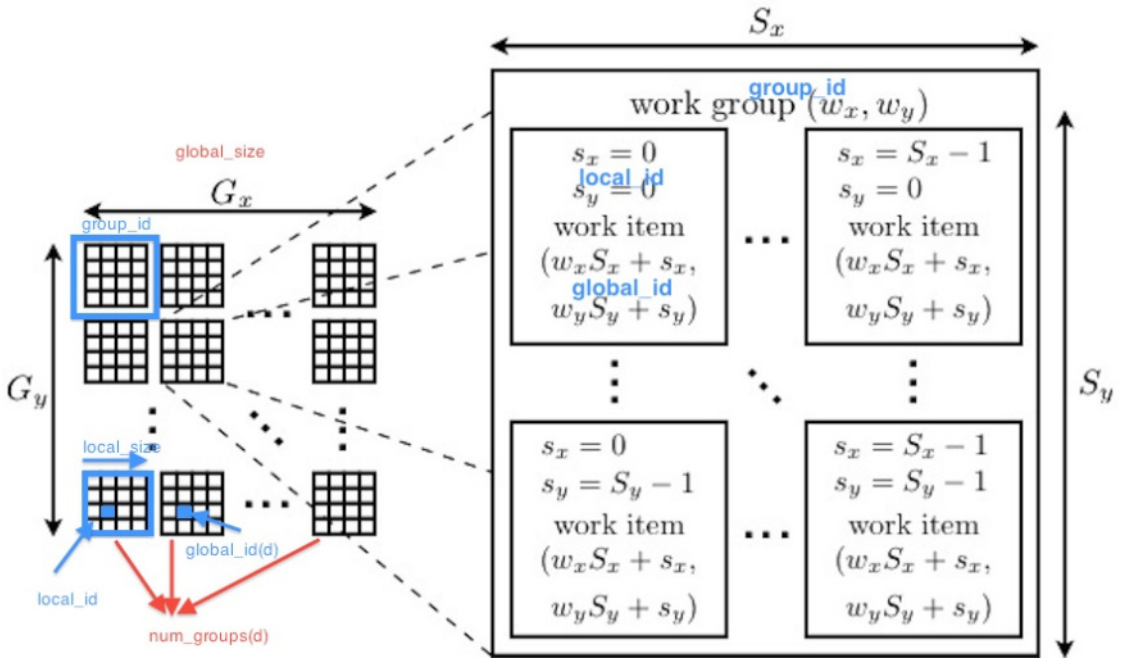

Figura 2.6: Identificaciones locales y globales de un elemento de trabajo [19]

Objetos búfer

Un objeto en memoria que almacena una colección lineal de bytes. Los objetos que están en búfer son accesibles usando un puntero en un *kernel* ejecutado en un dispositivo y pueden ser manipulados por el programador usando llamadas de la API de OpenCL. Un objeto del búfer encapsula la siguiente información:

- Tamaño en bytes.
- Propiedades que describen cual será su uso.
- Datos del búfer.

2.1.2. Gestión de memoria

Como podemos ver en la imagen 2.7, podemos ver la distribución de las diferentes memoria que componen un dispositivo GPU, estas las detallaremos a continuación.

Memoria global

La memoria global es una región de memoria accesible para la lectura y/o escritura a todos los elementos de trabajo ejecutados en un contexto. Un programa del *kernel* puede asignar una variable en la memoria global usando la etiqueta (*__global*). Esta memoria es un orden de magnitud más rápida que la memoria del del sistema [16].

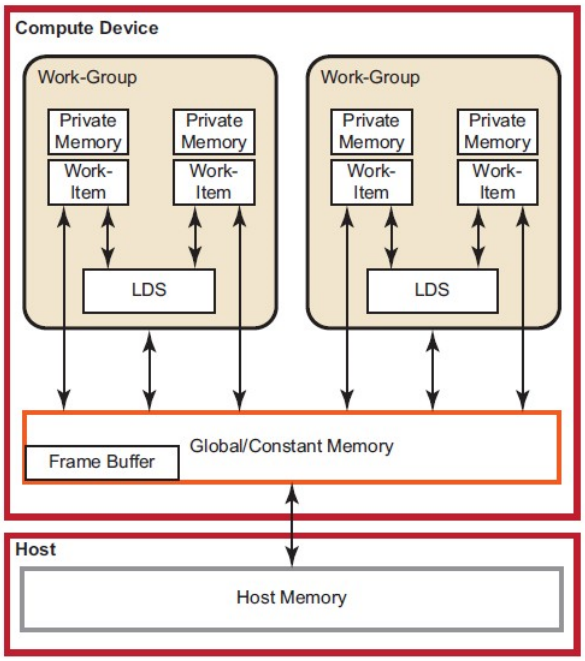

Figura 2.7: Diagrama de distribución espacial de las diferentes memorias en un dispositivo GPU de AMD y sistema principal [16].

Memoria constante

Una región de la memoria global de solo lectura que permanece constante durante la ejecución del *kernel*. Es posible asignar los objetos de memoria colocados en la memoria constante utilizando la etiqueta (*__constant* o *const*).

Memoria del sistema

Normalmente la memoria principal del sistema, de ella se obtienen los datos que utilizaremos en el dispositivo sobre el cual realizaremos el cálculo de las operaciones. Para realizar la copia a dicho dispositivo, OpenCL ofrece los siguientes métodos:

- Con la función *clEnqueueMapBuffer()* envía una solicitud para asignar un objeto del búfer a la memoria del sistema. Esto implica que cualquier cambio realizado en el espacio de memoria del sistema mapeado se copiará en el objeto del búfer.
- *clEnqueueWriteBuffer()*: Indica a la *command queue*, un comando que realizará la escritura de datos indicados de la memoria principal a la memoria global del dispositivo.
- *clCreateBuffer()*: con el *flag CL_MEM_HOST_PTR* indica que se quiere realizar la copia del dato seleccionado en la memoria global del dispositivo.

Para obtener los datos de salida, estos se copian de la memoria del dispositivo a la memoria principal usando la función *clEnqueueReadBuffer()*.

Memoria local

Cada unidad de cálculo tiene su propio almacenamiento de datos local de alta velocidad y baja latencia (LDS). Los datos almacenados en el LDS pueden ser compartidos por

todos los elementos de trabajo dentro de un grupo de trabajo. Un programa del *kernel* puede asignar una variable o un objeto de búfer en la memoria local utilizando la etiqueta (__local). Esta memoria es un orden de magnitud más rápida que la memoria global, pero más lenta que la memoria privada.

Esta memoria local a nivel físico, en la microarquitectura GCN que veremos posteriormente, está dividida en 32 bancos de memoria y cada una de ellas situadas cerca de las ALUs que ejecutan los cálculos. Individualmente, estos bancos pueden ser utilizados por un conjunto de datos a la vez para operaciones de lectura y escritura, pero en caso de que existan dos o más intentos de acceso al mismo banco por distintos conjuntos, ocurrirá un conflicto de banco [20]. Esto produce que las *wavefronts* que quieran obtener el recurso, definición que veremos en el apartado de AMD, estén a la espera de la resolución de los conflictos, accediendo de forma secuencial al banco de memoria añadiendo de esta forma latencia a la ejecución. Este problema generalmente es producido por el acceso a memoria local de manera no consecutiva, obteniendo un rendimiento similar a si utilizamos la memoria global.

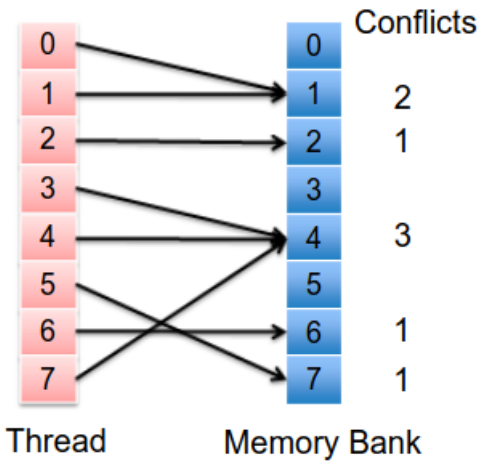

Figura 2.8: Conflictos en el banco de memoria [21].

Como podemos ver en la figura 2.8, se producen dos conflictos en los bancos de memoria, uno en el número uno producido por el acceso tanto de los hilos cero y uno y otro en el banco de memoria número cuatro en donde intentan acceder a sus recursos los hilos tres, cuatro y siete. La latencia producida será igual al valor más alto del conflicto producido, en este caso para completar todos los accesos tardaremos tres veces más de tiempo.

Memoria privada

Cada elemento de procesamiento tiene su memoria privada. Un programa del *kernel* puede asignar una variable o un objeto de memoria intermedia en la memoria privada utilizando la etiqueta *private* (o *__private*). La memoria privada sólo puede ser accedida por un único elemento de trabajo y es más rápida que la memoria local (LDS), global y la principal.

La memoria privada se asigna primero a los registros privados de la ALU. Si no hay suficiente espacio para mantener los datos privados, estos se almacenarán en la memoria global. Por lo tanto, si no se tiene en cuenta el uso de los elementos a utilizar, terminará afectando al rendimiento del programa.

2.1.3. Sincronización

Una de las tareas más difíciles de la programación paralela es manejar la secuencia de múltiples hilos. En determinados casos será necesario ejecutar en serie parte de los programas paralelos para evitar resultados no esperados. En OpenCL, existen dos tipos de sincronización: la sincronización entre los elementos de trabajo dentro de un grupo de trabajo y la sincronización entre las *command queue* dentro de un contexto.

Sincronización de los elementos de trabajo:

- Barreras
- Operaciones atómicas
- Tuberías
- Eventos

Sincronización de las *command queue*:

- Barrera de *command queue*
- Eventos

De los métodos de sincronización mencionados, utilizaremos para este trabajo las barreras para la sincronización de los elementos de trabajo y los eventos para las *command queue*.

Barrera

Cuando un elemento de trabajo llega a una barrera esperará hasta que todos los elementos de trabajo pertenecientes al grupo de trabajo hayan alcanzado el mismo punto. Una vez conseguido, todos los elementos continúan funcionando (ver imagen 2.9). Para utilizar este método de sincronización, se emplea el uso de la función *barrier* con el *flag* «CLK_LOCAL_MEM_FENCE» que indica que todos los accesos a la memoria local una vez lleguen a esta barrera estén completados. La sincronización entre elementos de trabajo pertenecientes a grupos diferentes no es posible, debido a que OpenCL no garantiza una forma segura de sincronizar la ejecución independiente de los grupos de trabajo.

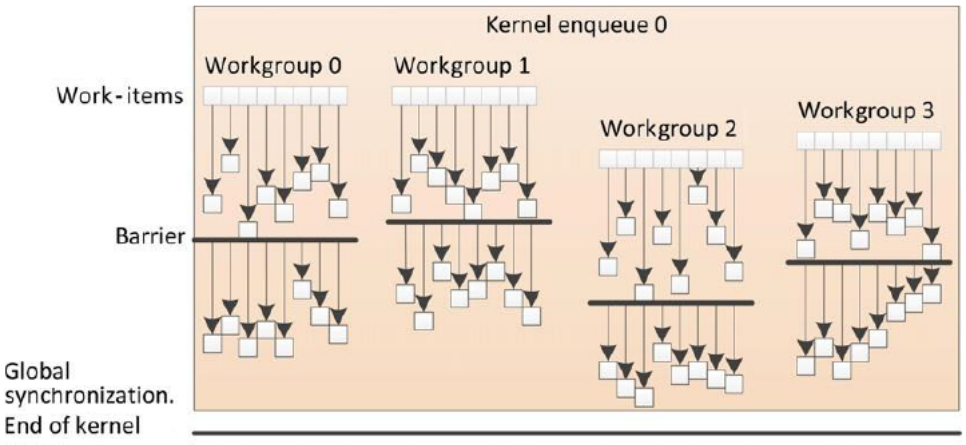

Figura 2.9: Barrera a nivel de grupo de trabajo [22]

Eventos

Para realizar una sincronización entre un programa del sistema y un *kernel* se debe implementar un comando que espera a que uno o más eventos cambien su estado a la etiqueta «CL_COMPLETE» o a un entero negativo. Hay tres pasos para una sincronización en OpenCL.

- Crear un comando que comience con *clEnqueue*()*.
- Insertar un punto de sincronización en el comando, mediante la creación de un objeto *cl_event*.
- Implementar la llamada a función *clWaitForEvents()*, la cual esperará a que el objeto evento cambie su estado a «CL_COMPLETE» en caso de haber terminado sin errores o por el contrario con un entero negativo.

2.2 AMD

Al ser OpenCL un estándar abierto para la programación paralela de sistemas heterogéneos, hemos de adaptar la programación al dispositivo con el cual estamos desarrollando. Es por ello que en este apartado explicaremos la GPU utilizada, tanto sus especificaciones como su arquitectura, necesarios para cuando abordemos su programación.

2.2.1. Microarquitectura

Fabricado por AMD, *Polaris* 10 es un circuito integrado de cuarta generación de la microarquitectura RISC (en castellano computador con conjunto de instrucciones reducido) SIMT (en castellano una instrucción, múltiples hilos) *Graphics Core Next* (GCN). En la siguiente figura 2.10, podemos visualizar la arquitectura de *Polaris* 10, en la cual destacamos las siguientes unidades:

- Memoria L2 cache de dos *megabytes* (MB).
- Interfaz de memoria de 256-bit dividida en ocho almacenadores de memoria (RB del inglés *Register Buffer*) de 32-bit DDR *cl_event*.
- 576 unidades de lectura/escritura de 32-bit repartidos en 16 por cada unidad de cómputo.
- Dos planificadores de *hardware* (HWS por sus siglas en inglés).
- Un procesador de comandos de gráficos.
- 36 unidades de cómputo (CU) las cuales veremos en detalle en la figura2.11
- Cuatro motores de cómputo asíncronos (siglas en inglés ACE) que presiden la asignación de recursos, el cambio de contexto y las prioridades de las tareas.

Como GCN está construido para trabajar simultáneamente en múltiples tareas, los ACEs programan independientemente los *wavefront* (definición que veremos posteriormente) a través de las CUs.

