UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

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

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Marc Gavilán Gil

Tutor: Flich Cardo, José
Museros Romero, Pedro

Curso 2019-2020

Resum

Aquest Treball de Fi de Grau té com a objectiu el desenvolupament de suport d'arquitectures heterogènies amb programació en OpenCL per a un suport eficient del càlcul en dos aplicacions actuals

La primera aplicació és una aplicació per al càlcul dinàmic d'estructures de ponts ferroviaris on, a més de la programació d'arquitectures heterogènies, s'ha optimitzat l'aplicació per a la seua execució en un sistema distribuït, empleat per a això una ferramenta de memòria distribuïda per mitjà de pas de missatges. Amb la dita ferramenta s'ha construït un sistema de comunicació que ha permés l'execució paral·lela de l'aplicació, a fi d'obtindre una millora en el cost temporal del càlcul.

El treball realitzat sobre la segona aplicació ha consistit en una extensió de la funcionalitat en el càlcul de l'entrenament de xarxes neuronals per a permetre l'execució d'aquesta en plataformes heterogènies. En concret hem utilitzat la plataforma GPU per a dur a terme les execucions.

En ambdós casos s'ha obtingut una millora substancial del temps d'execució a l'utilitzar dispositius GPU d'alt rendiment. La implementació realitzada és compatible i pot ser utilitzada en altres tipus de dispositius heterogenis, principalment FPGAs.

Paraules clau: OpenCL, GPU, clBLAS, Arquitectures Heterogènies, MPI, Xarxes Neuronals

Resumen

Este Trabajo de Fin de Grado tiene como objetivo el desarrollo de soporte de arquitecturas heterogéneas con programación en OpenCL para un soporte eficiente del cálculo en dos aplicaciones actuales.

La primera aplicación es una aplicación para el cálculo dinámico de estructuras de puentes ferroviarios donde, además de la programación de arquitecturas heterogéneas, se ha optimizado la aplicación para su ejecución en un sistema distribuido, empleado para ello una herramienta de memoria distribuida mediante paso de mensajes. Con dicha herramienta se ha construido un sistema de comunicación que ha permitido la ejecución paralela de la aplicación, al objeto de obtener una mejora en el coste temporal del cálculo.

El trabajo realizado sobre la segunda aplicación ha consistido en una extensión de la funcionalidad en el cálculo del entrenamiento de redes neuronales para permitir la ejecución de la misma en plataformas heterogéneas. En concreto hemos utilizado la plataforma GPU para llevar a cabo las ejecuciones.

En ambos casos se ha obtenido una mejora sustancial del tiempo de ejecución al utilizar dispositivos GPU de alto rendimiento. La implementación realizada es compatible y puede ser utilizada en otros tipos de dispositivos heterogéneos, principalmente FPGAs.

Palabras clave: OpenCL, GPU, clBLAS, Arquitecturas Heterogéneas, MPI, Redes Neuronales

Abstract

In this project we have developed support for heterogeneous architectures by using OpenCL programming and to provide efficient computing capabilities to two key representative applications.

The first application targets modeling of train bridges. For this application, we additionally provide an optimization to enable distributed computing of such application, using a distributed memory approach and message passing support through MPI. This enables the application to speedup the simulation process of the bridgea in a cluster of heterogeneous devices.

The second application targets neural network training and inference. In such application we have coded all the necesary funcionality to support OpenCL programming of GPUs for the improvement of the application (compared to CPU).

In both applications we have achieved remarkable execution time speedups by using high-performance GPUs. The provided implementation is compatible with other heterogeneous sysems, such as FPGAs.

Key words: OpenCL, GPU, clBLAS, Heterogeneous Architectures, MPI, Neural Networks

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Sistemas de computación heterogéneos

Los sistemas de computación tradicionalmente utilizan los procesadores (CPUs) de forma nativa. El uso de procesadores comerciales (Intel, AMD, ...) así como específicos diseñados por la grandes empresas (Intel, IBM, ...) suelen utilizarse cuando se abordaban problemas computacionalmente elevados. Ahora bien, el consumo excesivo de dichos procesadores y su poca eficiencia en algunos modelos de computación, ha ocasionado un cambio de paradigma hacia los sistemas heterogéneos.

Los sistemas de computación heterogéneos ofrecen una ventaja crucial frente al cálculo habitual en CPU. La computación heterogénea permite dividir el cómputo en diversas unidades de procesamiento de diferente tipología, aprovechando al máximo las características de cada una de ellas para así obtener un resultado superior al original.

En los sistemas de computación heterogéneos la unidad de procesamiento principal siempre es la CPU pero existen varios tipos de arquitecturas auxiliares a esta que se utilizan para acelerar determinado tipo de cálculo. Tales como GPUs, FPGAs[14], DSPs[18].

En este proyecto nos centramos en un tipo de arquitectura concreta: La unidad de procesamiento gráfico o también conocida por sus siglas en inglés GPU (*graphics processing unit*). La ventaja que ofrece esta arquitectura es su gran capacidad de paralelismo gracias al elevado número de *cores* de procesamiento que incorpora. Cada uno de ellos por separado no iguala la capacidad de cálculo equivalente a un *core* de CPU. Pero mientras una CPU de gama media puede contar del orden de decenas de *cores*, una GPU equivalente cuenta con una cifra que ronda los 1000 o 2000.

De este modo las tareas que son paralelizables, es decir descomponibles en subtareas idénticas mas sencillas, son encargadas a la GPU y el resto del programa continua siendo llevado a cabo por la CPU de manera convencional. En este proyecto mostraremos diversos ejemplos de estas tareas paralelizables.

Uno de los principales motivos por el que se apuesta por la programación heterogénea es la eficiencia energética. En la figura 1.1 se puede observar que la potencia de cálculo en arquitecturas heterogéneas está un orden de magnitud por encima de las obtenidas por las CPUs. Mientras que por otro lado el consumo de las mismas no aumenta de la misma forma (Figura 1.2). Por lo tanto, estas arquitecturas son mas eficientes desde el punto de vista energético.

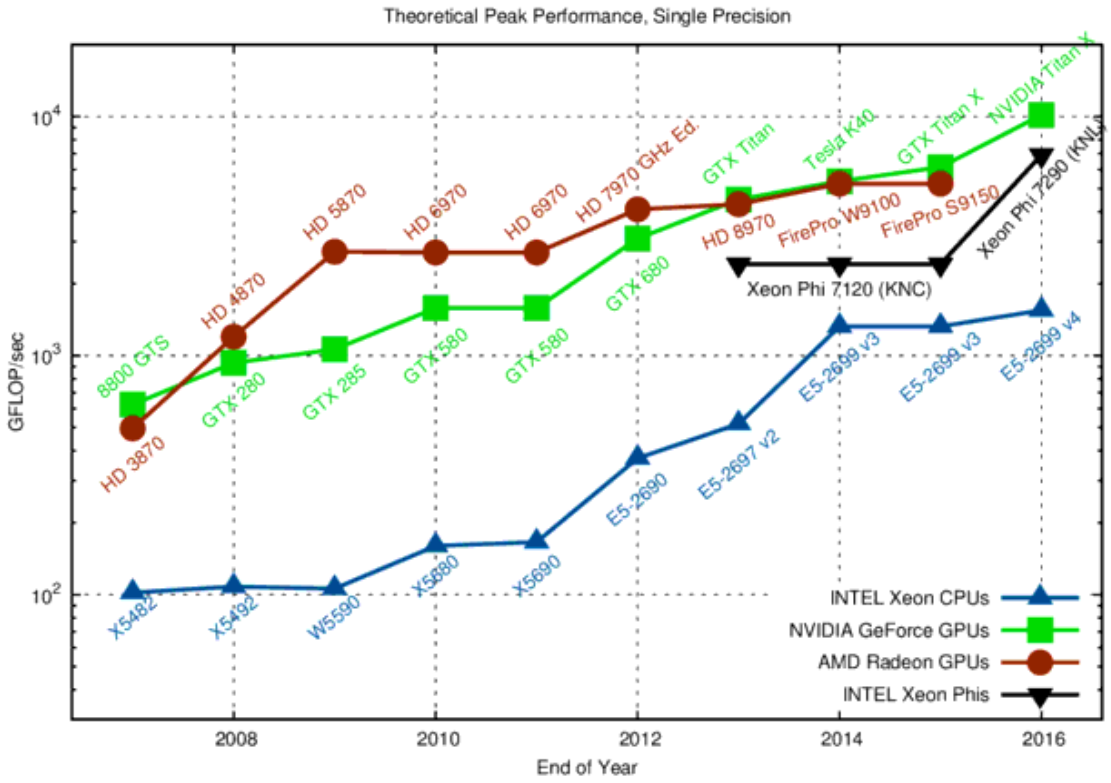


Figura 1.1: Comparativa de la evolución de la potencia de cálculo en FLOPS entre diversas plataformas a lo largo del tiempo. [27]

1.2 Motivación

En la actualidad existen múltiples dominios de aplicación donde el uso de GPUs es indispensable. El desarrollo de herramientas, librerias y aplicaciones que aporten un rendimiento en GPUs acorde con la evolución de la tecnología es crucial en estos conextos. Por ejemplo podemos ver el soporte en Inteligencia Artificial facilitado por Tensor Flow, Caffe, Keras .... Así como librerías para el uso de las GPUs en aplicaciones matemáticas como son BLAS, CLBLAS, ...