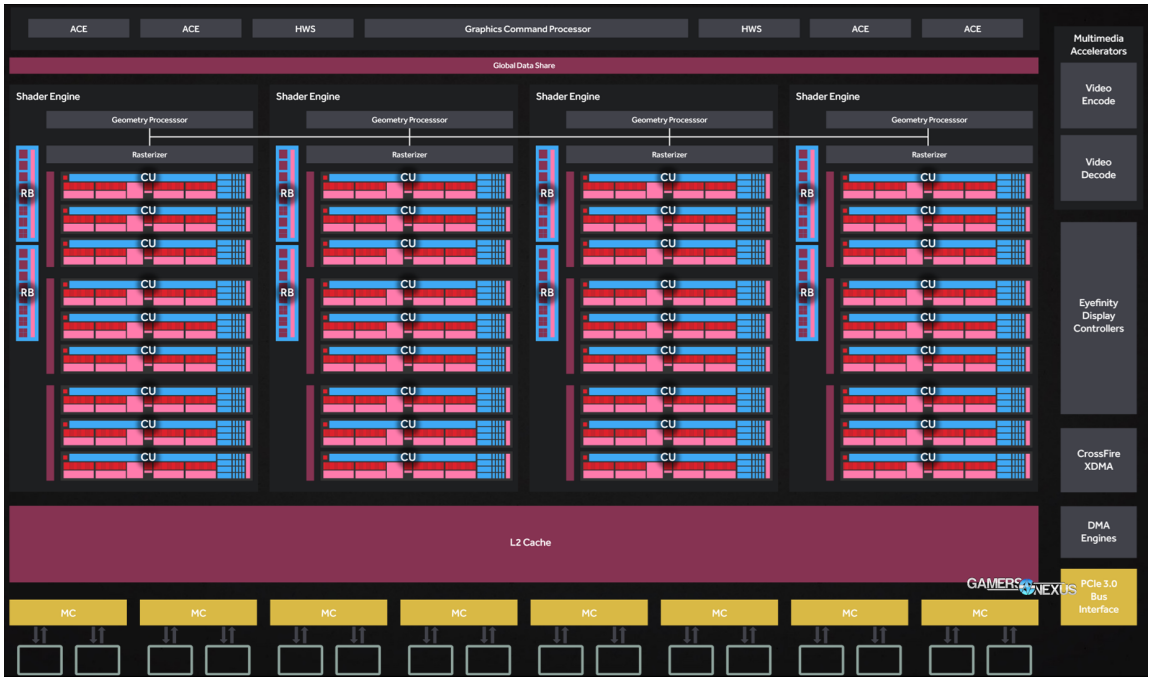

Figura 2.10: Arquitectura de *Polaris* 10 [23].

2.2.2. Unidades de cómputo

Las unidades de cómputo (CU) son el bloque de construcción computacional básico de la arquitectura GCN. Cada una de estas unidades está compuesta por los siguientes componentes:

- Cuatro núcleos vectoriales SIMD.
- Una unidad de salto y mensajes.
- Cuatro registros de propósito general vectorial (VGPR) de 64 *kilobytes* (KB), un planificador
- Cuatro unidades de filtro de textura, 16 unidades de carga/almacenamiento de textura
- 64KB de memoria local compartida entre toda la unidad de cómputo
- Un registro de propósito general escalar (SGPR) de cuatro *kilobytes* (KB).
- Una cache L1 de 16 KB para datos compartida entre 4 CUs adyacentes
- Una unidad escalar dedicada a operaciones matemáticas complejas, tales como funciones logarítmicas, seno, coseno, entre otras

La razón por la cual se ha empleado una unidad escalar dentro de la CU es para evitar que el núcleo vectorial de SIMD ejecute una instrucción sobre un escalar en lugar de un vector que es para lo que está diseñado y de esta manera no interrumpir la operación utilizando más ciclos de reloj para completar el cálculo.

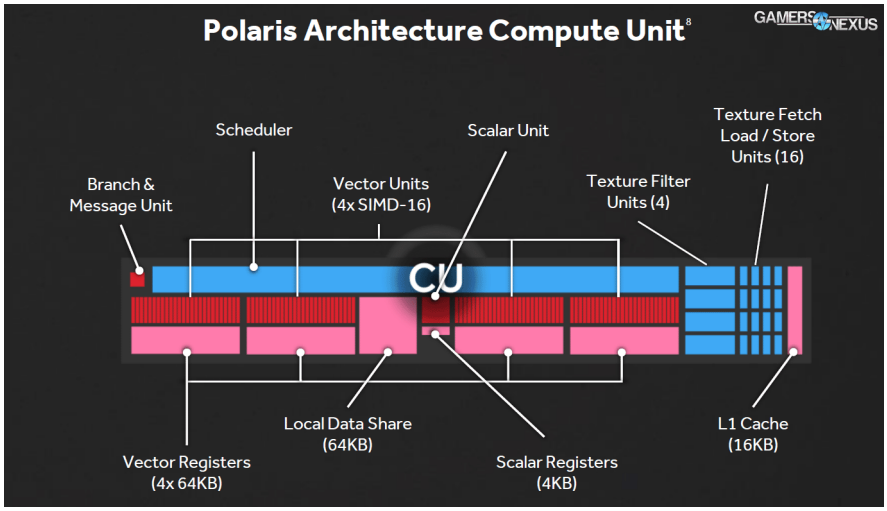

Figura 2.11: Arquitectura Unidad de cómputo en *Polaris* 10 [23].

2.2.3. Núcleo SIMD

Si realizamos una mirada en profundidad sobre las unidades de cómputo (CU) veremos que cada núcleo SIMD está formado por 16 ALUs vectoriales para tipos enteros y coma flotante por núcleo llamados *Stream Processors*, un contador de programa de 48-bit, una memoria intermedia de instrucciones para diez *wavefronts* y un registro de propósito general vectorial (VGPR) de 64 KB de espacio, uno de los cuatro que hemos visto anteriormente en la figura 2.12.

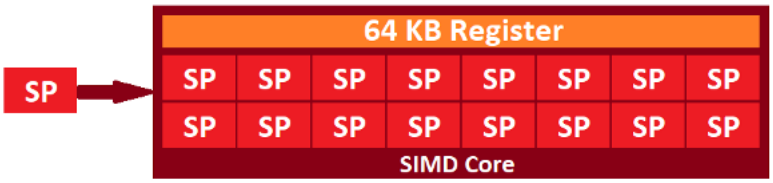

Figura 2.12: Diagrama de un núcleo SIMD [24]

2.2.4. Planificador de la unidad de cómputo y el concepto de *wavefront*.

Un *wavefront* es la unidad mínima de ejecución en la microarquitectura GCN y está compuesta por 64 elementos que trabajan de forma paralela ejecutando la misma instrucción, representando cada uno de estos elementos a un *work item*. Para su ejecución esta unidad debe ser asignada por el planificador a un núcleo SIMD, el cual como podemos ver en el subapartado anterior cada uno de estos componentes tiene la capacidad de procesar hasta 16 *work items* por cada ciclo de reloj. Esto supone que para que un *wavefront* se ejecute en su totalidad deberán pasar cuatro ciclos de reloj como mínimo, ya que diversos factores pueden retrasar la finalización del proceso como pueden ser la espera de un dato en memoria, ver figura 2.13, la complejidad de los datos a calcular o la divergencia de caminos producida una instrucción de salto en el control de flujo. Ya que todos los *work items* pertenecientes al *wavefront* deberán ser ejecutados en ambos caminos.

Como podemos ver en la figura 2.14, cuando un *wavefront* (T0) se queda en espera de un dato en memoria, el planificador decide que se utilice otro (T1) para no detener

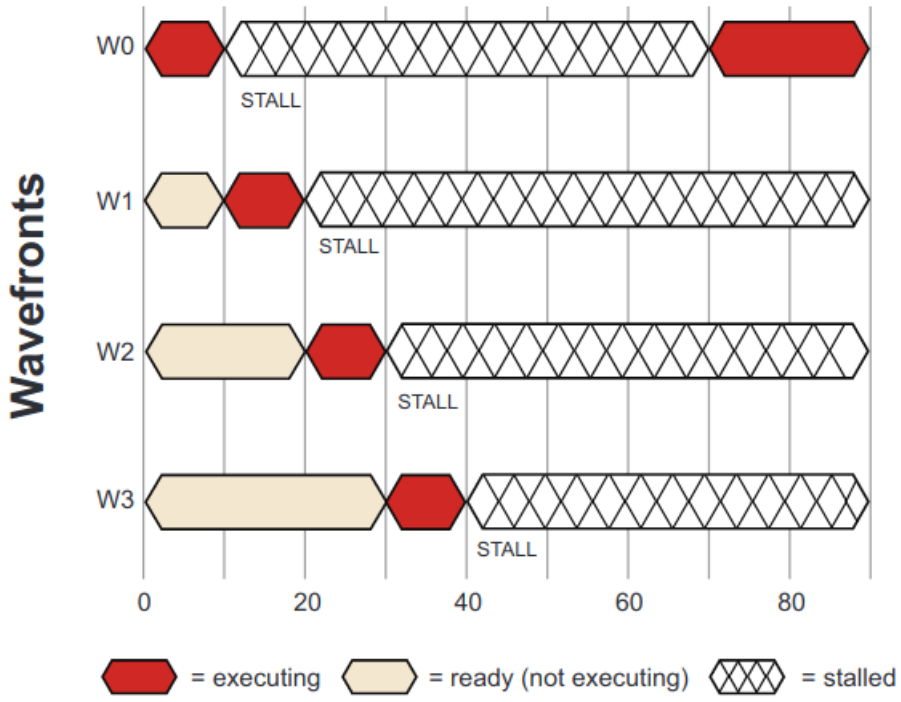

Figura 2.13: Unidad de cálculo en espera debido a dependencia de datos [16].

el flujo de ejecución y así sucesivamente ocultando de esta manera la latencia de la carga de datos de la memoria. Cabe destacar que el número máximo de *wavefronts* que se pueden asignar a los núcleos SIMD son diez cada uno, siendo no intercambiables entre estos componentes. En total, cada unidad de cálculo obtiene como máximo 40 *wavefronts* dependiendo de la complejidad de los datos que se encuentran en estas unidades, ya que pueden hacer disminuir este número.

Wavefront y OpenCL

En OpenCL, los *work-items* se dividen en *work-groups*, y estos luego son divididos en *wavefronts*, por tanto, un *wavefront* es un grupo de *work-items*. Normalmente, cada *work-group* se asigna a una unidad de cálculo. Durante el tiempo de ejecución, el primer *wavefront* se envía a la unidad de cálculo para que se ejecute, posteriormente se envía el segundo y así sucesivamente. Los *work-items* dentro de un *wavefront* se ejecutan en paralelo, a diferencia de los *wavefronts* que se ejecutan secuencialmente dentro de la unidad de cálculo (ver figura 2.15).

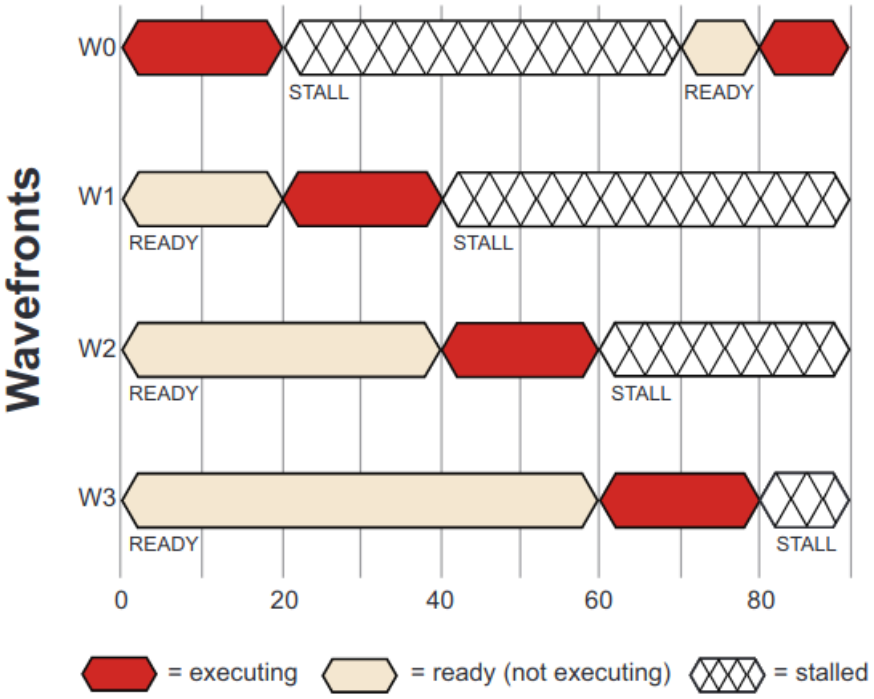

Figura 2.14: Ejecución de los distintos *wavefronts* en una unidad de cómputo [16].

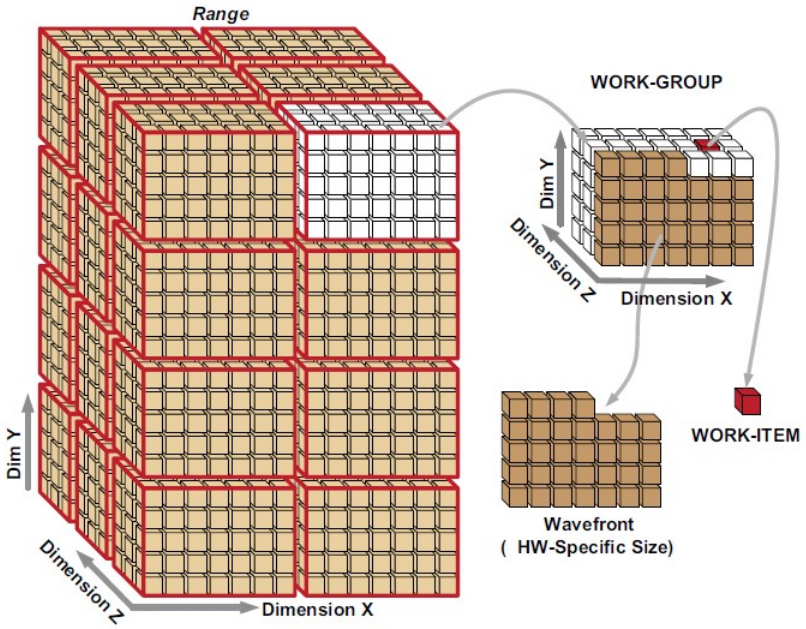

Figura 2.15: Relación entre los conceptos de *wavefront* con los grupos y elementos de trabajos en una matriz tridimensional [16].

CAPÍTULO 3
Aplicación para el entrenamiento de redes neuronales HELENNA

En este capítulo, se explica la aplicación HELENNA, la cual esta desarrollada para el entrenamiento de redes neuronales. Esta herramienta permite la obtención de modelos mediante una topología de red neuronal y un conjunto de datos de entrenamiento. Cabe resaltar que, HELENNA se ha desarrollado por el grupo de arquitecturas paralelas (GAP) de la Universitat Politècnica de València.

3.1 Descripción de la Aplicación

La aplicación HELENNA (HEterogeneous LEarning Neural Network Application) se centra en el entrenamiento de redes neuronales y su inferencia. Esta herramienta está destinada al ámbito de la investigación y docencia ligados con proyectos Europeos (H2020 RECIPE [25]), actualmente en ejecución. HELENNA es un proyecto que se ha creado en el Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València por los investigadores del grupo de investigación GAP, perteneciente a este departamento.

Esta herramienta se define por su especialización en diversas arquitecturas de cálculo como GPUs, CPUs y FPGAs. Teniendo el objetivo adicional de adaptarse a arquitecturas heterogéneas nuevas como JETSON XAVIER, una aplicación embebida creada por NVIDIA [26] o como sistemas similares a las TPUs de Google [27] y similares. Cabe mencionar que, por el momento, los últimos dispositivos mencionados no están soportados. Además, esta herramienta tiene como objetivo la utilización de procesos de inferencia de bajo consumo y en tiempo real.

Dentro de la aplicación, podemos ver que HELENNA permite al usuario elegir el dispositivo que vaya realizar el cálculo de la red neuronal. En este momento, la herramienta brinda soporte para los dispositivos GPU, CPU y FPGAs. Además de librerías y tecnologías como MKL, clBLAS, cuBLAS, AVX y AVX512.

Otra característica de HELENNA es que permite la realización de entrenamientos de redes neuronales en sistemas distribuidos en un *cluster*. Esto es realizado por el modelo de paralelismo de datos, en donde se distribuye y se sincroniza periódicamente la información a los distintos nodos pertenecientes al *cluster*. Para este tipo de soporte, se emplean las primitivas de sincronización y transferencia mediante la interfaz de paso de mensajes (MPI).

Para el desarrollo de la aplicación HELENNA se ha empelado el lenguaje de programación C. Utilizando una estrategia de llamadas a función que posibilita una programa-

ción eficiente y sencilla. Entrando en detalle, en un proceso de inferencia o entrenamiento se ha programado un multiplexor que, dado un dispositivo, elige las funciones de cálculo asociadas a la unidad las cuales están contenidas en un único fichero. Permitiendo de esta manera crear ficheros específicos para el dispositivo que queremos dar soporte. En este trabajo, se ha creado y enlazado a la herramienta HELENNA los módulos necesarios para la integración de la ejecución de GPU en OpenCL en esta aplicación.

Otra de las características del programa es la generación de estadísticas con los tiempos de ejecución de cada una de las operaciones que realizan los cálculos durante el entrenamiento de la red neuronal. Aportándonos información del impacto que tienen las diferentes implementaciones, dispositivos, arquitecturas y parámetros que componen la aplicación.

La realización del entrenamiento de una red neuronal requiere del uso de memoria para el conjunto de datos a entrenar. Este se crea mediante *buffers* de datos que almacenarán de forma temporal la información correspondiente a los datos de entrada, parámetros ligados a las capas, gradientes del proceso de entrenamiento, entre otros en el dispositivo seleccionado. Es por ello que en cada unidad se debe implementar las funciones de lectura y escritura correspondientes a los *buffers* de memoria para reducir al mínimo la transferencia de datos en el proceso de entrenamiento. Por último, destacaremos que en el presente trabajo se da a conocer la estrategia de memoria usada, así como también su implementación.

Por último y con respecto a la muestra de datos y topologías a analizar para el entrenamiento de redes neuronales, usaremos la colección de imágenes MNIST [28], en la cual se representan los números del cero al nueve de forma manuscrita, en escala de grises y normalizados a un tamaño de imagen de 28×28, ver figura 3.1. Adicionalmente utilizaremos la colección CIFAR10 [29], la cual contiene imágenes etiquetadas en diez clases diferentes y sus diferencias con respecto a la anterior colección es que su clasificación tiene una mayor complejidad de acierto. Esto se debe a que utilizan los colores azul, rojo y verde para su representación, además de que su tamaño de imagen está normalizada a un tamaño de 32×32, ver figura 3.2. Es por ello que en los resultados que se obtienen utilizando los mismos recursos, el porcentaje de acierto es menor que en los conseguidos en MNIST. En las topologías que utilizaremos en este proyecto, estas están especificadas en el apéndice A con el número de capas, su tipo, las neuronas y el tamaño de cada capa.

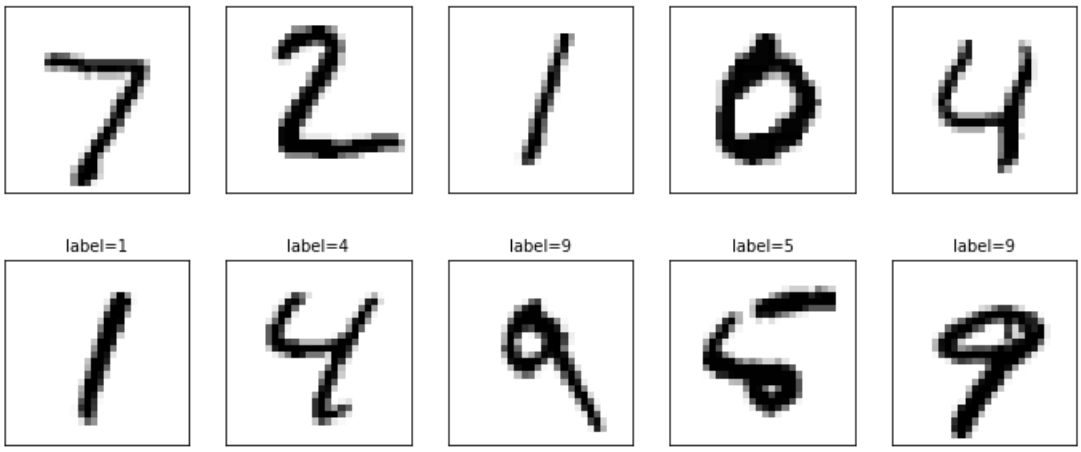

Figura 3.1: Imágenes pertenecientes al conjunto MNIST [30]

Figura 3.2: Imágenes pertenecientes al conjunto CIFAR10 [31]

3.2 Soporte a las GPUs con OpenCL

En el presente proyecto, se ha implementado el soporte de cálculo para las funciones que se utilizan en el entrenamiento de redes neuronales de la aplicación HELENNA para las diversas arquitecturas de GPUs. Para que todas estas funciones puedan ejecutarse en el coprocesador, se ha necesitado adicionalmente de la creación de *kernels* específicos escritos en el modelo de programación OpenCL. Las tablas 3.1 y 3.2 expuestas a continuación enumeran y describen las funciones implementadas en este trabajo:

Función	Descripción
matmul	Multiplicación entre dos matrices
matmul_bt	Multiplicación entre dos matrices de las cuales la segunda está traspuesta
matmul_at	Multiplicación entre dos matrices de las cuales la primera está traspuesta
mat_add	Adición entre dos matrices
mat_sub	Sustracción entre dos matrices
matrix_transpose	Transposición de una matriz
matrix_transpose_square	Optimización de la transposición de una matriz si su número de filas es igual a su número de columnas
vect_to_matrix	Copia el vector dado en una matriz por filas el número de veces como elementos tenga dicho vector
vect_scalar_product	Multiplicación de un número escalar por un vector
mat_reduce_rows	Dada una matriz, se realiza en cada una de sus filas la suma de todos sus elementos y su resultado se almacena en el vector de salida
vec_axpy	Realiza la multiplicación de un número escalar con un elemento del vector de entrada, el resultado obtenido lo suma y almacena en el correspondiente elemento del vector de salida
matrix_relu	Dado un vector, escribe el valor cero si el elemento de entrada es negativo y mantiene su valor si es positivo
matrix_relu_der	Dado un vector, escribe el valor FALSE si el elemento de entrada es negativo y TRUE su valor si es positivo
matrix_softmax	Función de softmax
vec_V2subV1xK	Dado un escalar k y dos vectores V1 y V2, se realiza la siguiente operación V2 = V2 - (k * V1)

Tabla 3.1: Funciones implementadas (parte I).

Función	Descripción
vect_trunc	Dado un valor escalar, se recorre los elementos de un vector en donde si el valor seleccionado es menor que el escalar, es sustituido por el cero
set_vec	Dado un vector, se almacena en todos sus elementos el valor pasado por parámetro
vec_copy	Realiza la copia del vector de entrada y lo almacena en el vector de salida
vec_copy_with_stride	Realiza la copia del vector de entrada y lo almacena en el vector de salida teniendo en cuenta la distancia entre elementos del vector.
im2col	Realiza la función im2col
col2im	Realiza la función col2im
matadd_col	Dada una matriz, se realiza en cada una de sus columnas la suma de todos sus elementos y su resultado se almacena en el vector de entrada
copy_src_dst	Realiza la copia con desplazamiento del vector de entrada y lo almacena en el vector de salida teniendo en cuenta el desplazamiento pasado por parámetro
vect_mult	Multiplicación entre dos vectores dados
matrix_elwise	Dadas dos matrices, se realiza la multiplicación en forma de vector
vect_step	Recorre los elementos del vector de entrada y en caso de ser positivo escribe un uno en el vector de salida, de lo contrario se almacena un cero
maxpooling	Realiza la función *maxpooling*
demaxpooling	Realiza la función *demaxpooling*
zero_vec	Dado un vector, almacena el valor cero en todos sus elementos
vector_limit	Dados dos valores que actúan como umbrales mínimo y máximo y un vector, se normaliza todos los elementos entre dichos valores
vect_mult_add	Realiza el sumatorio del resultado obtenido de la multiplicación de los elementos pertenecientes a los dos vectores de entrada y se almacena en el vector de salida
vect_mult_add_offset	Realiza el sumatorio del resultado obtenido de la multiplicación de los elementos pertenecientes a los dos vectores de entrada y se almacena en el vector de salida teniendo en cuenta la separación entre los elementos de los vectores
matrix_sigmoid	Realiza la función *sigmoid*
matrix_sigmoid_der	Ejecuta la derivada de la función *sigmoid*

Tabla 3.2: Funciones implementadas (parte II).

3.3 Optimización de cálculo con OpenCL

Debido a que OpenCL es un estándar abierto el cual pueden utilizar diferentes arquitecturas, para obtener mejores resultados tenemos que optimizarlo acorde al dispositivo seleccionado. Para este trabajo, desarrollaremos en HELENNA la implementación de las funciones que permitirán dar soporte a la ejecución de redes neuronales en GPU y posteriormente optimizaremos para el dispositivo empleado en este proyecto, una tarjeta gráfica AMD RX 480.

3.3.1. Función *matmul*

El objetivo de la función *matmul* es obtener la matriz producto de dos matrices dadas, estas se corresponden con la imagen de entrada a procesar y los pesos de las neuronas. Esta operación se realiza cada vez que las diferentes imágenes de entrada atraviesan las distintas capas que posee la red neuronal en las cuales se encuentran las neuronas. La

multiplicación de matrices por tanto, es fundamental en el cálculo de redes neuronales, ya que la mayor parte del tiempo de la ejecución se emplea en el cómputo de esta función.

Una vez vista la importancia que tiene la función *matmul* en el cálculo de redes neuronales, analizaremos su coste computacional tanto en redes convolucionales como en densamente conectadas. Para ello ejecutamos en la aplicación HELENNA el entrenamiento de las topologías *MNIST-Medium* (densamente conectada) y *MNIST-Conv* (convolucional) sobre la GPU con el modelo de programación OpenCL que hemos dado soporte.

Para la ejecución en HELENNA, necesitamos de los siguientes parámetros para configurar y desarrollar el entrenamiento:

- Número de épocas: indica las veces que utilizaremos el conjunto de datos en el proceso de aprendizaje.
- *Learning rate*: indica el tamaño de movimiento que haremos para alcanzar el valor mínimo en la función de perdida (*loss function*).
- *Momentum*: constante que se utiliza para ponderar el *Learning rate* gradiente descendiente desde posiciones aleatorias para evitar el se quede atascado en un mínimo local.
- *Batch size*: particiona los datos de entrada en lotes que vamos a utilizar en cada iteración del entrenamiento para la actualización de los parámetros del modelo.

A estos parámetros les daremos los valores que están listados en la tabla 3.3 para obtener posteriormente los resultados que veremos en la figura 3.3.

Parámetro	**Valor**
Número de épocas	5
Learning rate	0,01
Momentum	0,9
Batch size	64

Tabla 3.3: Valores con los que se ejecutan los parámetros para el entrenamiento de redes neuronales en HELLENA.

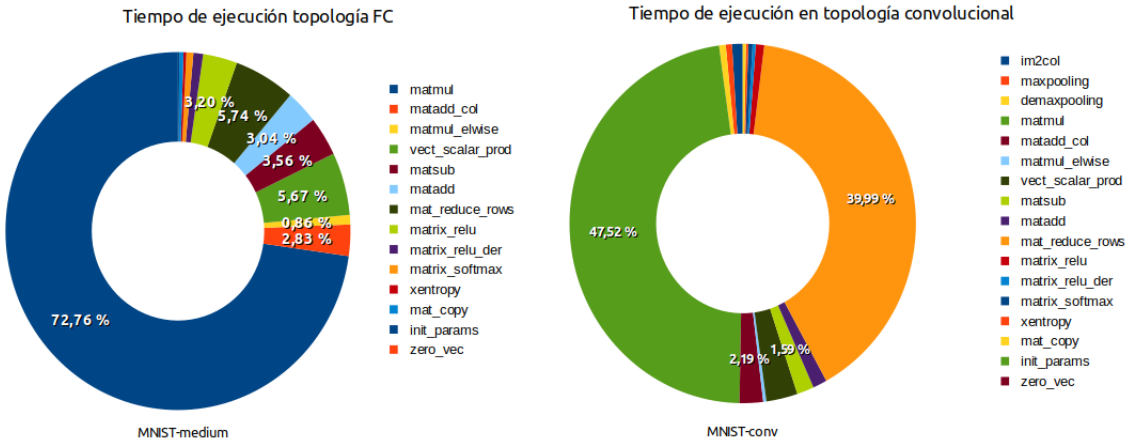

Figura 3.3: Coste en porcentaje de la ejecución de la clasificación MNIST en HELENNA

Como podemos visualizar, para ambas topologías la función de la multiplicación de matrices es la de mayor tiempo de cómputo. Debido a esto, para obtener resultados en

menor tiempo, realizaremos y detallaremos la optimización del código de la función *matmul* y su impacto en el desarrollo de esta mejora. Inicialmente, tenemos la función con la cual se ha dado soporte para las GPUs en OpenCL para la aplicación HELENNA (ver listado 3.1).

```
__kernel void matrixMultiplication(const __global float* A,const __global float
    * B,__global float* C,
                                   const int M, const int K, const int N) {

    // Identificador de los hilos a nivel global
    int globalRow = get_global_id(0);
    int globalCol = get_global_id(1);
    //Ejecucion del calculo sobre la dimension K
    float value = 0.0f;
    for (int k = 0; k < K; k++)
    {
        value += A[(globalRow * K) + k] * B[(N * k) + globalCol];
    }

    //Almacenamiento de la variable en la matriz de salida
    C[(globalRow * N) + globalCol] = value;
}
```

Listing 3.1: Código de la función matrixMultiplication

La función problema tendrá como parámetros las matrices de entrada A ($M \times K$), B ($K \times N$) y la matriz de salida C cuya dimensión es $M \times N$. Del tamaño de esta última paralelizaremos y desplegaremos los hilos, tantos como el número de filas (*globalRow*) y de columnas (*globalCol*) posea la matriz C. Acto seguido, realizaremos la ejecución recorriendo el parámetro K que es común entre ambas matrices de entrada, siendo el número de columnas en el caso de la matriz A y el número de filas en el de la matriz B (ver figura 3.4). Además, la variable *value* la utilizaremos como registro para almacenar el producto resultante de la fila A por la columna B y posteriormente guardaremos dicho valor en el lugar correspondiente en la matriz de salida. Cabe mencionar que, las matrices de entrada están almacenadas en memoria mediante una ordenación de filas, lo cual nos exigirá guardar de esta misma manera la matriz de salida. Podemos visualizar que no hay ningún método de sincronización explicito en el código, esto se debe a que los elementos de diferentes grupos de trabajo se ejecutan en diferentes unidades de cálculo y por ello nunca utilizarán la misma memoria para realizar lecturas o escrituras.