Sin embargo, hay campos de aplicación donde el uso de arquitecturas paralelas como las GPUs no es frecuente y su uso se circunscribe a contextos muy reducidos. Este es el caso en el campo de la simulación numérica de estructuras ferroviarias, por ejemplo. De hecho, **en este proyecto abordaremos una aplicación típica de cálculo dinámico de puentes ferroviarios y le incorporaremos el uso de GPUs** para su mejor eficiencia en el cálculo.

Además, aunque hay dominios de uso con mejor soporte de GPUs, también existe necesidad de nuevas adaptaciones. Por ejemplo, el uso de GPUs en Inteligencia Artificial está muy extendido con el uso de TensorFlow y Caffe. Ahora bien, hay nuevas iniciativas, tanto docentes como de investigación que precisan de un soporte de uso de GPUs eficiente. Este es el caso de HELENNA, un motor de entrenamiento e inferencia desarrollado en la UPV. **En este proyecto incorporamos soporte específico de GPUs en HELENNA**.

En relación al uso de arquitecturas heterogéneas, mas allá del uso exclusivo de GPUs, existe la necesidad de soportar otras arquitecturas. El caso mas evidente es el uso de FPGAs. En este sentido, utilizar modelos de programación compatibles entre todos los tipos de dispositivos es interesante y necesario. OpenCL es uno de esos modelos de programa-

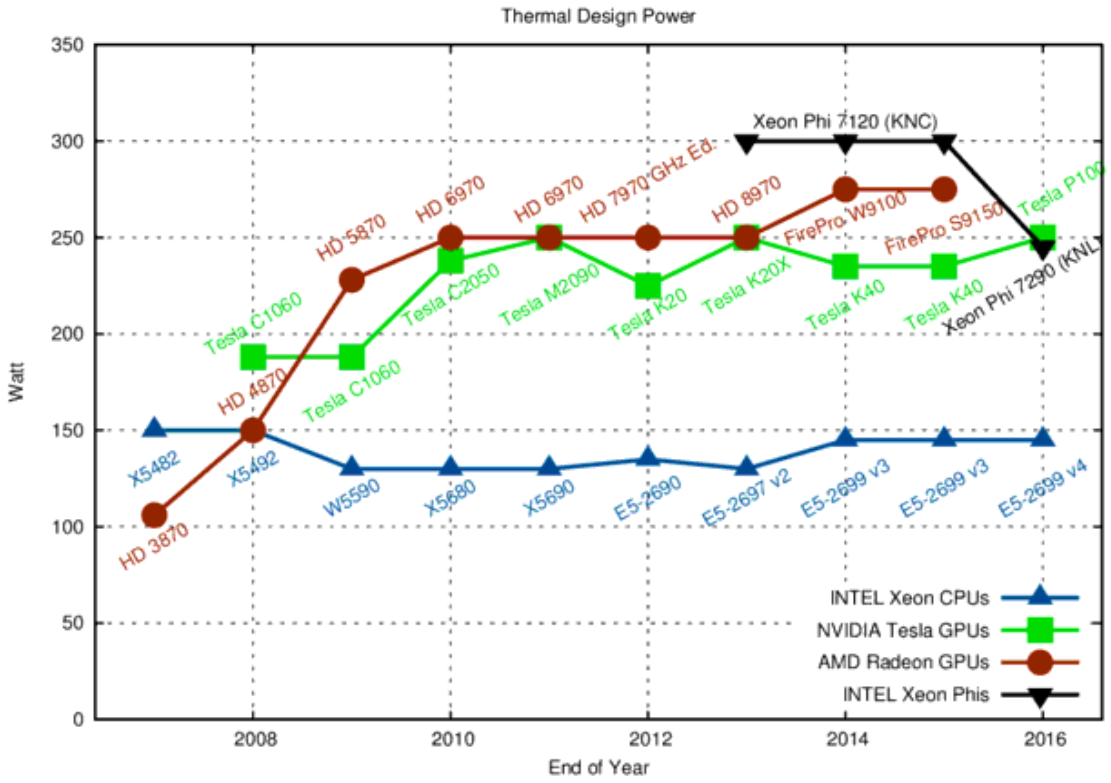


Figura 1.2: Comparativa de la evolución del consumo energético entre diversas plataformas a lo largo del tiempo. [27]

ción, y en este proyecto nos centraremos en su soporte tanto para el cálculo dinámico de puentes de ferrocarril (FFCC) como en el uso en HELENNA.

Las herramientas de cálculo dinámico de puentes utilizadas por empresas de consulting de ingeniería en nuestro país no se encuentran al nivel de cómputo del nivel tecnológico actual. Pese a esto, debido a la creciente demanda en número de pasajeros y transporte de mercancías la importancia que tiene el ferrocarril en nuestro país va en aumento año tras año. Además de la creciente demanda, debido a la antigüedad de la infraestructura ferroviaria en Europa, el 35 % de los 300,000 puentes ferroviarios ha alcanzado ya 100 años desde su construcción. El desgaste provocado por el paso del tiempo en estas estructuras es preocupante desde el punto de vista de la seguridad [24].

Debido a estos motivos, es necesario tener un control de las rutas que componen la red de ferrocarril y los puentes que se encuentran en ellas. Estos estudios son llevados a cabo utilizando cálculos de simulación dinámica, los cuales en la mayoría de los casos se realizan utilizando el Método de los Elementos finitos. Dicho método discretiza la estructura mediante interpolación polinómica de las ecuaciones de campo (vibraciones de medios continuos) y consigue soluciones numéricas de gran precisión.

Adicionalmente al uso de arquitecturas heterogéneas, en los sistemas de cómputo de altas prestaciones se utilizan miles de nodos interconectados entre si por una red de altas prestaciones. Muchas aplicaciones ofrecen posibilidad de distribuir el cálculo entre los diferentes nodos. Para ello, principalmente se utiliza MPI como especificación de paso de mensajes. **En este proyecto abordamos la comunicación entre procesos, principalmente en el modelado de puentes para un soporte a GPUs remotas**.

1.3 Objetivos

El objetivo principal de este trabajo es el soporte a la programación de arquitecturas hetergoéneas y al cálculo intensivo, siendo la GPU la tecnología que se ha elegido como arquitectura paralela para llevar a cabo la puesta en práctica de las implementaciones. En las dos aplicaciones que mas adelante se detallarán se ha llevado a cabo una implementación que permite la utilización de la plataforma GPU para la aceleración del cálculo y la optimización del tiempo de ejecución.

El objetivo principal se puede dividir en subobjetivos:

- Desarrollo de una interfaz OpenCL para HELENNA satisfaciendo todas las necesidades de cálculo para GPUs apoyándose en la utilización de la librería matemática para OpenCL: CLBLAS.
- Desarrollo de soporte OpenCL para diversas capas a utilizar en topologías de redes neuronales. Tales como capas convolucionales, capas *dropout* y capas *batch normalization*.
- Implementación de comunicación MPI entre diversos computadores interconectados a través de la red para llevar a cabo un sistema de cómputo con memoria distribuida.
- Optimización de una aplicación de simulación numérica de puentes de ferrocarril ante acciones dinámicas (trenes rápidos) mediante el uso de arquitecturas heterogéneas con OpenCL.

1.4 Estructura de la memoria

Esta memoria está compuesta de tres capítulos principales y un capítulo de conclusiones. El primer capítulo se centra en una explicación de las tecnologías utilizadas durante el desarrollo del proyecto con tal de que los siguientes capítulos puedan ser comprendidos con mayor facilidad. El segundo capítulo detalla el desarrollo del proyecto de simulación ferroviaria donde se ha implantado una implementación de la librería de arquitecturas heterogéneas OpenCL con tal de poder aprovechar la capacidad de la tecnología GPU. En el tercer capítulo se explica el trabajo llevado a cabo en el proyecto HELENNA perteneciente al Departamento de Informática de Sistemas y Computadores (DISCA) de la UPV. En el capítulo se detallan las tareas realizadas de desarrollo de un módulo que permite la utilización de GPUs mediante las librerías OpenCL y CLBLAS.