Como podemos ver en la figura 3.4 el acceso a los elementos en el caso de la matriz A se realizan de forma contigua mientras que los de la matriz B no, esto último produce que a la hora de seleccionar el siguiente dato, se tenga que buscar y acceder al bloque de memoria global de la GPU en la que se encuentre este dato, agregando más tiempo al cómputo total.

3.3.2. Especificaciones del dispositivo

Antes de empezar con la optimización de la función *matrixMultiplication* debemos tener en cuenta las especificaciones del dispositivo en donde se ejecutarán los *kernels*. Para este trabajo utilizaremos la tarjeta gráfica AMD RX 480 en su versión de 4GB de VRAM. Desde la página del fabricante, podemos obtener los siguientes datos de la GPU en cuestión (ver tabla 3.4).

Adicionalmente, necesitaremos de más parámetros para poder desarrollar una optimización sobre el dispositivo que estamos empleando. Para ello, utilizaremos la herra-

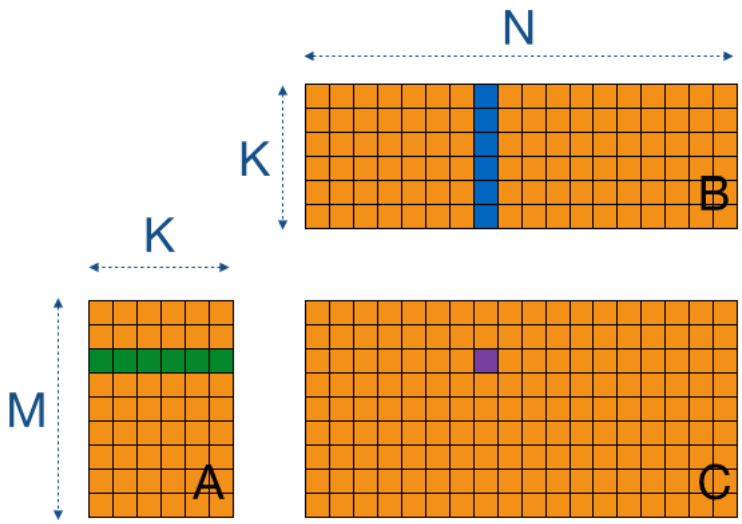

Figura 3.4: Ilustración de la función *matmul* [14]

Arquitectura	Polaris 10
Unidades de cómputo	36
Número de núcleos	2304
Frecuencia base de reloj	1120 MHz
Rendimiento (máximo teórico)	5.8 TFLOPS
Tamaño de memoria	4 GB GDDR5
Ancho de memoria	224 GB/s
Velocidad de la memoria	7 Gbps
Soporte para operación de multiplicación-suma fusionadas (FMA)	Sí
Interfaz de memoria	256-bit
Soporte para OpenCL	Sí
Soporte para CUDA	No

Tabla 3.4: Especificaciones AMD RX 480.

mienta *clinfo*, la cual enumera el número de dispositivos en el sistema que pueden utilizar OpenCL y sus propiedades, como podemos ver en la tabla 3.5.

Como podemos ver, hay definiciones de las cuales se hacen referencia en el capítulo dos. Con estas tablas 3.4 y 3.5 podremos saber los limites y valores preferidos por el dispositivo para obtener un rendimiento mejor a la hora de establecer variables en la optimización de la función.

3.3.3. Técnicas de optimización

Las técnicas de optimización para el producto de matrices que veremos a continuación han sido estudiadas y extraídas de la página *OpenCL GEMM Tutorial* [14]. En este artículo se explica paso a paso y con ejemplos prácticos la optimización a mano de una implementación de multiplicación de matrices en OpenCL, con la cual podemos superar en rendimiento a la libreria clBLAS. Las principales diferencias entre la optimización presentada en este trabajo y las vistas en la página son el ámbito en donde serán implementadas. Debido a que en el presente proyecto tenemos en cuenta los diversos tamaños de matriz para las distintas multiplicaciones de matrices existentes dentro del cálculo para el entrenamiento de redes neuronales en la aplicación HELENNA. Mientras que en el artículo solamente se realizan multiplicaciones de matrices cuadradas conociendo previamente su tamaño. Adicionalmente, tenemos que destacar que el método de ordenación en este trabajo es mediante es filas, es decir *row major order* debido a que la aplicación

Dimensión máxima de ítem de trabajo	3
Tamaño máximo del grupo de trabajo	256
Tamaño máximo del grupo trabajo preferido por el dispositivo	256
Tamaño de grupo de trabajo preferido (múltiplo)	64
Ancho del Wavefront	64
Tamaño nativo del vector preferido para el tipo *float*	1
Tamaño nativo del vector preferido para el tipo *double*	1
Tamaño de la memoria global cache	16384 Bytes
Tamaño de la memoria local	32768 Bytes
Tamaño de la memoria local por unidad de cómputo	65536 Bytes
Número de bancos de memoria local	32

Tabla 3.5: Especificaciones AMD RX 480 con la herramienta clinfo.

almacena de esta forma en memoria los datos. Mientras que en la página se utiliza una ordenación por columnas (*column major order*).

Por último, destacaremos dos de las técnicas empleadas en el artículo que no utilizaremos. La primera de ellas es la transposición de una de las matrices de entrada [32], en concreto la matriz que leemos por columnas, ya que accedemos a sus datos de forma no contigua. Es por ello que, empleando este método, se utilizan las propiedades de la matriz cuadrada y la paralelización de OpenCL obteniendo de esta manera un coste computacional bajo con respecto a la multiplicación de matrices que se realiza después de esta acción. Así, en el momento de realizar la operación GEMM, accederemos a los datos de estas matrices en memoria de forma contigua logrando una mejor eficiencia. La segunda técnica es la del relleno, mejor conocida por traducción al ingles, *padding*. Este método consiste en aumentar el tamaño de la matriz de entrada agregando información que no afecte su resultado de manera que obtengamos una matriz que sea divisible por las dimensiones de la submatriz. Explicaremos en detalle de la importancia de este hecho en el siguiente apartado. De esta forma evitamos utilizar en el *kernel* instrucciones de un coste elevado para las GPUs como son las de sentencias condicionales. Debido a los diferentes tamaños de matriz que se ejecutan en la aplicación HELENNA, siendo en el peor de los casos matrices vectorizadas, podemos obtener un elevado coste computacional que nos obliga a descartar esta técnica.

3.3.4. Optimización mediante la utilización de la memoria local

Una de las estrategias de optimización que emplearemos será utilizar la memoria local, ya que es una memoria con mayor rapidez y menor latencia que la memoria global, en donde están almacenadas las matrices a calcular. En el código anterior 3.1 podemos ver que en la memoria global se realizan $2 \cdot M \cdot N \cdot K$ lecturas y $M \cdot N$ escrituras de los datos. Este hecho sumado a que el cálculo para obtener el resultado se puede realizar en una sola instrucción debido a que el dispositivo soporta instrucciones fusionadas de multiplicación - suma (FMA) (ver fila novena de la tabla 3.4), estaremos obteniendo una sola instrucción por cada dos accesos a memoria. Dado al bajo rendimiento que ofrece la utilización de la memoria global para la multiplicación de matrices, emplearemos un método que disminuya el número mayor posible de accesos a esta memoria.

Este método consiste en crear bloques en memoria local, los cuales serán submatrices de las matrices de entrada. En estas estructuras guardarán en memoria local los datos de la memoria global y permitirán la reutilización de los mismos, con ello disminuiremos el coste en el acceso a memoria global. Con este cambio, para calcular un bloque de la matriz de salida, seguiremos necesitando de las filas y columnas correspondientes de las matrices de entrada A (en tonalidad verde) y B (en tonalidad azul) respectivamente. Pero con los bloques situados en memoria local, podemos actualizar iterativamente los valores

en la submatriz de salida sumando los resultados de las multiplicaciones de los bloques A por B correspondientes para obtener el resultado (ver figura 3.5).

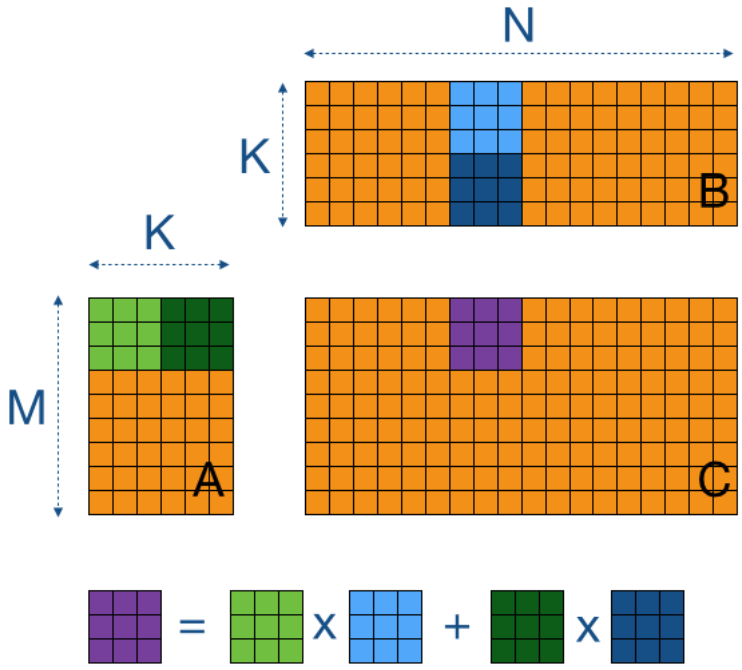

Figura 3.5: Diagrama ejecución *matmul* con memoria local [14]

Para implementar esta mejora, previamente a modificar el *kernel*, se ha de modificar su llamada a ejecución. En el código anterior 3.1, solo se debía especificar el tamaño del problema, es decir, el número de elementos que se iba a ejecutar en paralelo y OpenCL implícitamente se encargaba de la división en grupos de trabajo [16]. Para la nueva optimización del código, necesitaremos especificar en la llamada al *kernel* de *matmul* el número y dimensión de los elementos de trabajo que formarán parte del grupo de trabajo. Ya que crearemos en memoria local dos bloques, una por cada matriz de entrada, en las cuales se almacenarán los elementos de trabajo de cada grupo que ejecute la función.

Para ello, debemos tener en cuenta que cada dispositivo tendrá un limite físico tanto en la dimensión como en el número máximo permitido de elementos de trabajo por cada grupo, para la tarjeta gráfica RX 480 la dimensión máxima permitida es de tres, mientras que el valor limite de elementos de trabajo es de 256 (primera y segunda fila de la tabla 3.5 respectivamente). Esto significa que podremos ejecutar en total 256 hilos para un grupo de trabajo, en el cual cada hilo se encargará de un elemento, en una dimensión. Como necesitaremos crear una submatriz bidimensional a nivel de memoria local, debemos reducir este valor a 16 para cada una de las dimensiones. Cabe destacar que la creación de estas estructuras realizará de forma estática ya que OpenCL no permite asignaciones de memoria en tiempo de ejecución[7].

Otro aspecto a tener en cuenta es que el número de elementos totales del problema debe ser divisible en todas sus dimensiones por el tamaño del grupo de trabajo especificado. Este requisito es necesario para OpenCL C versión 1.2 [7], la cual utiliza el dispositivo que empleamos. Debido a que el tamaño de las matrices de entrada es variable incluso en una ejecución de entrenamiento de redes neuronales, debemos modificar el valor de entrada del número total de elementos para que sea múltiplo del tamaño del grupo de trabajo. Esto lo realizaremos en cada llamada al *kernel* de *matmul* mediante la función *resize_global* que verificará si los valores anteriormente descritos cumplen con este criterio y en caso de no coincidir cambiará el valor del número total de elementos al múltiplo

mayor más cercano, ver listado 3.2.

```
    //Redimensionar las dimensiones de la matriz para proporcionar
        divisibilidad
    int tm, tn;
    resize_global(rows_a, rows_b, &tm, &tn, TS);

    //Declarar el numero de elementos de trabajo dentro del grupo y el numero
        total de elementos por dimension
    const size_t localThreads[2] = { TS, TS };
    const size_t globalThreads[2] = { tm, tn };

    // Enviar a la cola de comandos el kernel
    ret = clEnqueueNDRangeKernel(command_queue, kernel_matrix_multiplication,
                        2, NULL, globalThreads, localThreads, 0, NULL, &event);
```

Listing 3.2: Modificación al código de la llamada al *kernel* de matrixMultiplication con memoria local

Con la redimensión de las matrices se soluciona el problema de la divisibilidad de los tamaños de los mismos y de grupo de trabajo, pero crea otro. El acceso a memoria de un elemento que este dentro del espacio declarado en la llamada al *kernel* pero no de la matriz. Este inconveniente lo resolveremos dentro del *kernel* de *matmul* mediante expresiones condicionales comprobando si los limites de la matriz han sido o no superados, en caso de acceder a un elemento que no se encuentre dentro de las dimensiones de la matriz, le asignaremos el valor de 0 a la submatriz, ya que este número no altera el resultado final.

A continuación, expondremos y explicaremos el *kernel* de *matrixMultiplication*, contenido en el listado 3.3.

```

__kernel void matrixMultiplication(const __global float* A,const __global float
    * B,__global float* C,
                                   const int M, const int K, const int N) {

    // Identificador de los hilos
    const int row = get_local_id(0);
    const int col = get_local_id(1);
    const int globalRow = TS*get_group_id(0) + row;
    const int globalCol = TS*get_group_id(1) + col;

    // Creacion de submatrices en memoria local para almacenar TS*TS elementos
        de las matrices de entrada A y B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    // Inicializar los registros de acumulacion
    float value = 0.0f;

    // Ejecucion del calculo sobre el numero de submatrices creadas
    const int numTiles = K/TS;
    for (int t=0; t<numTiles; t++) {
        // Carga de una submatriz de A y B en la memoria local
        const int tiledRow = TS*t + row;
        const int tiledCol = TS*t + col;
        if(globalRow < M && tiledCol < K)
            Asub[col][row] = A[globalRow*K + tiledCol];
        else
            Asub[col][row] = 0;
        if(tiledRow < K && globalCol < N)
```

```
29              Bsub[col][row] = B[tiledRow*N + globalCol];
30          else
31              Bsub[col][row] = 0;
32          // Sincronizar hasta que las submatrices esten completas
33          barrier(CLK_LOCAL_MEM_FENCE);
34
35          // Ejecucion del calculo sobre todos los valores de una submatriz
36          for (int k=0; k<TS; k++) {
37              value += Asub[k][row] * Bsub[col][k];
38          }
39
40          // Sincronizar hasta que todos los hilos ejecuten el calculo
41          barrier(CLK_LOCAL_MEM_FENCE);
42      }
43
44      // Guardar el resultado final en la matriz de salida
45      if(globalRow < M && globalCol < N)
46          C[(globalRow * N) + globalCol] = value;
47  }
```

Listing 3.3: Código del *kernel* matrixMultiplication con memoria local

Como podemos observar, para el desarrollo de esta solución se añaden las siguientes variables.

- *TS*: indica el tamaño de la dimensión de la submatriz. *TS* está definida como constante y su valor dependerá de la longitud del problema y limite de memoria local asignado para cada unidad de cálculo del dispositivo.
- *numTiles*: número de submatrices creadas.
- *tiledRow*: indica la fila de la submatriz.
- *tiledCol*: indica la columna de la submatriz.

Explicaremos el código en tres partes, la primera de ellas es la carga de datos desde la memoria global donde se sitúan las matrices hacia la memoria local mediante los bloques creados en este último componente. Para seleccionar el valor correspondiente modificaremos además los identificadores de los hilos, dejaremos de utilizar el identificador global (*get_global_id*) para usar una combinación entre los identificadores local (*get_local_id*) y de grupo (*get_group_id*). Con estos índices y el tamaño de submatriz en la dimensión en la que se encuentre *TS* podremos calcular el identificador global en caso de ser necesario (linea octava y novena del código 3.3).

Como en el anterior código 3.1, la carga de las submatrices lo haremos recorriendo la dimensión *K* correspondiente a las columnas de la matriz A y las filas de la matriz B, con la diferencia de que a este valor lo dividiremos por el tamaño de la constante *TS* para calcular el número de submatrices creadas y con ello poder recorrer, leer y almacenar correctamente los datos. Por cada iteración del bucle exterior, cada hilo se encargará de realizar dos cargas globales, una por cada matriz de entrada, y almacenándolos en su correspondiente bloque.

Al utilizar variables locales, debemos de asegurarnos de su sincronización para evitar obtener resultados erróneos, para ello utilizaremos las barreras explicadas en el capítulo dos. Estos métodos de sincronización estarán antes del cálculo de la obtención del resultado garantizando que los hilos hayan completado los bloques y después de finalizar dicha operación para inicializar otra carga de datos en la siguiente submatriz.

La segunda sección trata sobre cómputo de los bloques para la obtención del resultado, una vez obtengamos las submatrices completas, iteraremos sobre una de las dimensiones del bloque, de esta manera toda la fila de la submatriz A será multiplicada por la columna de la submatriz B un total de TS veces (ver figura 3.6). y almacenada en el registro *value*.

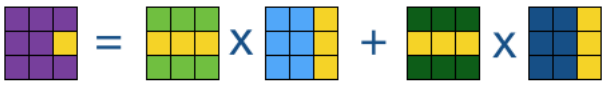

Figura 3.6: Diagrama de la obtención de un elemento en las submatrices [14]

Esto reducirá considerablemente los accesos a memoria global en un factor de TS consiguiendo de esta manera una mejora de rendimiento gracias a la baja latencia que nos proporciona la memoria local. Adicionalmente, se ha remplazado el acceso a la matriz B en elementos de memoria no consecutivos por elementos juntos dentro del bloque al almacenarlos por fila.

Finalmente, cuando se termine de procesar cada submatriz, se guardará el dato almacenado en el registro *value* en el lugar correspondiente de la matriz de salida ubicada en la memoria global.

A continuación, analizaremos el impacto de este método de optimización producido por tamaño del bloque, modificando la constante TS hasta obtener el valor óptimo de elementos por grupo de trabajo. Hay que tener en cuenta que cada dispositivo indica teóricamente el número en el cual se obtiene el mejor resultado posible, aunque existen factores como la complejidad computacional del *kernel* que pueden provocar que no se alcance dicho valor óptimo. De esta manera, ejecutaremos la aplicación con la modificación anteriormente comentada del *kernel* de *matrixMultiplication* con diferentes tamaños de bloque y comprobar de esta forma cual es la configuración que nos dará mejores resultados. Estos, los podemos ver en la figura 3.7.

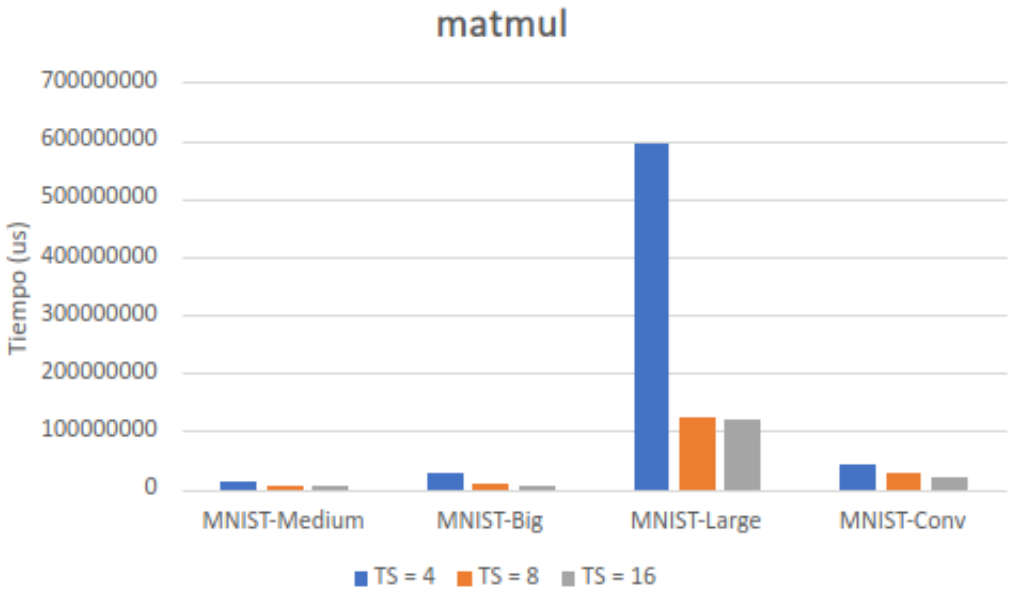

Figura 3.7: Tiempo de ejecución con diferentes valores de tamaño de bloque

Los valores utilizados en el diagrama han sido escogidos teniendo en cuenta el valor preferido de elementos de trabajo por grupo del dispositivo, el cual es de 256 (tercera fila

de la tabla 3.5), coincidiendo este valor con el máximo permitido por la GPU. Una vez aclarado esto, explicaremos los resultados obtenidos.

Como podemos ver, el valor preferido indicado por el dispositivo ($TS = 16$) ha sido el que mejor resultado nos ha dado siendo el que menor tiempo ha tardado en ejecutar las diferentes topologías. Además, observamos que la diferencia de tiempo con respecto a configuración de $TS = 8$ son mínimas mientras que utilizando una submatriz de tamaño 4×4 vemos que aumenta considerablemente el tiempo empleado para finalizar las ejecuciones, sobre todo cuando se calculan topologías más grandes como es el caso de *MNIST-Large*. Esto se debe a la microarquitectura utilizada por el dispositivo y su flujo de ejecución, explicados en el capítulo de OpenCL y AMD. Si revisamos la tabla 3.5 podemos ver que el ancho preferido del *wavefront* es de 64, eso significa que tanto ese valor como sus múltiplos serán los que mayor rendimiento nos den al aprovechar de forma óptima todo ese espacio. De esta forma, el bloque de menor tamaño solamente posee 16 elementos ocupando solo un cuarto del *wavefront*, provocando que el núcleo SIMD solo procese en el primer ciclo de reloj estos elementos y los siguientes tres ciclos esté a la espera de que termine el *wavefront* para devolverlos a la memoria. Es por ello que con los tamaños de submatriz de $TS = 8$ y $TS = 16$ con 64 y 256 elementos respectivamente, consiguen un mejor rendimiento. Finalmente, la diferencia de tiempo entre estos dos últimos valores mencionados se debe principalmente por una optimización en los accesos a memoria, ya que la configuración con el bloque de memoria más grande se dividirá en cuatro partes a la hora de ser ejecutado en el núcleo SIMD. Creando de esta manera cuatro controles de flujo que irán a la memoria privada del núcleo ocultando de esta manera la latencia con la memoria en lugar de solo un *wavefront* que es lo que envía el bloque de 8×8 al núcleo SIMD.

Por último, ejecutamos en la aplicación HELENNA las topologías *MNIST-Medium*, *MNIST-Big* y *MNIST-Conv* y medimos el tiempo que se ha necesitado para entrenar estas redes sobre la función *matmul*, utilizando el modelo con el que se ha dado soporte a OpenCL para GPU y comparándolo con su optimización a nivel de memoria local con bloques de (16×16).

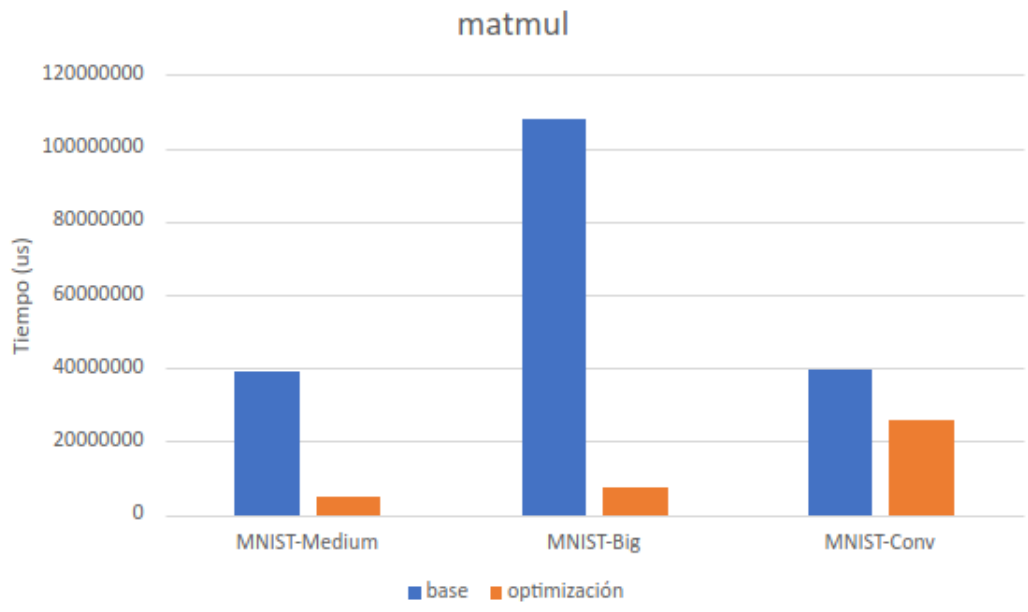

Figura 3.8: Comparación tiempo de ejecución

Como podemos ver en la figura 3.8, los resultados conseguidos en el entrenamiento de redes neuronales demuestran una reducción del tiempo de ejecución en la función

de la multiplicación de matrices de hasta 13 veces en topologías de mayor tamaño como *MNIST-Big*. Con esto podemos ver el impacto que tiene la utilización de la memoria local con respecto a la global y su optimización del tamaño del bloque para aprovechar al máximo las prestaciones que nos ofrece el dispositivo de cálculo.

3.3.5. Optimización mediante el incremento de elementos por hilo

Una vez reducido los accesos a memoria global, nos centraremos en el bucle interior donde realizamos el cómputo para la obtención del valor final. Podemos observar que en cada iteración del bucle se realizan dos cargas de un elemento desde la memoria local correspondientes a las submatrices y una operación de multiplicación-suma fusionada. Esto supone que una de cada tres instrucciones es de cálculo, por esta razón, nos centraremos a nivel de registro en incrementar el trabajo por hilo. Con esta optimización, podremos reducir los accesos a memoria local en beneficio de obtener una eficiencia a nivel de instrucción.

Para aumentar la cantidad de trabajo por hilo, crearemos una matriz bidimensional de registros con la cual cada hilo al momento de realizar el cómputo almacenará en ella más de un elemento por submatriz, produciendo de esta forma una disminución en los accesos a memoria local, así como también de los hilos utilizados. Lo que nos permite aumentar el tamaño de estos bloques para mantener la misma cantidad de hilos sin superar el limite del dispositivo del número de elementos de trabajo pertenecientes a un grupo (segunda fila de la tabla 3.5). Para lograr esto, debemos introducir dos nuevas constantes:

- WPT: indica el número de elementos en los que trabajará cada hilo por submatriz.
- RTS: representa el tamaño de la submatriz reducida, es decir, indicará el tamaño del grupo de trabajo y su valor está definido como $\frac{TS}{WPT}$.

Con estos recursos, podemos modificar el valor de TS teniendo en cuenta el limite superior delimitado por el tamaño de la memoria local (segunda fila de la tabla 3.5). Para calcular el espacio que necesitaremos en memoria debemos realizar la operación $(2 \cdot TS \cdot TS \cdot 4)$, en donde el primer valor representa la cantidad de submatrices declaradas en el *kernel*, seguido por tamaño en ambas dimensiones de estas estructuras y el último valor indica el tamaño de bytes del tipo declarado de la submatriz. También debemos tener en cuenta el valor de la constante WPT debido a que el tamaño del grupo de trabajo está representado por RTS.

Después de declarar las constantes con su valor, debemos en el momento de la llamada al *kernel* realizar la siguiente modificación (ver código 3.4), ya que con la anterior configuración nos hubiese dado un error con el tamaño del grupo de trabajo.

```
1        //Redimensionar las dimensiones de la matriz para proporcionar
             divisibilidad
2        int tm, tn;
3        resize_global(rows_a, rows_b, &tm, &tn, TS);
4
5        //Declarar el numero de elementos de trabajo dentro del grupo y el
             numero total de elementos por dimension
6        const size_t localThreads[2] = {TS/WPT, TS/WPT}; // RTS
7        const size_t globalThreads[2] = {(size_t) tm/WPT, (size_t)tn/WPT};
```

Listing 3.4: Modificación al código de la llamada al *kernel* de matrixMultiplication con aumento de trabajo por hilo

Una vez modificada la llamada al *kernel*, realizaremos la optimización del *kernel* de *matrixMultiplication* (ver código 3.3) para permitir el incremento de trabajo por hilo y un mayor tamaño de memoria local. Para ello, veremos la implementación del código 3.5.

```
__kernel
void matrixMultiplication(const __global float* A, const __global float* B,
    __global float* C, const int M, const int K, const int N) {

    //Identificador de los hilos
    const int  row = get_local_id(0);
    const int  col = get_local_id(1);
    const int  globalRow = TS*get_group_id(0)+row;
    const int  globalCol = TS*get_group_id(1)+col;

    // Creacion de submatrices en memoria local para almacenar TS*TS elementos
        de las matrices de entrada A y B
    __local float sA[TS][TS];
    __local float sB[TS][TS];

    //Asignar espacio en los registros
    float regA;
    float regB[WPT];
    float value[WPT][WPT];

    // Inicializar los registros de acumulacion
    for (int wm=0; wm<WPT; wm++) {
        for (int wn=0; wn<WPT; wn++) {
            value[wm][wn] = 0.0f;
        }
    }
    // Ejecucion del calculo sobre el numero de submatrices creadas
    const int T = (K-1)/TS+1;
    for(int t=0; t < T; t++){
        // Carga de una submatriz de A y B en la memoria local
        for (int w=0; w<WPT; w++) {
            for (int wn=0; wn<WPT; wn++) {
            int tiledRow = (t*TS+row) + w*RTS;
            int tiledCol = (t*TS+col) + wn*RTS;
            if((globalRow + w*RTS) < M && tiledCol < K)
                sA[col + wn*RTS][row + w*RTS] = A[(globalRow+w*RTS)*K+tiledCol
                    ];
            else
                sA[col + wn*RTS][row + w*RTS] = 0;
            if(tiledRow < K && (globalCol + wn*RTS) < N)
                sB[col + wn*RTS][row + w*RTS] = B[tiledRow*N+(globalCol+wn*RTS)
                    ];
            else
                sB[col + wn*RTS][row + w*RTS] = 0;
            }
        }
        // Sincronizar hasta que las submatrices esten completas
        barrier(CLK_LOCAL_MEM_FENCE);


        for (int k=0; k<TS; k++) {
            // Almacenar en registro los valores de sB
            for (int wn=0; wn<WPT; wn++) {
                regB[wn] = sB[col + wn*RTS][k];
            }
            // Cargar valor en registro de sA y ejecucion del calculo
            for (int wm=0; wm<WPT; wm++) {
                regA = sA[k][row + wm*RTS];

```

```
56                  for (int wn=0; wn<WPT; wn++) {
57                      value[wm][wn] += regA * regB[wn];
58                  }
59              }
60          }
61          // Sincronizar hasta que todos los hilos ejecuten el calculo
62          barrier(CLK_LOCAL_MEM_FENCE);
63      }
64      // Guardar el resultado final en la matriz de salida
65      for (int wm=0; wm<WPT; wm++) {
66          for (int wn=0; wn<WPT; wn++) {
67              if((globalRow + wm * RTS) < M && (globalCol + wn * RTS) < N)
68                  C[(globalRow + wm * RTS) * N + (globalCol + wn * RTS)] = value[
                        wm][wn];
69          }
70      }
71  }
```

Listing 3.5: Código de la función matrixMultiplication con incremento del trabajo por hilo

Debido a que se ha realizado la optimización a nivel de registros, destacaremos que en la sección del código relacionada con la carga de datos en memoria local, solo se ha modificado para que un hilo realice en cada dimensión una carga de WPT valores por submatriz en vez de solo una. Mientras que en la escritura de datos en memoria global de la matriz de salida, utilizaremos un doble bucle para almacenar este incremento de datos. A continuación, nos centraremos en las partes del código en los que se utilizan los registros.