El presente trabajo se ha realizado por completo por el autor. Ahora bien, se ha colaborado en diferentes aspectos con Laura Medina en equipo. De hecho, cada uno de los dos proyectos ha tenido como objetivo el desarrollo del mismo soporte pero para plataformas distintas y complementarias. Este trabajo en equipo ha redundado en un mejor aprovechamiento y desarrollo de ambas plataformas, así como haber adquirido competencias de trabajo en equipo en el campo de la computación de altas prestaciones y en el uso de GPUs para su uso eficiente.

CAPÍTULO 2

OpenCL, BLAS, CLBLAS, MKL y MPI

En este capítulo describimos brevemente los diferentes modelos de programación y librerías de cálculo utilizadas en el trabajo. En primer lugar describimos OpenCL, un modelo de programación para arquitecturas heterogéneas. Seguidamente describimos BLAS, una librería de álgebra lineal, así como clBLAS, la instanciación de BLAS con OpenCL. Seguidamente describimos MKL, que supone una librería de álgebra lineal también pero optimizada para su uso en CPUs. Por último describimos MPI, un modelo de programación por paso de mensajes, así como una herramienta de *profiling* para MPI. Todos estos modelos y librerías se utilizan en el proyecto en menor o mayor medida.

2.1 BLAS

Las rutinas BLAS (Basic Linear Algebra Subprograms) [17] fueron propuestas en 1973 como un estándar de funciones Fortran dedicadas al cálculo de operaciones de álgebra lineal. Esta necesidad surgió del hecho de que antes de BLAS, cualquier aplicación debía codificar manualmente estas funciones. Esto presenta tres inconvenientes: Aumento del tiempo de desarrollo de software dedicado a esta tarea; Posibilidad de cometer errores en la implementación; Menor eficiencia en el cómputo final.

Hoy en día, pese a que el estándar BLAS comparte nombre con la librería de Fortran que implementa las funciones, ha surgido una gran cantidad de implementaciones para diversos lenguajes de programación y arquitecturas. Entre otras Intel MKL, CUBLAS, CLBLAS.

En secciones posteriores de este capítulo se describirá el funcionamiento de las librerias CLBLAS e Intel MKL ya que han sido utilizadas en las distintas aplicaciones llevadas a cabo en este proyecto.

La primera propuesta abarcó lo que hoy conocemos como el nivel 1 de rutinas BLAS. Este nivel está caracterizado por incluir únicamente operaciones vector-vector. Mas adelante, en 1988, con la necesidad de realizar operaciones matriz-vector y explotar la capacidad de los procesadores vectoriales se expandió al nivel 2 [16]. Esto supone un aumento significativo de la eficiencia ya que una operación matriz-vector (Nivel 2) es equivalente a realizar n operaciones vector-vector (Nivel 1). De hecho, se aplican optimizaciones que no serían posibles con operaciones de nivel 1 al estar restringido a operaciones vector-vector. Tal es el caso de aprovechar las propiedades matemáticas de la matriz que vamos a utilizar en la operación para llevarla a cabo de manera mas eficiente, así como poder distribuir las operaciones que deben llevarse a cabo de manera distinta para aprovechar

la localidad de la memoria u obtener una mejora en la eficiencia del cálculo. En la sección dedicada a OpenCL se muestra un ejemplo que ilustra esta ventaja. Finalmente en 1990 se presentó el nivel 3 [15] con soporte para operaciones matriz-matriz.

Cada una de las funciones existentes en BLAS presenta 4 modos de operación:

S	Simple precisión
D	Doble precisión
C	Simple precisión con aritmética de números complejos
Z	Doble precisión con aritmética de números complejos

A continuación se presenta una tabla que muestra todas las funciones que existen hoy en día en el estándar BLAS. Cada una de las funciones tiene precedida una x en su nombre, esto hace referencia a los cuatro modos de operación mencionados anteriormente.

BLAS Nivel 1	
Nombre	Función
xROTG	Generar rotación del plano
xROTMG	Generar rotación modificada del plano
xROT	Aplicar rotación del plano
xROTM	Aplicar rotación modificada del plano
xSWAP	$x \Leftrightarrow y$
xSCAL	$x \leftarrow \alpha x$
xCOPY	$y \leftarrow x$
xAXPY	$y \leftarrow \alpha x + y$
xDOT	$dot \leftarrow x^T y$
xDOTU	$dot \leftarrow x^T y$
xDOTC	$dot \leftarrow x^H y$
xxDOT	$dot \leftarrow \alpha + x^T y$
xNRM2	$nrm2 \leftarrow \|\|x\|\|_2$
xASUM	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$
IxAMAX	$x \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$

BLAS Nivel 2	
Nombre	Función
xGEMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$
xGBMV	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$
xHEMV	$y \leftarrow \alpha Ax + \beta y$
xHBMV	$y \leftarrow \alpha Ax + \beta y$
xHPMV	$y \leftarrow \alpha Ax + \beta y$
xSYMV	$y \leftarrow \alpha Ax + \beta y$
xSBMV	$y \leftarrow \alpha Ax + \beta y$
xSPMV	$y \leftarrow \alpha Ax + \beta y$
xTRMV	$y \leftarrow \alpha Ax, x \leftarrow A^T x, x \leftarrow A^H x$
xTBMV	$y \leftarrow \alpha Ax, x \leftarrow A^T x, x \leftarrow A^H x$
xTPMV	$y \leftarrow \alpha Ax, x \leftarrow A^T x, x \leftarrow A^H x$
xTRSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$
xTBSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$
xTPSV	$x \leftarrow A^{-1} x, x \leftarrow A^{-T} x, x \leftarrow A^{-H} x$
xGER	$A \leftarrow \alpha x y^T + A, A - m \times n$
xGERU	$A \leftarrow \alpha x y^T + A, A - m \times n$
xGERC	$A \leftarrow \alpha x y^T + A, A - m \times n$
xHER	$A \leftarrow \alpha x x^H + A$
xHPR	$A \leftarrow \alpha x x^H + A$
xHER2	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$
xHPR2	$A \leftarrow \alpha x y^H + y(\alpha x)^H + A$
xSYR	$A \leftarrow \alpha x x^T + A$
xSPR	$A \leftarrow \alpha x x^T + A$
xSYR2	$A \leftarrow \alpha x y^T + \alpha y x^T + A$
xSPR2	$A \leftarrow \alpha x y^T + \alpha y x^T + A$

BLAS Nivel 3	
Nombre	Función
xGEMM	$C \leftarrow \alpha op(A) op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$
xSYMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$
xHEMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$
xSYRK	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$
xHERK	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$
xSYR2K	$C \leftarrow AB^T + \overline{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \overline{\alpha} B^T A + \beta C, C - n \times n$
xHER2K	$C \leftarrow \alpha AB^H + \overline{\alpha} + BA^H + \beta C, C \leftarrow \alpha A^H B + \overline{\alpha} B^H A + \beta C, C - n \times n$
xTRMM	$B \leftarrow \alpha op(A) B, B \leftarrow \alpha B op(A), op(A) = A, A^T, A^H, B - m \times n$
xTRSM	$B \leftarrow \alpha op(A^-1) B, B \leftarrow \alpha B op(A^-1), op(A) = A, A^T, A^H, B - m \times n$

2.2 OpenCL

OpenCL es un modelo de programación para dispositivos de cálculo heterogéneos (conocidos como hardware heterogéneo), el cual es capaz de distribuir el cómputo en arquitecturas tales como CPU, GPU, FPGA[14] y DSP[18].

El objetivo principal de OpenCL consiste en la mejora de la eficiencia en la ejecución de determinadas funciones. Estas funciones deben ser suficientemente paralelizables para poder obtener los beneficios del uso de arquitecturas heterogeneas con una gran capacidad de paralelismo, por consiguiente, proporcionando una aceleración que no podría conseguirse al ejecutar estas funciones en una arquitectura basada exclusivamente en CPUs.