En primer lugar, la asignación del espacio en los registros ha sido modificada para admitir más valores por hilo, la variable *value* ha pasado de almacenar un valor a $WPT \cdot WPT$ elementos y han sido creados los registros $regA$ y $regB$. Esto nos obliga a modificar la carga y cálculo de estos registros haciéndolos más complejos, ya que pasamos de una operación de multiplicación-suma fusionadas dentro de un bucle a emplear WPT operaciones FMA por cada cuatro bucles.

En la sección de cálculo, podemos visualizar que se mantiene el bucle exterior que recorre todo el bloque hasta obtener el resultado final. El cambio más importante está dentro de esta instrucción, ya que por cada submatriz se realizarán las siguientes acciones:

- Almacenar WPT valores correspondientes del bloque B en el registro $regB$.
- Almacenar un valor correspondiente del bloque A en el registro $regA$ y ejecutar el cálculo FMA WPT veces hasta completar una fila de la matriz de registros. Para llenar totalmente esta estructura, necesitaremos ejecutar estas instrucciones tantas veces como filas tenga la matriz de registros, en este caso WPT.

Como podemos ver, aunque hemos aumentado la complejidad del código, se han reducido la cantidad de accesos a memoria local por un factor de WPT, logrando de está manera una menor latencia.

A continuación, analizaremos el impacto de este método de optimización producido por tamaño de trabajo por hilo, manteniendo la proporción con el valor TS para tener en la constante RTS el valor óptimo de elementos por grupo de trabajo conseguido en la anterior optimización. Para ello, veremos el siguiente diagrama 3.9.

Como podemos comprobar, el aumento del trabajo por hilo y del tamaño de la submatriz obtiene mejores resultados cuando el tamaño de la topología y de las matrices sean más grandes. Esto se debe a que, al tener un bloque de memoria local TS mayor, el

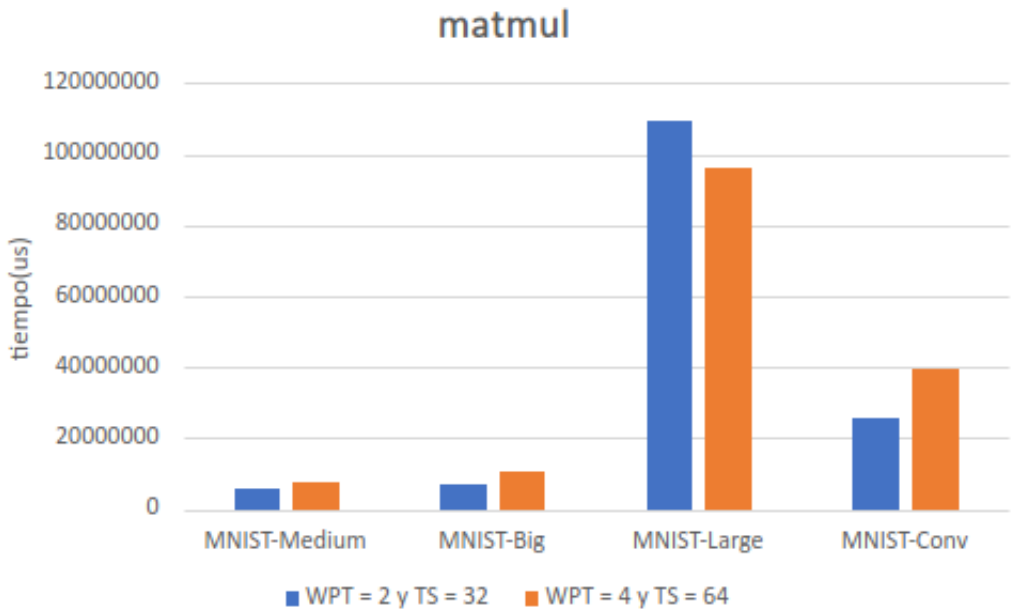

Figura 3.9: Tiempo de ejecución con diferentes valores de trabajo por hilo

cálculo para obtener un valor divisible en las matrices de entrada aumentará también, redimensionando estas matrices. De esta manera, en el momento de la ejecución estaremos procesando más elementos que ocupan valores de relleno. Por este motivo, solo conseguimos una mejora en la topología *MNIST-Large*, mientras que en las redes de menor tamaño aumenta el tiempo de ejecución necesario para la obtención del resultado.

Por último, compararemos con la configuración de $TS = 64$ y $WPT = 4$ con la optimización sin incremento de hilos y con un tamaño de memoria de 16 (ver figura 3.10).

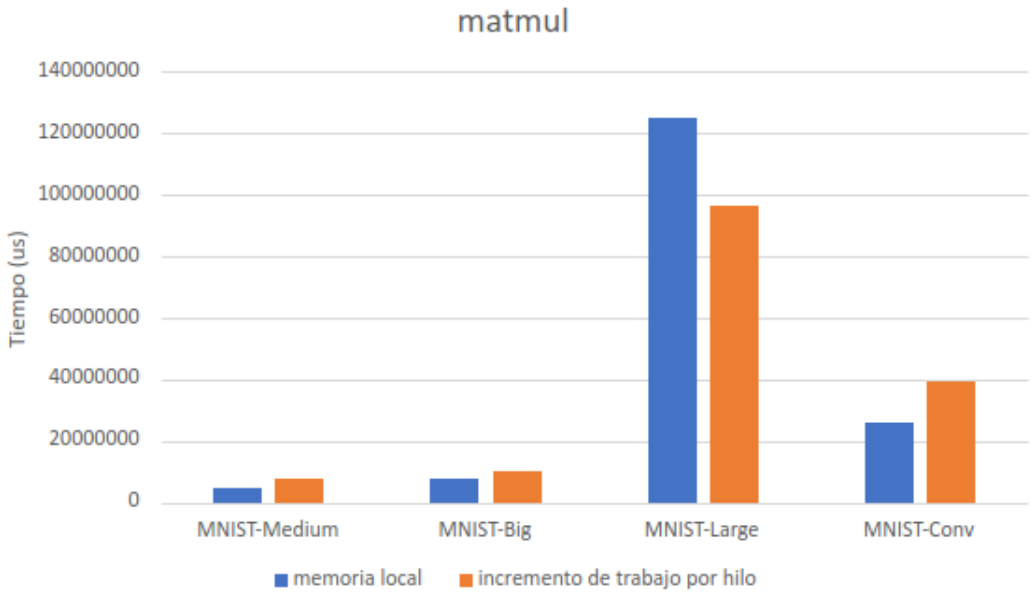

Figura 3.10: Comparación tiempo de ejecución entre las optimizaciones con memoria local e incremento de trabajo por hilo

Como podemos visualizar, menos en la topología *MNIST-Large* obtenemos mejores resultados con la optimización de solo la memoria local. Esto se corresponde a lo visto en el diagrama 3.9, en donde un mayor tamaño de bloque solo es aprovechado si el tamaño de las matrices es elevado. Aun así y a pesar de los valores finales obtenidos, optamos por la versión del incremento de trabajo por hilo con la configuración $TS = 64$ y $WPT = 4$ debido a que vamos a evaluar en el siguiente capitulo el rendimiento que obtiene contra la librería clBLAS en redes de mayor tamaño y requerimiento de cálculo.

3.3.6. Funciones optimizadas

Finalmente, la utilización de la memoria local y el incremento de hilos solo proporcionarán una mejora de rendimiento si la función que vayamos a optimizar reutilice constantemente los datos como es el caso de las multiplicaciones de matrices (GEMM), consideradas de nivel tres por la especificación BLAS debido a su resolución en un tiempo de coste cúbico [33]. De otro modo, el coste que tiene el desplazar la información hacia la memoria local y el aumento de su complejidad computacional para realizar el correspondiente cálculo hacen que el uso de las técnicas anteriormente desarrolladas consigan una perdida de rendimiento elevando de esta manera el tiempo necesario para la obtención del resultado. Teniendo en cuenta esto, hemos optimizado las funciones que podemos ver en la tabla 3.6.

Función	Tipo de optimización
matmul	Incremento de trabajo por hilo
matmul_at	Incremento de trabajo por hilo
matmul_bt	Incremento de trabajo por hilo
mat_reduce_rows	Utilización de la memoria local
matrix_elwise	Utilización de la memoria local

Tabla 3.6: Funciones optimizadas e implementadas.

Todas estas funciones son empleadas en la sección siguiente de evaluación para la obtención de los resultados finales mediante la etiqueta «Optimización» cuando comparemos con la versión implementada para dar soporte a OpenCL sin haber utilizado ninguna técnica de mejora de rendimiento. De la misma forma, con la etiqueta «OpenCL» cotejamos la información obtenida contra la librería clBLAS.

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

Como hemos mencionado anteriormente, para los cálculos en la GPU mediante OpenCL se necesita que los datos estén almacenados en la memoria de este dispositivo. Para ello se ha necesitado implementar las funciones necesarias para llevar un control sobre la memoria de la GPU mediante OpenCL las cuales se pueden observar en la tabla 3.7.

3.4 Evaluación

El principal objetivo de HELENNA es el entrenamiento y clasificación de un conjunto de muestras sobre las diferentes topologías de redes neuronales. En tiempo de ejecución, la aplicación es capaz de mostrar y modificar el porcentaje de acierto, mostrando de esta manera si la red neuronal se está entrenando correctamente o no. Para el cálculo del tiempo que tarda el programa en ejecutar al completo el aprendizaje de la red neuronal utilizaremos el comando *time*. Mientras que los parámetros utilizados para la obtención

Función	Descripción
fn_allocate_buffer_opencl_gpu	Realiza la asignación dinámica de memoria en la GPU
fn_deallocate_buffer_opencl_gpu	Libera la región de memoria de la GPU asociada al puntero
fn_read_buffer_opencl_gpu	Copia desde los datos desde la GPU hacia la CPU
fn_value_opencl_gpu	Copia un elemento de la matriz indicada desde la GPU hacia la CPU
fn_write_buffer_opencl_gpu	Copia los datos desde la CPU hacia la GPU

Tabla 3.7: Funciones implementadas relacionadas con el soporte de *buffers*.

de todos los resultados son los representados en la tabla 3.3 junto con sus correspondientes valores.

A continuación, en la figura 3.11 vemos la ejecución en HELENNA de la versión base implementada para dar soporte a OpenCL de color azul y su optimización con las funciones descritas en la tabla 3.6 de color naranja. Como muestra, se utilizan las topologías densamente conectadas que utilizan los conjuntos de datos MNIST.

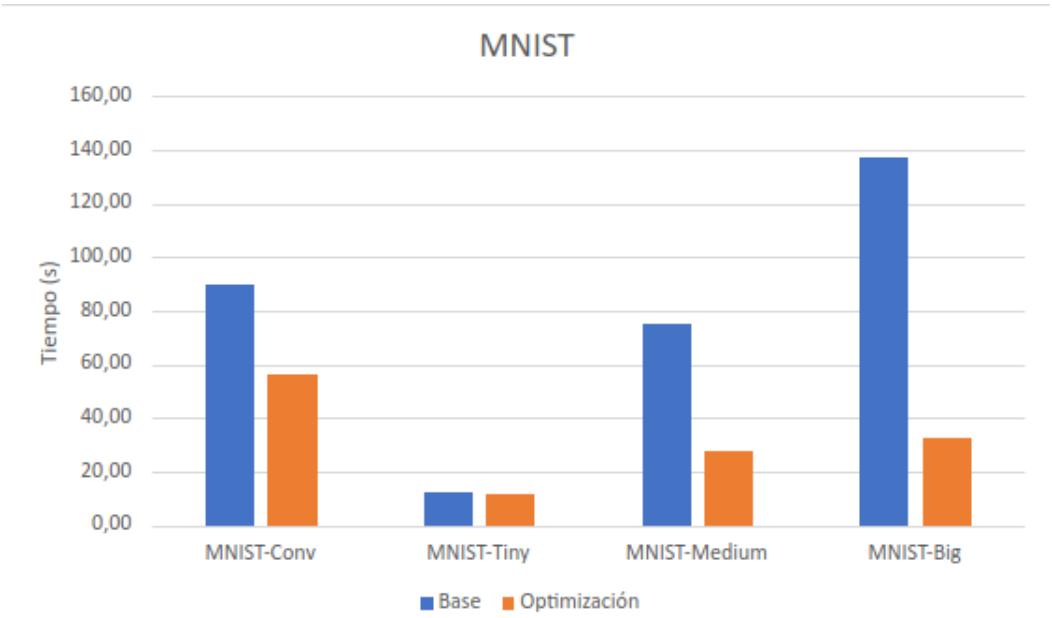

Figura 3.11: Comparación tiempo de ejecución entre la versión en OpenCL y la versión final de su optimización

Como se puede apreciar, conseguimos una reducción importante del tiempo cuando el tamaño de la topología es mayor y se tiene que utilizar un mayor número de cálculos para conseguir el resultado final. Adicionalmente, si vemos el diagrama 3.12 correspondiente a la precisión de acierto alcanzada por ambas versiones, obtendremos un porcentaje similar. De esta manera, con la optimización realizada conseguimos entrenar los distintos modelos de una forma más rápida sin perder precisión.

Por último, en la figura 3.13 observamos la aceleración conseguida con respecto a la versión base. Que comprende entre las topologías de menor tamaño donde apenas se consigue una mejora de rendimiento hasta obtener unos resultados cuatro veces más rápido en las de mayor dimensión.

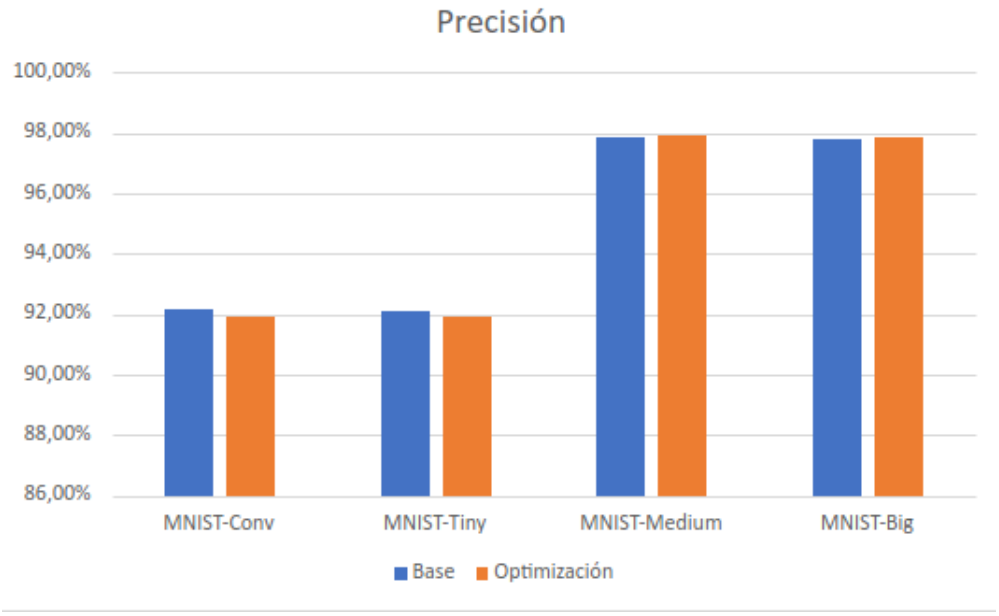

Figura 3.12: Comparación del porcentaje de precisión entre la versión en OpenCL y la versión final de su optimización

A continuación, entramos en detalle en las topologías *MNIST-Big* y *MNIST-Conv* para ver el impacto que tienen las funciones optimizadas en las redes neuronales densamente conectadas y convolucionales respectivamente. Para ello, vemos como la figura 3.14 correspondiente a una topología densamente conectada, la función *matmul* es donde se consigue una clara reducción del tiempo de ejecución.

Mientras que en el diagrama 3.15, la topología *MNIST-Conv* (convolucional), apenas conseguimos una mejora de rendimiento en la multiplicación de matrices. A pesar de ello, en la función *mat_reduce_rows* podemos ver como reduce significativamente el tiempo de ejecución. Esto se debe a la topología que estemos utilizando, ya que los tamaños de matriz que se calculan en estas funciones varían y cuanto mayor sea este tamaño, mejor rendimiento obtendremos con la optimización realizada.

Debido a que las topologías que emplean el conjunto de datos CIFAR10 tienen un tamaño y complejidad mayores que los de MNIST, vamos a ejecutar y comparar la optimización implementada en este trabajo con la librería clBLAS. Debido a que, si realizamos la comparación con la versión base, como venimos haciendo hasta ahora, para obtener el resultado de la ejecución tardaremos horas en conseguirlo.

Una vez explicada la razón de la realización de la comparación entre la optimización a mano y la librería clBLAS, visualizaremos en las figuras 3.16 y 3.17 los tiempos de ejecución en las topologías MNIST y CIFAR10 correspondientes.

Como podemos observar, los resultados obtenidos en clBLAS en todas las topologías menos en las convolucionales que utilizan el conjunto de datos de MNIST, son mejores al obtener el resultado en un tiempo de ejecución menor con respecto a la optimización empleada a mano.

Mientras, los resultados con respecto a la precisión de acierto en las topologías ejecutadas son similares en ambas implementaciones. Teniendo en cuenta que la diferencia entre los entrenamientos producidos en MNIST, acierto superior al 90 %, con respecto a los de CIFAR10 se deben a las razones ya comentadas al principio de la sección.

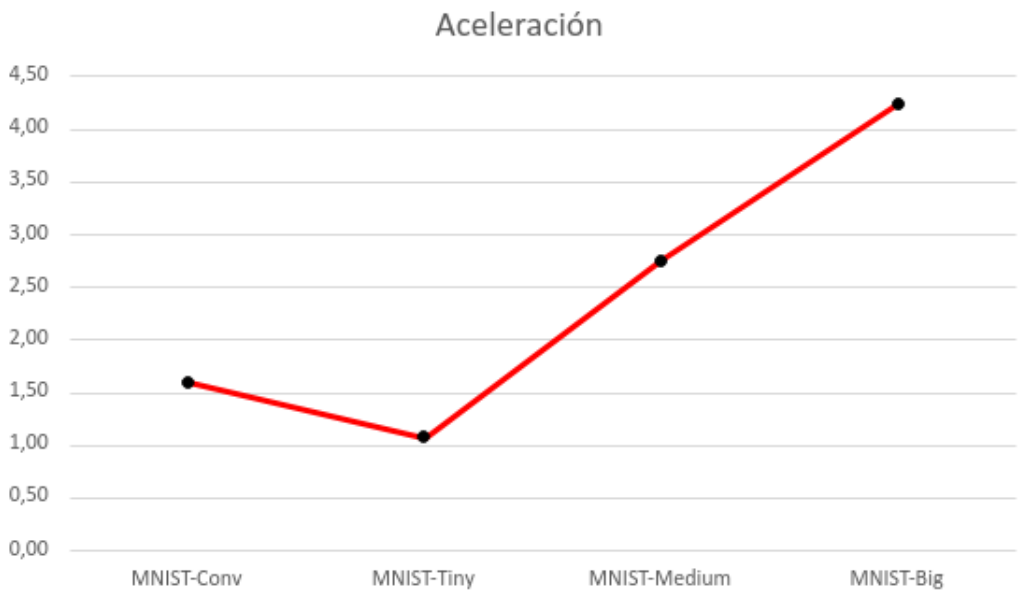

Figura 3.13: Aceleración obtenida entre las versiones base y optimizada de OpenCL

Finalmente, si bien los resultados conseguidos en clBLAS son mejores en cuanto al tiempo de ejecución necesario para completar el entrenamiento, logramos una aproximación a esta librería mediante las técnicas de optimización realizadas y el ajuste de la aplicación a la arquitectura utilizada. Obteniendo de esta manera un uso eficiente de los recursos en aquellos dispositivos en los cuales no se puede utilizar correctamente esta librería, como sistemas empotrados, FPGAs o GPUs de bajo consumo.

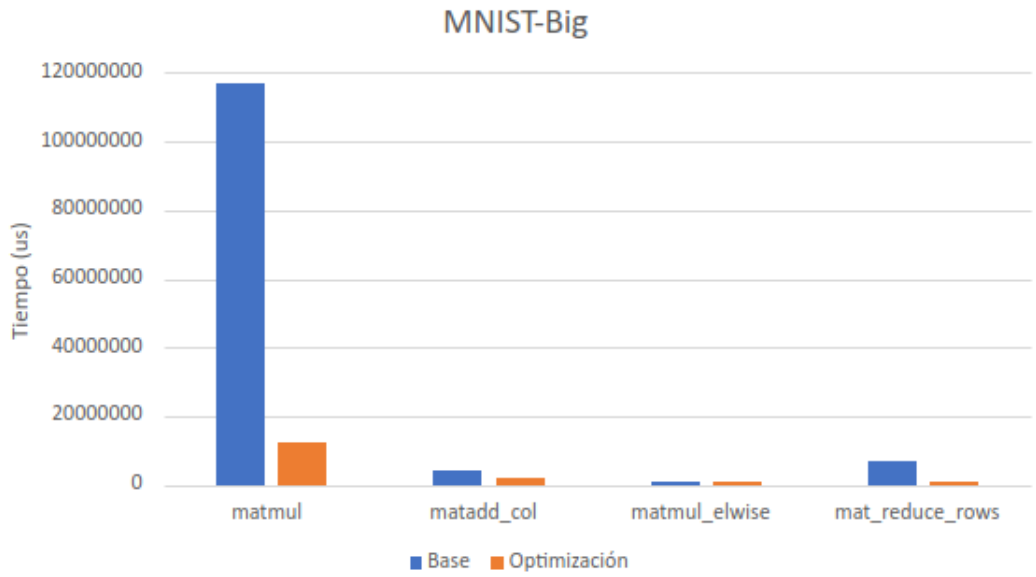

Figura 3.14: Comparación del tiempo de ejecución entre las funciones optimizadas y sus versiones base en la topología *MNIST-Big*

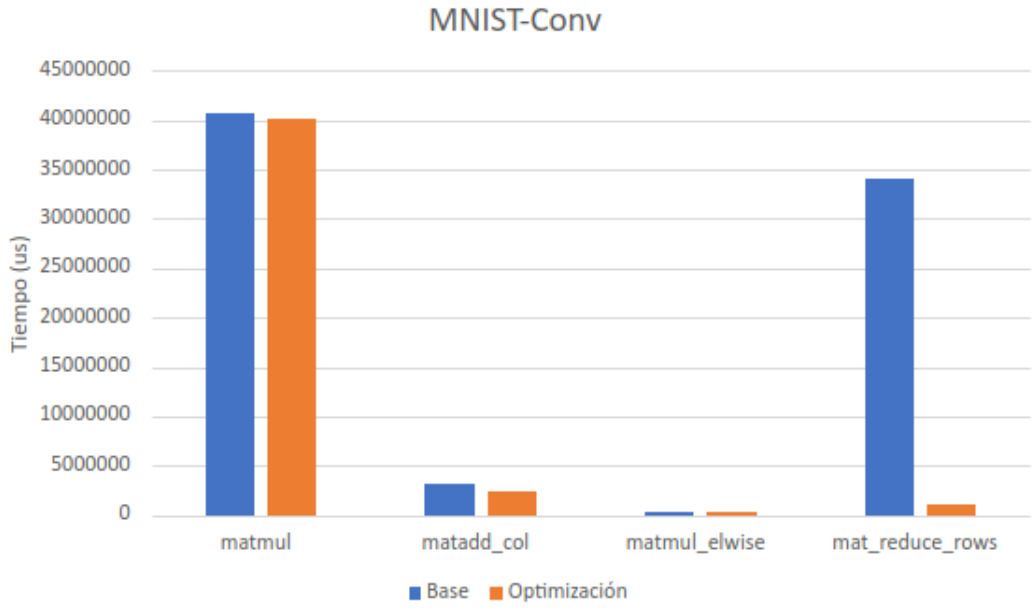

Figura 3.15: Comparación del tiempo de ejecución entre las funciones optimizadas y sus versiones base en la topología *MNIST-Conv*

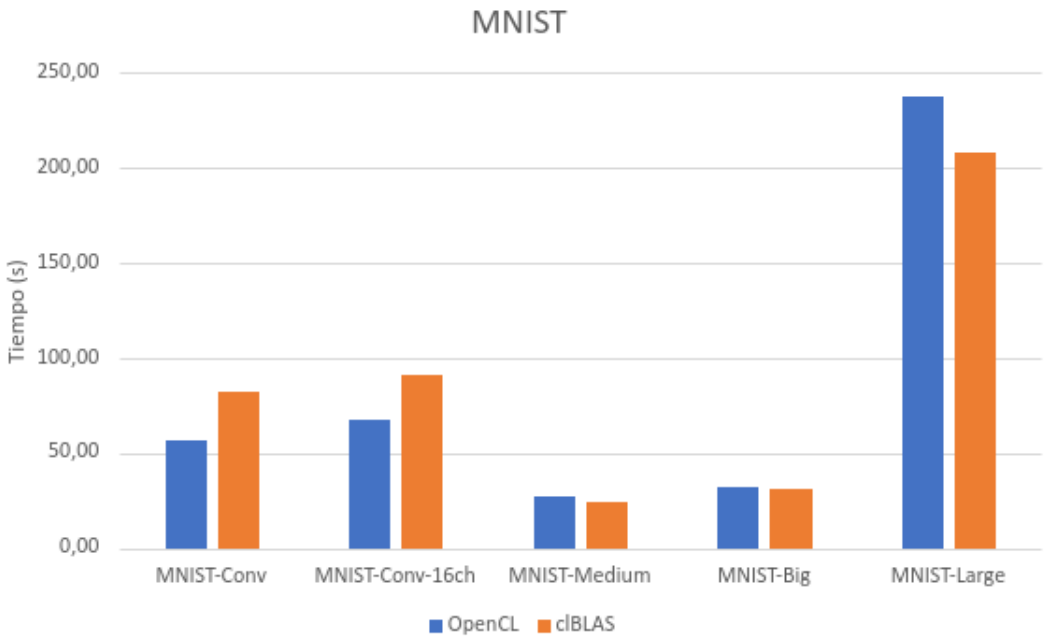

Figura 3.16: Comparación del tiempo de ejecución entre las optimización a mano y clBLAS en las topologías que utilizan el conjunto de datos MNIST

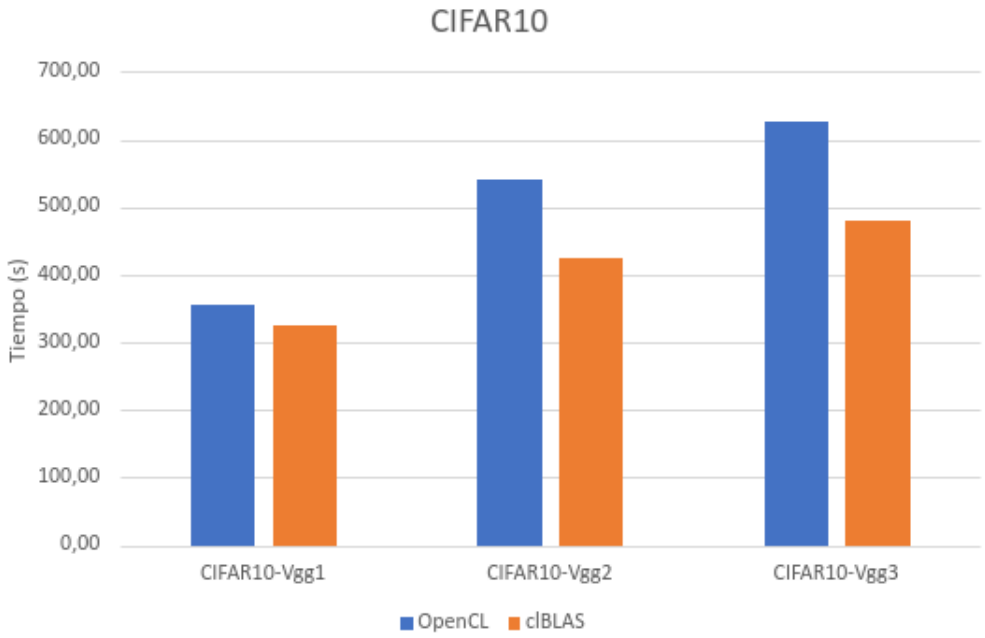

Figura 3.17: Comparación del tiempo de ejecución entre las optimización a mano y clBLAS en las topologías que utilizan el conjunto de datos CIFAR