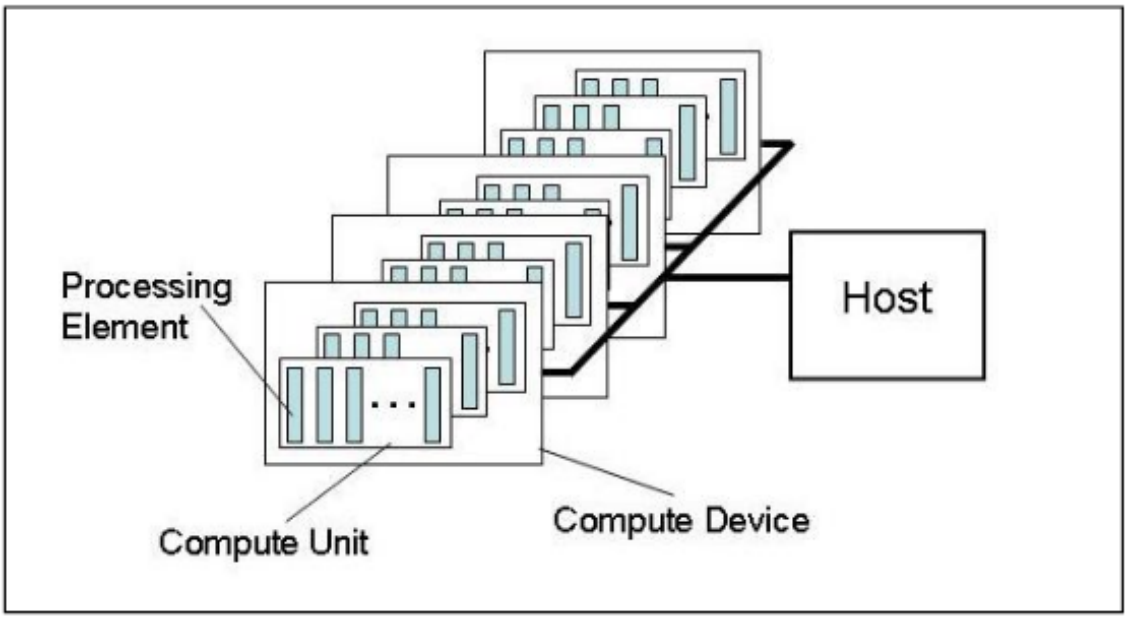


Figura 2.1: Esquema de funcionamineto del modelo OpenCL [29]

2.2.1. Modelo de ejecución

El modo de funcionamiento del modelo de programación OpenCL (Figura 2.1) consiste en un esquema donde como punto central encontramos el código de gestión, denominado *host*, el cual se ejecuta en una CPU. El host reparte el cómputo a diferentes dispositivos (*devices*) mediante el uso de colas de mensajes (*Command queues*: CQs), las cuales se configuran en tiempo de ejecución. Cada *device* tiene asociado una *CQ*.

Cada uno de estos *devices* contiene elementos de cómputo (*compute units*; *CU*) destinados al procesamiento de instrucciones. Una *CU* esta compuesta por un determinado número de elementos de procesamiento (*processing elements*; *PEs*). Conceptualmente podemos asociar estos elementos a la idea de hilos. En el momento en el que un *device* recibe una función a través de su correspondiente *CQ*, la misma función se ejecuta en cada uno de los *PEs*.

Las funciones que se ejecutan en OpenCL se denominan *kernels*. Un *kernel* se codifica en un lenguaje de programación especial para esta tarea. Su sintaxis es muy similar a C. También cuenta con ciertas subrutinas matemáticas que facilitan la codificación de las funciones. Una vez codificado un *kernel*, se enlaza en tiempo de ejecución al contexto OpenCL que contiene diversas *CQs*. En el momento de la ejecución del programa en el que se necesita hacer uso de un acelerador, el *kernel* que sustituye a la función en cuestión es configurado para ser encolado en la *CQ*.

OpenCL proporciona una abstracción para la ejecución de *kernels*. De hecho, la estructura de *devices* y *CUs* son transparentes al desarrollador. Esta abstracción se basa en dos elementos: *work-items* (*WI*) y *work-groups* (*WG*). La figura 2.2 muestra los dos elementos y como se distribuyen en un *CU*.

A continuación se muestra un ejemplo de código de un kernel utilizado en el proyecto. En concreto el kernel que calcula la función de activación *relu*. Es un ejemplo sencillo que no divide los WIs en WGs. Además únicamente define una dimensión en la *grid* global compuesta de WIs del problema.

$$relu(x) = max(0, x)$$

```
//Funcion que prepara la ejecucion del kernel
void fn_matrix_relu_clblas(cl_mem mid_src, cl_mem mid_dst, int rows, int cols) {
    //Preparacion del primer y segundo argumento de kernel para ser enviado a GPU
    set_kernel_arg(&relu, 0, sizeof(cl_mem), (void *)&mid_src, "relu");
    set_kernel_arg(&relu, 1, sizeof(cl_mem), (void *)&mid_dst, "relu");

    //Especificacion del numero de work_items que van a ser utilizados en GPU
        definiendo una unica dimension
    size_t globalThreads[1] = {rows * cols};
    execute_kernel(&relu, 1, globalThreads, NULL, "relu"); }
//Kernel de la funcion relu
__kernel void relu(__global float *src, __global float *dst) {
    //Obtencion del identificador del work_item
    int element = get_global_id(0);
    //Calculo del elemento del array correspondiente al work_item
    dst[element] = (src[element] < 0) ? 0.0 : src[element]; }
```

La función toma dos parámetros, dos punteros a zonas de memoria que contienen los datos de entrada en el caso del primer argumento y el resultado de la función en el caso del segundo. Ambas variables son globales, por lo tanto son comunes a todos los *WIs*. Dado que el problema no necesitaba la utilizacion de memoria local, ya que todas las operaciones sobre elementos del array son idependientes, únicamente ha sido especificado el tamaño global del problema. Por lo tanto no contamos con la utilidad de la memoria local y de la agrupación en work-groups.

Una aclaración sobre la obtención del identificador de WI, *get_global_id(0)*. Como es lógico existen dos funciones que acceden a identificadores de WI: *get_local_id()* y *get_global_id()*. Como ya se ha mencionado no utilizamos memoria local, por lo tanto únicamente hacemos uso de la función global. Por otro lado, ambas funciones toman un parámetro. Este parámetro hace referencia a la dimensión de la que estamos obteniendo el identificador en la malla de WIs. En el caso de esta función hemos definido una única dimensión, por lo tanto el parámetro toma el valor 0.

Para cada ejecución de un *kernel* se definen dos parámetros que influyen en la ejecución del *kernel*: *global_size* y *local_size*. Estos dos parámetros definen el número de *WIs* en una *CU* y el número de *WIs* en cada *WG*, respectivamente. El valor de *global_size* debe ser múltiplo de *local_size* mientras que el tamaño máximo de *local_size* viene determinado por la arquitectura del *device*.

Encontramos dos ventajas en la utilización de *WGs*: sincronización y velocidad de acceso a memoria. En referencia al aspecto de sincronización entre *WI*, OpenCL permite el uso de dos tipos de barreras: locales y globales. Las barreras locales afectan a los elementos de un *WG* y las barreras globales a todos los *WIs* de todos los *WGs*. En relación a la memoria, encontramos tres niveles. Memoria privada, local y global. La memoria privada es accedida por cada *work-item*. La memoria local es compartida por los elementos de un *work-group*. Finalmente, la memoria global puede ser accedida por todos los *work-items*. Como es lógico, el tiempo de acceso a cada una de estas memorias es distinto siendo la memoria privada la mas rápida en acceso y la memoria global la mas lenta. La figura 2.3 muestra la relación entre los diversos tipos de memoria y los *WIs* y *WGs*.

Para finalizar esta sección se muestra un ejemplo de kernel OpenCL donde se acelera la rutina GEMM, la multiplicación de matrices. Se ha decidido mostrar este ejemplo debido al uso que se realiza de todos los tipos de memoria que ofrece OpenCL previamente comentados, además de las rutinas de sincronización. Por lo tanto es un ejemplo muy completo que muestra la capacidad de OpenCL. Podemos encontrar el código de este ejemplo en el apéndice A.0.2. Además para ilustrar nuestra explicación haremos uso de una implementación simple cuyo código también se encuentra en el apéndice del do-

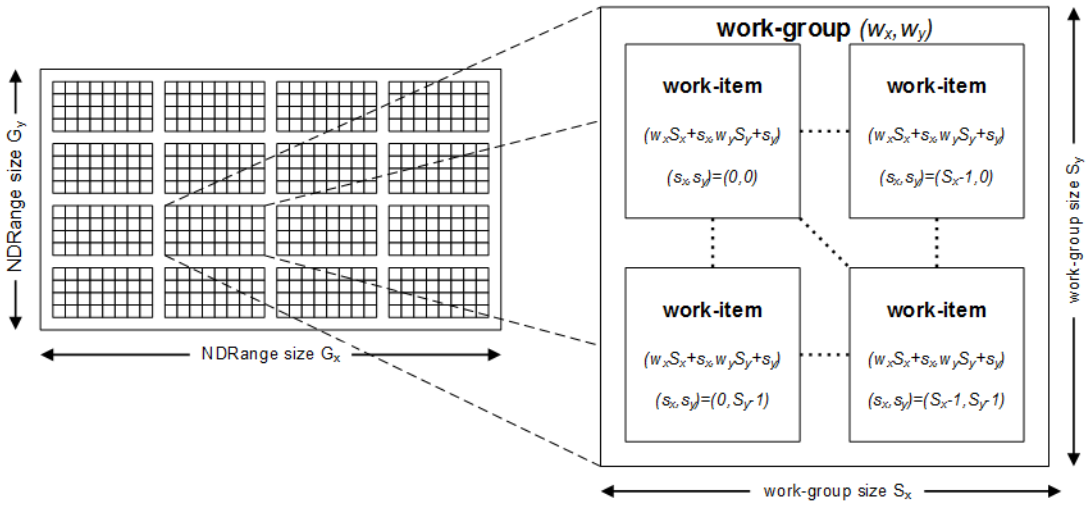


Figura 2.2: Esquema conceptual de la distribución de cómputo mediante *WIs* y *WGs* [12]