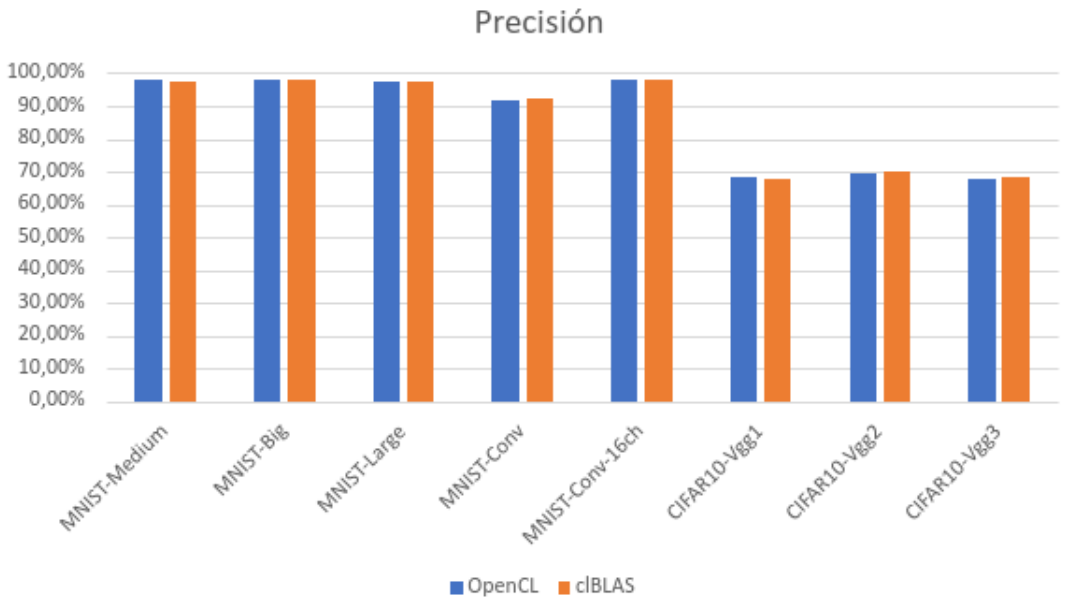

Figura 3.18: Comparación de la precisión entre las optimización a mano y clBLAS en las topologías anteriormente ejecutadas

CAPÍTULO 4
Conclusiones

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

- Se ha hecho un estudio de la microarquitectura GCN, la cual es la utilizada en el dispositivo de este trabajo para conseguir una optimización *ad hoc* de este componente.
- 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 estudiado en profundidad el modelo de programación de OpenCL así como también de diferentes técnicas de optimización en este *framework*, consiguiendo implementar estos métodos logrando una reducción significativa del tiempo de ejecución en las funciones relativas al producto de matrices.
- Se ha dado soporte con el *framework* OpenCL a la aplicación HELENNA para ejecutar sobre GPU cálculos complejos y funciones relativas al entrenamiento de las capas convolucionales y densamente conectadas.

4.1 Relación del trabajo desarrollado con los estudios cursados

En el presente TFG, se han necesitado conceptos de las siguientes asignaturas:

- Álgebra, ya que se han requerido los conocimientos adquiridos en esta asignatura, tanto para el desarrollo de funciones que utilizan álgebra lineal, así como también en el desarrollo de la optimización de la función de la multiplicación de matrices.
- Computación Paralela por la introducción a los sistemas que emplean modelos de programación paralela para razonar, analizar y desarrollar soluciones que abarcan procesos multihilo y utilización de memoria compartida para implementar funciones para dispositivos GPUs en OpenCL.
- Las asignaturas de Arquitectura e Ingeniería de Computadores y Arquitecturas Avanzadas, de las cuales se han necesitado conocimientos avanzados relacionados con la arquitectura de las unidades de procesamiento gráfico. Tanto en la distinción de los diferentes componentes destinados al cálculo computacional como también de las distintas memorias empleadas en este trabajo.

- Sistemas Inteligentes en la introducción a las redes neuronales, consiguiendo así una mejor comprensión de los problemas, conceptos y resoluciones ligados a este ámbito.

CAPÍTULO 5
Trabajo futuro

En este último capítulo, expondremos las mejoras e implementaciones que se pueden realizar a futuro en las soluciones explicadas en este TFG.

5.1 Bloques en memoria local de tamaño rectangular

Una de las técnicas de optimización que podemos utilizar son los bloques en memoria local de tamaño rectangular, los cuales nos permitirán tener una mejor flexibilidad para ajustar los valores del bloque dependiendo del tamaño de matriz y del espacio en memoria que disponemos.

5.2 Optimizaciones a funciones

Funciones como *matadd_col* y *im2col*, tienen un costo elevado de ejecución, esto pueden ser reducido mediante la optimización de las técnicas desarrolladas en este trabajo. Mejorando así el rendimiento el tiempo de ejecución, especialmente en redes convolucionales, las cuales utilizan los dos métodos.

5.3 Soporte a las capas de *batch normalization* y *dropout*

Dar soporte a la capa *dropout*, ya que HELENNA emplea topologías de redes neuronales con la técnica de regularización *dropout*. Esta capa reduce el sobre ajuste y mejorar el error de generalización de una red neuronal mediante la desactivación aleatoria de neuronas durante el entrenamiento.

Dar soporte a la capa de *batch normalization*, utilizada en HELENNA para el entrenamiento de redes neuronales. Esta técnica se utiliza para estandarizar las entradas a las capas por cada *batch*, acelerando el entrenamiento y proporcionando regularización. Por lo que reduce el error de generalización.

Bibliografía

[1] Mohamed Meselhi, Saber Elsayed, Daryl Essam y col. «Fast differential evolution for big optimization». En: dic. de 2017, págs. 1-6. DOI: 10.1109/SKIMA.2017.8294137.

[2] Luis Salcedo. *Introducción a las Redes Neuronales - Parte #1: Elementos básicos de una Red Neuronal*. URL: http://www.pythondiario.com/2018/07/introduccion-las-redes-neuronales-parte_12.html (visitado 2020-08-29).

[3] Felipe Ardila. «Predicción de niveles del río Magdalena usando sistemas adaptativos basados en conocimiento». Tesis doct. Ene. de 2009.

[4] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. URL: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e (visitado 2020-08-31).

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

[6] Justin Johnson. *cnn-benchmarks*. URL: https://github.com/jcjohnson/cnn-benchmarks#resnet-18 (visitado 2020-08-31).

[7] *The OpenCL™ Specification*. Version 1.2-19. Khronos® OpenCL Working Group. Nov. de 2012.

[8] Khronos. *Khronos OpenCL Registry*. URL: https://www.khronos.org/registry/OpenCL/ (visitado 2020-06-16).

[9] Inc. Advanced Micro Devices. *Welcome to AMD ROCm Platform*. URL: https://rocmdocs.amd.com/en/latest/ (visitado 2020-08-17).

[10] Inc. Advanced Micro Devices. *AMD ROCm Profiler*. URL: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html (visitado 2020-08-17).

[11] Inc. Advanced Micro Devices. *AMD ROCm Debbuger*. URL: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCgdb.html (visitado 2020-08-17).

[12] Intel. *Intel® Developer Zone*. URL: https://software.intel.com/content/www/us/en/develop/home.html (visitado 2020-08-30).

[13] NVIDIA. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone (visitado 2020-08-23).

[14] Cedric Nugteren. *Tutorial: OpenCL SGEMM tuning for Kepler*. URL: https://cnugteren.github.io/tutorial/pages/page1.html (visitado 2019-11-20).

[15] clMathLibraries. *clBLAS*. URL: https://github.com/clMathLibraries/clBLAS (visitado 2020-08-27).

[16] *AMD APP SDK OpenCL™ User Guide*. Version 1.0. Advanced Micro Devices, Inc. Ago. de 2015.

[17] Robert Leroy Kruse. *Data structures and program design*. Englewood Cliffs, N.J. : Prentice-Hall, 1987. ISBN: 0131958844.

[18] James Smith y Andrew Pleszkun. «Implementation of Precise Interrupts in Pipelined Processors.» En: vol. 13. Jun. de 1985, págs. 36-44. DOI: 10.1145/285930.285988.

[19] Johannes Rudolph. *OpenCL Work Item Ids: Global/Group/Local*. URL: https://jorudolph.wordpress.com/2012/02/03/opencl-work-item-ids-globalgrouplocal/ (visitado 2020-07-30).

[20] *Reference Guide Graphics Core Next Architecture, Generation 3*. Version 1.1. Advanced Micro Devices, Inc. Ago. de 2016.

[21] Sven-Bodo Scholz. *Heterogeneous Computing using openCL lecture 4 F21DP Distributed and Parallel Technology*. URL: https://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/sbsl04.pdf (visitado 2020-08-28).

[22] mql5. *OpenCL: From Naive Towards More Insightful Programming*. URL: https://www.mql5.com/en/articles/407 (visitado 2020-08-27).

[23] Steve Burke. *AMD RX 480 8GB Review, Overclocking, & Exhaustive Benchmark*. URL: https://www.gamersnexus.net/hwreviews/2496-amd-rx-480-8gb-review-and-benchmark-vs-gtx-970-1070?_escaped_fragment_=/ccomment-comment=10005131 (visitado 2020-08-03).

[24] Abheek Gulati. *An Architectural Deep-Dive into AMD's TeraScale, GCN & RDNA GPU Architectures*. URL: https://medium.com/high-tech-accessible/an-architectural-deep-dive-into-amds-terascale-gcn-rdna-gpu-architectures-c4a212d0eb9 (visitado 2020-08-03).

[25] *H2020 RECIPE*. URL: http://www.recipe-project.eu/ (visitado 2020-08-14).

[26] *Jetson AGX Xavier*. URL: https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/ (visitado 2020-08-14).

[27] *Cloud Tensor Processing Unit (TPU)*. URL: https://cloud.google.com/tpu/docs/tpus?hl=es-419 (visitado 2020-08-14).

[28] Corinna Cortes Yann LeCun y Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/ (visitado 2020-07-28).

[29] Vinod Nair Alex Krizhevsky y Geoffrey Hinton. *The CIFAR-10 dataset*. URL: https://www.cs.toronto.edu/~kriz/cifar.html (visitado 2020-07-28).

[30] CH.Tseng. *Neural Networks Keras MNIST*. URL: https://chtseng.wordpress.com/2017/07/31/neural-networks-%E4%BA%8C-keras-mnist/ (visitado 2020-07-31).

[31] Parneet Kaur. *Convolutional Neural Networks (CNN) for CIFAR-10 Dataset*. URL: http://parneetk.github.io/blog/cnn-cifar10/ (visitado 2020-07-31).

[32] K. Matsumoto, N. Nakasato y S. G. Sedukhin. «Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs». En: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, págs. 396-405.

[33] Netlib. *BLAS (Basic Linear Algebra Subprograms)*. URL: http://www.netlib.org/blas/ (visitado 2020-08-22).

APÉNDICE A

Topologías utilizadas en el entrenamiento de redes neuronales

A continuación, veremos las topologías utilizadas en el entrenamiento de redes neuronales para los resultados vistos en los capítulos tres y cuatro. Cada una de las imágenes indica en la lista de dispositivos cual será la unidad que va a realizar el entrenamiento, para este trabajo se utilizará en todas las pruebas el valor *opencl-gpu* que representa la ejecución del modelo de programación OpenCL en GPU. Además, cada topología hace un sumario de las diferentes capas que la compone, su tipo, el número de neuronas que contiene y su configuración. Por último, cada una de estas topologías se ejecuta con las mismas configuraciones que podemos visualizar en la figura A.1.

```
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   5, dynamic   0                                                                     |
| MPI: no - 1 processes                                                                          |
| Force synchronization (for cublas and clblas): no                                              |
|------------------------------------------------------------------------------------------------|
```

Figura A.1: Configuración utilizada en las topologías

```
|devices
|
| mkl         | avx         | avx512      | opencl-fpga | opencl-gpu  | cublas      | clblas      |
|             |             |             |             | yes         |             |             |
| OMP: CPUs   5, dynamic   0
| MPI: no - 1 processes
| Force synchronization (for cublas and clblas): no
```

```
|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 A.2: Topología MNIST-Medium

```
|devices
|
| mkl         | avx         | avx512      | opencl-fpga | opencl-gpu  | cublas      | clblas      |
|             |             |             |             | yes         |             |             |
| OMP: CPUs   5, dynamic   0
| MPI: no - 1 processes
| Force synchronization (for cublas and clblas): no
```

```
|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 A.3: Topología MNIST-Big

```
|devices
|
| mkl         | avx         | avx512      | opencl-fpga | opencl-gpu  | cublas      | clblas      |
|             |             |             |             | yes         |             |             |
| OMP: CPUs   5, dynamic   0
| MPI: no - 1 processes
| Force synchronization (for cublas and clblas): no
```

```
|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 A.4: Topología MNIST-Large

```
devices
| mkl | avx | avx512 | opencl-fpga | opencl-gpu | cublas | clblas |
|     |     |        |             | yes        |        |        |

OMP: CPUs  5, dynamic  0
MPI: no - 1 processes
Force synchronization (for cublas and clblas): no

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|maxpl_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 |maxpl_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 A.5: Topología CIFAR10-VGG1

```
-----------------------------------------------------------------------------------------------|
|devices                                                                                       |
|----------------------------------------------------------------------------------------------|
| mkl         | avx         | avx512      | opencl-fpga | opencl-gpu  | cublas      | clblas      |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|             |             |             |             | yes         |             |             |
|----------------------------------------------------------------------------------------------|
| OMP: CPUs   5, dynamic   0                                                                   |
| MPI: no - 1 processes                                                                        |
| Force synchronization (for cublas and clblas): no                                            |
|----------------------------------------------------------------------------------------------|

|-------------------------------------------------------------------------------------------------------------------------------------------------------|
|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|conv3_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|conv3_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|fc1_0      |fully connected         |      128|    524416|maxp2_0   |inputs 4096, outputs 128                                            |     0.01 GB|
|   12|relu5_0    |relu                    |      128|         0|fc1_0     |                                                                    |     0.00 GB|
|   13|fc2_0      |fully connected         |       10|      1290|relu5_0   |inputs 128, outputs 10                                              |     0.00 GB|
|   14|softmax_0  |softmax                 |       10|         0|fc2_0     |                                                                    |     0.00 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------|
|     |           |TOTAL                   |   209172|    591274|          |Memory (no layers):   0.00 GB                                       |     0.43 GB|
|-------------------------------------------------------------------------------------------------------------------------------------------------------|
```

Figura A.6: Topología CIFAR10-VGG2

```
devices

| mkl | avx | avx512 | opencl-fpga | opencl-gpu | cublas | clblas |
|     |     |        |             | yes        |        |        |

OMP: CPUs   5, dynamic  0
MPI: no - 1 processes
Force synchronization (for cublas and clblas): no

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 | | 550570 | | Memory (no layers): 0.00 GB | 0.50 GB |
```

Figura A.7: Topología CIFAR10-VGG3