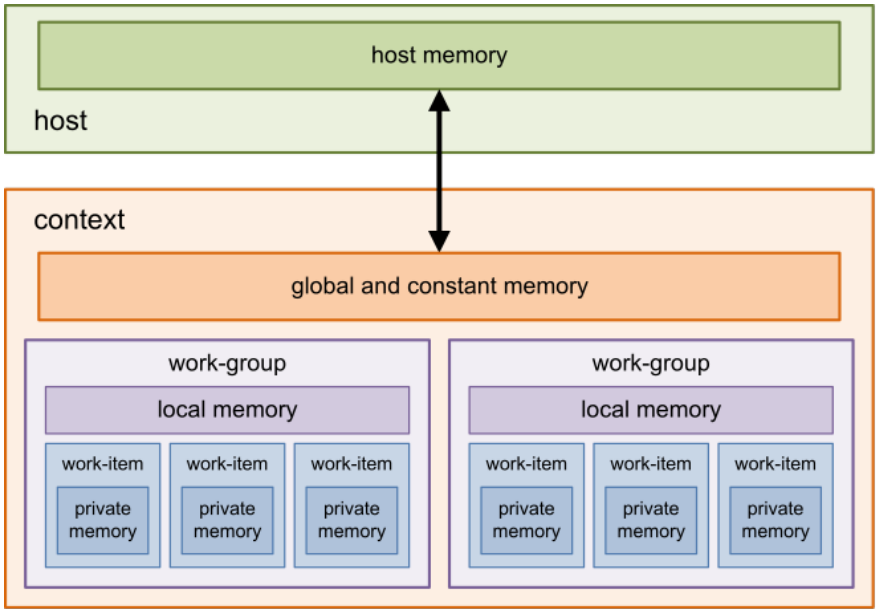


Figura 2.3: Tipos de memoria en un dispositivo OpenCL.

cumento A.0.1. El código del ejemplo pertenece a Cedric Nugteren, un desarrollador e investigador especializado en arquitecturas paralelas y heterogéneas. [25]

En primer lugar, este ejemplo muestra la ventaja que se obtiene al utilizar una función de nivel 3 BLAS frente a realizar la operación equivalente mediante el uso de rutinas de nivel 2 (Mediante operaciones vector-matriz) o incluso de nivel 1 (Mediante operaciones vector-vector). Las cuales obtendrían el mismo resultado pero al no conocer el contexto de la operación no se podrían llevar a cabo las mejoras que se han propuesto en este caso.

En comparación a la implementación simple A.0.1, donde se hace uso exclusivo de accesos a memoria global con un total de **2*M*N*K** *loads*. Nuestro ejemplo consigue reducir este problema utilizando memoria local. Dividiendo el trabajo en grupos llamados *tiles*. El siguiente esquema muestra el razonamiento matemático de esta división de las operaciones (Figura 2.5). Como resultado se consigue reducir el número de accesos a memoria: **2*M*N*#tiles** *loads* a memoria global; **2*M*N*K** *loads* a memoria local y **2*M*N*#tiles** *stores* en memoria local. El resto de coste temporal de operaciones se mantiene igual. La

ventaja de utilizar memoria local es que presenta un tiempo de acceso considerablemente menor al de memoria global.

De esta forma obtenemos los resultados mostrados en la figura 2.4.

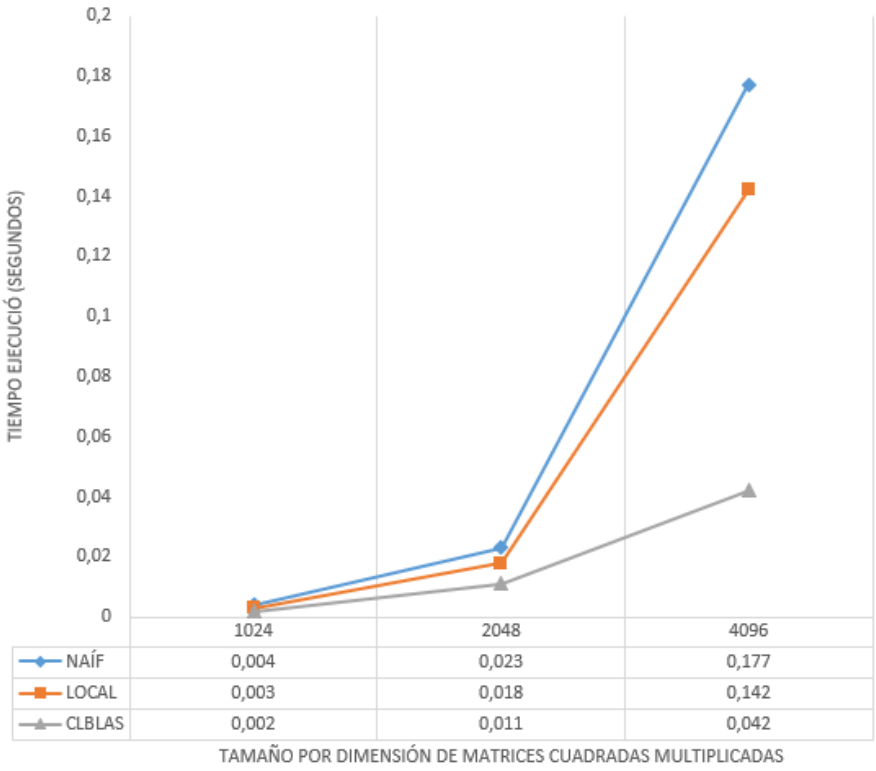


Figura 2.4: Tiempos multiplicación por implementaciones. Ejecutados en la misma CPU y GPU respectivamente. Simple precisión.

Hemos decidido comparar ambas implementaciones con una tercera, CLBLAS. Esta implementación cuenta con un mayor número de optimizaciones y ha sido realizada por expertos en la materia. A continuación se presenta con mayor detalle en que consiste esta librería.

2.3 CLBLAS

CLBLAS es la implementación para OpenCL de las rutinas BLAS. Forma parte de un proyecto Open Source llamado clMathLibraries dedicado a la implementación de rutinas matemáticas para OpenCL. [2].

El proyecto clMathLibraries incluye clRNG, dedicado a la generación de números aleatorios; clSPARSE, dedicado a la implementación de funciones de algebra lineal para matrices dispersas y clFFT, dedicado a la implementación de funciones que sirven para el cálculo de la transformada rápida de Fourier (FFT).

El objetivo de CLBLAS reside en permitir la máxima flexibilidad posible al programador. Por tanto la creación de un contexto de OpenCL, la gestión de una cola de comandos (*CQ*) e incluso de *kernels* propios no interfiere con el funcionamiento de CLBLAS. De hecho, su utilización es muy similar al de cualquier librería convencional para CPU.

En la figura 2.6 podemos observar una llamada a la función de multiplicación de matrices de nivel tres, GEMM. En concreto en su versión para números en simple precisión.

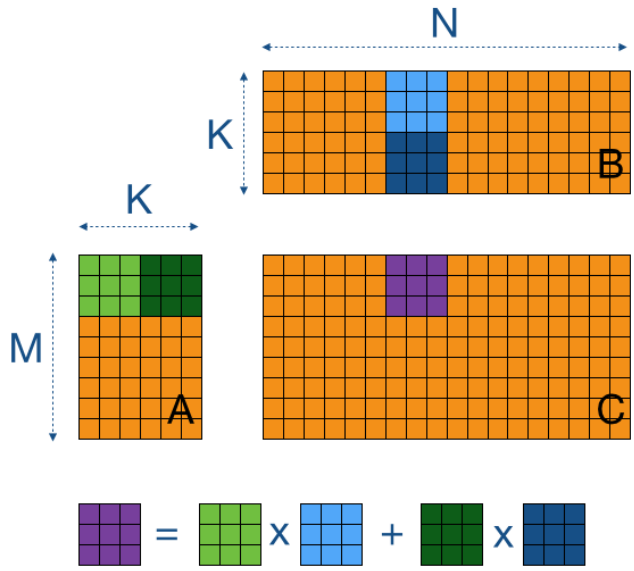


Figura 2.5: Esquema conceptual del funcionamiento del código de ejemplo. [25]

```
clblasOrder order = clblasRowMajor;
cl_float alpha = 1;
cl_float beta = 0;
clblasTranspose transA = clblasNoTrans;
clblasTranspose transB = clblasTrans;

int ret = clblasSgemm(order, transA, transB, rows_a, rows_b, cols_a, alpha,
    mid_a,
    0, cols_a, mid_b, offset_c, cols_c, 1, &queue_clblas, 0, NULL,
        &event_clblas);

synchronization(ret, "clblasSgemm");
```

Figura 2.6: Ejemplo de llamada a función CLBLAS

A continuación, describimos cada uno de los parámetros y funciones utilizadas que pueden ser de interés:

1. Los primeros parámetros sirven para indicar las matrices de entrada, sus tamaños y si se desean o no interpretar como matrices transpuestas.

2. *event_clblas* es una variable global que sirve para la sincronización ya que la llamada a la librería se realiza de forma asíncrona. La sincronización se realiza en la siguiente función (*synchronization*).

3. *queue_clblas* es la cola que envía las tareas al *device*. En el caso del ejemplo está configurada con la tarjeta gráfica con la que se han realizado los experimentos.

4. *synchronization* es una función construida mediante el uso de diversas rutinas OpenCL que sirve para gesitionar los errores que puedan surgir en la llamada a Sgemm.

2.4 MPI

MPI (*message passing interface*) [28] es una interfaz de programación estandarizada que consiste en la comunicación entre nodos de un sistema de computación paralela mediante

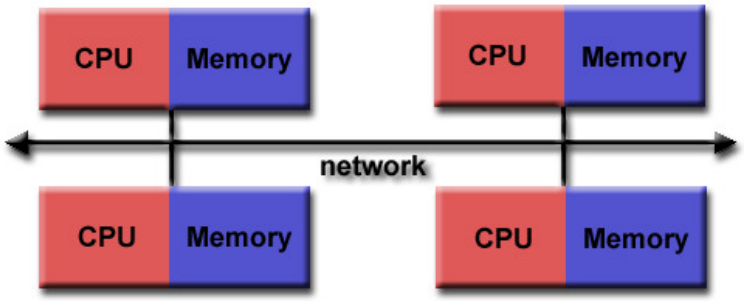


Figura 2.7: Esquema de memoria colectiva MPI. [13]

envío de mensajes. Permite, por tanto, la elaboración de algoritmos y aplicaciones con memoria distribuida a lo largo de una red de nodos intercomunicados.

Las funciones de envío de mensajes en MPI incluyen como principales parámetros un puntero *void* con la dirección de memoría de la cola (*buffer*) que se desea enviar y un entero (*integer*) indicando el número de elementos a enviar. El tipo de dato de los elementos que se encuentran en el *buffer* deben indicarse, existiendo los siguientes tipos de datos primitivos ofrecidos por MPI comparados con su equivalente en lenguaje C:

MPI	C
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char

Además, también existe la posibilidad de crear un tipo de dato derivado usando una combinación de los tipos de datos primitivos mencionados anteriormente.

```
MPI_Datatype newtype;
MPI_Type_<sometype>( < oldtype specifications >, &newtype );
MPI_Type_commit( &newtype );
```

En este ejemplo observamos el mecanismo por el cual se crea un nuevo tipo de dato MPI. Existen tres tipos de bases a partir de las cuales crear nuevos datos derivados: *MPI_Type_contiguous*, *MPI_Type_vector*, *MPI_Type_struct*. La explicación en profundidad de estos tipos y este mecanismo de creación se extiende mas allá del objetivo de este documento ya que no ha sido utilizado en el proyecto. Simplemente comentar que el mas simple es *MPI_Type_contiguous*, el cual simplemente consiste en un conjunto de *oldtype* consecutivos en memoria.

MPI ofrece funciones de comunicación punto a punto, funciones de comunicación colectivas, funciones de sincronización y funciones de gestión de procesos MPI. Para indicar los *hosts* que se desea que participen en el intercambio de mensaje se utilizan comunicadores. Los comunicadores son conjuntos de *hosts* identificados por un número. En la inicialización de MPI, en la llamada MPI_Init se crea un comunicador que engloba

todos los procesos MPI presentes, MPI_COMM_WORLD. Existen funciones que permiten la creación de comunicadores para casos de uso mas complejos. Las comunicaciones MPI se realizan entre los procesos del comunicador indicado, y en el caso de la comunicación punto a punto se indica el identificador del proceso destino dentro del comunicador. Seguidamente, se listan los diferentes tipos de funciones:

- Comunicación punto a punto:
  - MPI_Send: Envía un mensaje desde el proceso *root*, el cual realiza la llamada, hasta el *host* indicado. Esta llamada debe estar emparejada con una llamada a MPI_Recv en el proceso destinatario.
  - MPI_Recv: Espera un mensaje desde el *host* indicado.
- Comunicación colectiva:
  - MPI_Broadcast: Envía la misma información a todo el resto de procesos en el comunicador.
  - MPI_Gather: Recibe información de los procesos del comunicador (*root* incluido). Equivalente a que el *root* realize n llamadas Recv y que los n procesos realicen una llamada Send.
  - MPI_Scatter: Divide el *buffer* y envía a cada proceso del comunicador una parte.
  - MPI_Reduce: Combina los mensajes recibidos por todos los procesos.
- Sincronización:
  - MPI_Wait: Espera a la finalización de una llamada Send o Recv asíncrona.
- Gestión procesos MPI:
  - MPI_Comm_spawn: Crea un número determinado de procesos en los *hosts* (IP's) indicados. Crea dos comunicadores:
    - Un intercomunicador que comunica el comunicador que ha realizado la llamada y el grupo de procesos creados en la llamada a MPI_Comm_Spawn.

    Un intracomunicador que comunica el grupo de procesos creados.
  - MPI_Finalize: Es una llamada que debe ser invocada por todos los procesos MPI existentes y sirve para finalizar el entorno MPI.

Para ilustrar esta explicación, a continuación se muestra un ejemplo de uso de MPI para el cálculo del número de números primos entre un intervalo de enteros. Este ejemplo ha sido obtenido de la web de la universidad de Florida, en una sección donde se ofrecen diversos ejemplos de utilización de MPI. [6]

Se puede observar la utilidad de las rutinas colectivas MPI previamente explicadas.

```
int main ( int argc, char *argv[] ) {
   //Se inicializa el entorno MPI
   MPI_Init ( &argc, &argv );
   //Se obtiene informacion del comunicador creado
   MPI_Comm_size ( MPI_COMM_WORLD, &p );
   MPI_Comm_rank ( MPI_COMM_WORLD, &id );
   while ( n <= n_hi ) {
      //El proceso 0 envia un int al resto de procesos: n
      MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );

      //Cada proceso calcula el numero de primos desde 1 hasta n
```

```
        //Para no solapar tareas la funcion comienza la busqueda en 2+id
        //y va avanzando de p (#procesos) en p.
        primes_part = primes_between ( n, id, p );

        //Aqui se suma el numero de primos totales encontrados entre 1 y n
        //que han calculado todos los procesos
        MPI_Reduce ( &primes_part, &primes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
        if ( id == 0 ) {
          printf ("%8d %8d\n", n, primes);
        }
        n = n * n_factor;
    }
    MPI_Finalize();
    return 0;
}

int primes_between ( int n, int id, int p ) {
    int i, j, prime, total;
    total = 0;
    for ( i = 2 + id; i <= n; i = i + p ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( ( i % j ) == 0 ) {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}
```

2.5 MKL

Math Kernel Library (MKL) es una librería matemática desarrollada por Intel. Es una aplicación ampliamente utilizada en el ámbito de ciencia, ingeniería, financiero, etc. Por su facilidad de utilización y su eficiencia.

La implementación de las rutinas que ofrece MKL es específica para CPU. Esto supone ventajas e inconvenientes en comparación a una implementación en GPU. La principal ventaja es su facilidad de utilización y su robustez ya que las CPUs son un hardware muy extendido y en una fase de madurez superior al de GPUs. Por otro lado, pese a la gran eficiencia de esta librería, la capacidad de cómputo que ofrece una GPU en tareas paralalelizables puede competir y superar fácilmente los resultados ofrecidos por MKL. Por este motivo hemos decidido estudiar la utilización de arquitecturas paralelas y no contentarnos con los resultados de CPUs en nuestras aplicaciones.

Entre las funciones que implementa esta librería encontramos: Los tres niveles de rutinas BLAS; Funciones para matrices dispersas; Funciones que calculan la Transformada rápida de Fourier (FFT).

A continuación se muestra un ejemplo de uso de MKL en una de las aplicaciones del proyecto.

```
    for (int i = 0; i < rows_a; i++) {
        v[i] = cblas_sdot(cols_a, &a[i * cols_a], 1, &ones, 0);
    }
```

En el ejemplo se muestra una función que ha sido llevada a cabo mediante una rutina MKL. En concreto la rutina de álgebra lineal BLAS de nivel 1, SDOT. Que lleva a cabo el producto escalar de dos vectores.

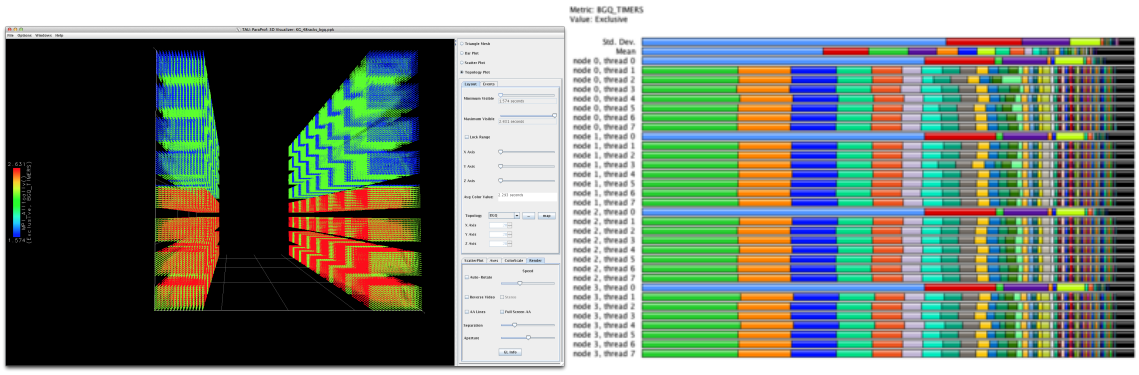
Los vectores que se multiplican en la rutina son: Vector columna de la matriz **a** y un vector de 1's. El sentido de esta función es sumar las filas de **a** en un vector columna **v**. La función del bucle es sumar todos los valores de cada columna de la matriz en un vector de tamaño número de columnas.

2.6 Profiling MPI

La herramienta que hemos escogido para estudiar las llamadas MPI es TAU. Una herramienta desarrollada por la Universidad de Oregon y Los Alamos National Laboratory en USA además de *Forschungszentrum Jülich* en Alemania. [10].

La utilización de TAU es sencilla, se compila junto a la aplicación utilizando un script llamado tau_cc.sh, el cual se encarga de enlazar y compilar todas las librerias y headers relacionados con la herramienta de forma transparente al programador.

Una vez compilado el programa, TAU genera un archivo de profiling por cada nodo del programa que se ha creado. Los archivos de profiling estan preparados para ser analizados mediante la herramienta *Paraprof*, (Figura 2.8). Esta herramienta proporciona una GUI para analizar los datos generados. En el caso de nuestro proyecto no ha sido necesaria la utilización de la visualización 3D debido a la sencillez de la aplicacion. Este es un ejemplo de un uso similar al que se ha llevado a cabo de Paraprof en nuestro proyecto (Figura 2.8b).



(a) Ejemplo de visualización de los procesos en 3D

(b) Ejemplo de la visualización en gráfico de barras [10]

Figura 2.8: Ejemplos de la herramienta Paraprof de TAU [11]

CAPÍTULO 3
Aplicación de modelado de puentes

En este capítulo presentamos los desarrollos realizados para el soporte de cálculo en GPUs para la aplicación de modelado de puentes. En un primer punto introducimos el contexto histórico de las estructuras ferroviarias en un ámbito nacional. A continuación explicamos el punto de partida de nuestro desarrollo a partir de una aplicación ya desarrollada de simulaciones físicas. En la siguiente sección realizamos un recorrido por el desarrollo de la aplicación hasta el punto actual del mismo. Finalmente detallamos el uso de las herramientas que hemos empleado para llevar a cabo el desarrollo y el análisis de sus prestaciones. Para concluir realizamos una evaluación de los resultados obtenidos y discutimos posibles mejoras que podrían llevarse a cabo en un futuro.

3.1 Contexto de la aplicación

3.1.1. Contexto actual y futuro de las estructuras ferroviarias nacionales

Debido a que la demanda de número de pasajeros y transporte de mercancías aumenta constantemente en nuestro país y también en todo el mundo, también es creciente la importancia del ferrocarril (FFCC). Sin embargo, a causa de que es un servicio de transporte considerablemente antiguo, más del 35 % de los 300,000 puentes ferroviarios en Europa han alcanzado ya los 100 años de antigüedad, y su fiabilidad a medio plazo es un punto crítico en la seguridad de la red ferroviaria [30].

Además de su antigüedad, existe un elevado número de puentes de red ferroviaria convencional en España que necesitan un trabajo de conservación. Conservar estas infraestructuras ferroviarias es una necesidad básica, siendo los puentes de las redes ferroviarias un punto clave que precisa inspecciones periódicas a fin de reducir riesgos de seguridad y costes de mantenimiento.

Adicionalmente a las necesidades derivadas de estudios dedicados al mantenimiento de puentes, existen otras problemáticas de interés. Una de ellas es la búsqueda de un servicio mas eficiente que cubra las crecientes necesidades de la sociedad, lo cual conlleva a un necesario aumento de la velocidad de paso de los trenes y de la carga que estos transportan.

Por tanto y en resumen, la modelización de puentes para cálculo estructural es necesario emplearla para realizar simulaciones de su respuesta dinámica futura ante el paso de trenes en los siguientes casos de interés: (A) análisis asociados a tareas de mantenimiento ordinario en puentes que por su edad lo requieran; (B) análisis de estructuras

existentes sometidas a tráfico más rápido y/o más pesado; (C) proyecto de estructuras nuevas en líneas de tráfico rápido de pasajeros. Los casos (A) y (B) pueden denominarse conjuntamente como revaluación".

3.1.2. Computación de altas prestaciones aplicada al campo del cálculo estructural de puentes ferroviarios

La actualización de los métodos de cálculo para realizar un seguimiento fiable de la evolución de los puentes a lo largo del tiempo es uno de los puntos fundamentales a la hora de alcanzar la fiabilidad y seguridad de las estructuras ferroviarias. De esta manera, los puentes que forman el conjunto de rutas ferroviarias son localizaciones críticas, debido a que un fallo en uno de ellos provocaría el estrangulamiento completo del tráfico que recorre su ruta. Por ejemplo, no ocurre lo mismo que sucedería en el caso del fallo en un puente de carretera, donde el tráfico puede ser fácilmente redirigido.

A causa del grado de incertidumbre que presenta el cálculo relativo al estudio del estado de los puentes, se imponen cada vez con mayor frecuencia los cálculos probabilistas, basados en la realización de un elevado número de simulaciones. Estos cálculos probabilistas requieren una creciente capacidad de cómputo para ser realizados. En este punto situamos la necesidad del uso de sistemas heterogéneos que sean capaces de combinar la capacidad de CPUs con la potencia de cómputo de arquitecturas paralelas como GPUs o FPGAs.

La aplicación de métodos probabilistas permiten el estudio en profundidad de la variabilidad intrínseca de los parámetros asociados al tren, la vía y el puente. M. Gilbert [19] manifiesta la necesidad de evolucionar los métodos de cálculo que estudian estas estructuras ya que en dicho estudio se menciona que el coste de sustituir todos los puentes más antiguos de Gran Bretaña sería del orden de 20.000 millones de libras. Este coste puede evitarse si se poseen las herramientas necesarias para estudiar los límites de estas estructuras y no se sustituyen de manera conservadora.

Las dimensiones de los sistemas de ecuaciones utilizados en los cálculos dinámicos de puentes ponen de manifiesto la necesidad de utilización de métodos computacionales de altas prestaciones. Las simulaciones lineales son más manejables desde un punto de vista computacional ya que generan dimensiones menores del problema. Pero estas simulaciones no ofrecen la precisión con la que se cuenta en simulaciones no lineales. De esta manera, si no se desea realizar un cálculo excesivamente conservador, es preferible emplear métodos no lineales, los cuales generan problemas con dimensiones mayores. Estos cálculos requieren tiempos de ejecución inabordables si se emplean métodos de computación habituales. Por ello, de nuevo se observa la necesidad de los métodos computacionales de altas prestaciones.

3.2 Descripción de la Aplicación

Partimos de una aplicación de simulaciones de puentes codificada íntegramente en lenguaje Fortran y desarrollada en el contexto del desarrollo de una tesis doctoral en la Universidad Politécnica de Madrid [26]. La tesis ha sido desarrollada por el profesor de la Escuela Técnica Superior de Ingeniería de Caminos, Canales y Puertos (UPV) Pedro Museros Romero, el cual figura como uno de los tutores de este trabajo y ha sido de gran ayuda durante todo el desarrollo de nuestra aplicación y de este proyecto.

Esta aplicación original tiene como entrada de datos un conjunto de ficheros donde el contenido se basa en los parámetros necesarios para la descripción tanto del puente

ferroviario como de los trenes. Así pues, el programa realiza un análisis no determinista (cálculos probabilistas) sobre cada puente simulando el paso del tren previamente descrito. Este análisis recoge los resultados obtenidos de las diferentes simulaciones lanzadas, donde todas las simulaciones se ejecutan secuencialmente con una velocidad de paso del tren diferente. Cada simulación realiza un conjunto de cálculos complejo, lo cual hace que la aplicación tenga un índice de GFLOPS de potencia muy bajo.

El análisis no determinista consiste en realizar variaciones aleatorias a las características del puente para cada simulación del paso de un tren sobre este. Las variables aleatorias son creadas siguiendo el método Monte Carlo. La generación de dichas variables aleatorias ha sido llevado a cabo mediante un *script* de Matlab, desarrollado conjuntamente entre el profesor Museros y el profesor Roberto Palma Guerrero de la Universidad de Granada [22].

Dado que la finalidad de cada simulación es pronosticar la respuesta ante el paso de un tren a una velocidad determinada por un puente dado, cada una de estas simulaciones solo depende de los parámetros necesarios para ella misma, sin dependencia alguna de las otras. Esto significa que cada una de las simulaciones se pueden realizar de manera separada e independiente, por lo que nos encontramos ante un contexto fácilmente paralelizable.

La aplicación original de partida en este proyecto se ejecuta solamente sobre un computador y en su CPU exclusivamente. Utiliza para el cálculo de matrices las librerias de Fortran. Por tanto, no tiene ningún soporte inicial para GPUs ni para realizar un procesamiento distribuido sobre más de un computador al mismo tiempo.

Cabe comentar que, debido a que el código original ha sido programado en Fortran, nuestra aplicación ha requerido de una implementación mixta entre Fortran y C (lenguaje en el que están programadas tanto las librerías de MPI como de OpenCL y CLBLAS). Para llevar a cabo esta programación mixta fue necesario un trabajo no trivial de consulta bibliográfica [23] y diversas pruebas. Finalmente se obtuvo un módulo Fortran que permitía la comunicación entre funciones de ambos lenguajes. El código de este módulo se encuentra en el anexo (Figura A.0.4).

3.3 Estructura de la aplicación

Para la mejora y adaptación de la aplicación se ha planteado un modelo estructural de comunicación entre diversos procesos mediante MPI: La aplicación esta formada por tres tipos de procesos: *master*, *kernel* y *server*. A continuación se describe la función de cada uno de ellos (Figura 3.1).

La aplicación de simulaciones en Fortran resuelve una serie de ecuaciones de álgebra lineal a partir de unos parámetros. Es decir, resuelve las mismas ecuaciones para distintos valores, los cuales están asociados a diversos tipos de trenes, velocidades y variaciones en las características del puente (Figura 3.2). Nuestra aplicación por tanto aprovecha que cada uno de estos casos de estudio son independientes del resto para poder paralelizar la resolución de las ecuaciones. En la gestión de la resolución de las ecuaciones y las simulaciones encontramos los procesos **master** y **kernel**.

Las máquinas donde los procesos deben ser ejecutados tienen solamente una restricción: Los procesos *server* necesitan una GPU y un entorno OpenCL correctamente configurado. Por lo tanto es posible una multitud de combinaciones de procesos y máquinas como por ejemplo: *Disponemos de una sola máquina con una GPU y ejecutamos un proceso de cada en ella*; *Disponemos de dos máquinas con una GPU en cada una y ejecutamos un master en la primera, un kernel y un server en cada una de ellas.*; *Disponemos de una máquina con GPU*

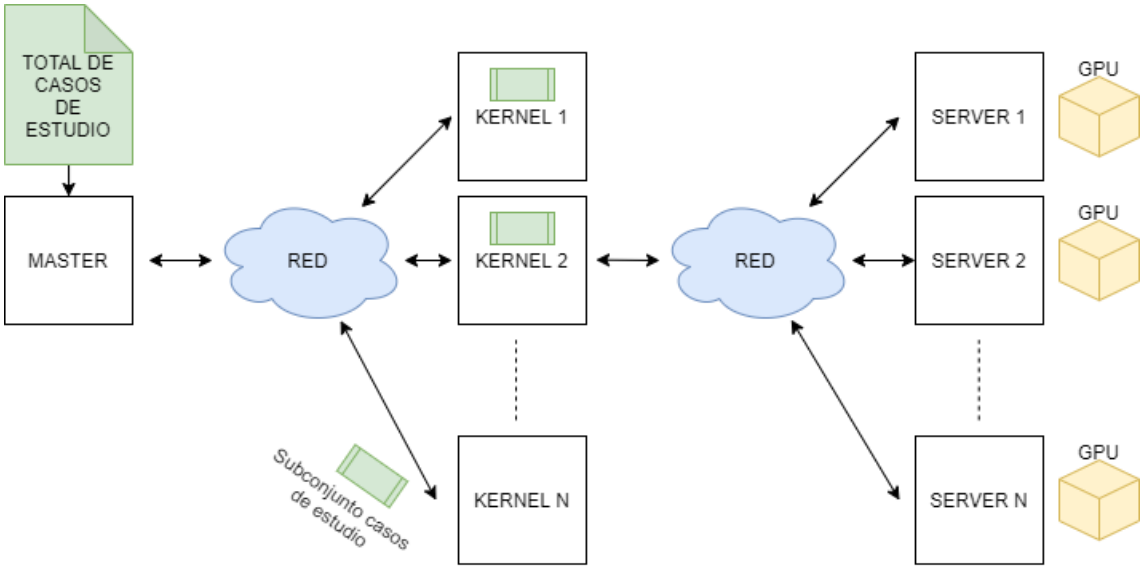


Figura 3.1: Esquema del modelo Master/Kernel/Server.

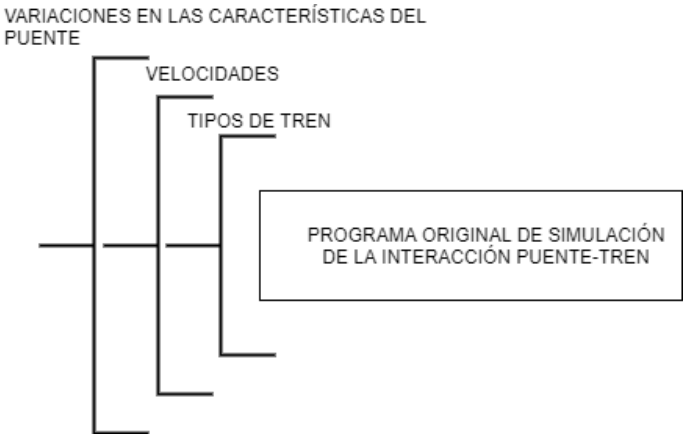


Figura 3.2: Esquema de las variables que se tienen en cuenta en las simulaciones.

y otra sin GPU y ejecutamos el proceso master y cuatro procesos kernel en la máquina sin GPU y un proceso server en la máquina con GPU. Todas estas combinaciones son decisión del usuario en el momento en el que ejecuta la aplicación. El usuario deberá evaluar cual es la utilización mas lógica de sus recursos.

3.3.1. Proceso master

Master es el único proceso que se crea al ejecutar la aplicación y es el encargado de cuatro tareas:

- Hacer *spawn* del resto de procesos tanto **kernels** como **servers**.
- Procesar todos los casos de estudio que se van a llevar a cabo y repartirlos de forma homogénea a los **kernels**.
- Recibir la información de los casos de estudio terminados y agruparla en un único fichero resultado para el posterior análisis del mismo.
- Comunicar a **kernels** y **servers** que la ejecución ha terminado y deben finalizar sus procesos.

El proceso **master** puede ser ejecutado en cualquier máquina sin necesidad de disponer una GPU.

3.3.2. Procesos kernel

Los procesos **kernel** son creados por **master** y en ese momento reciben el subconjunto del problema a resolver mediante mensajes MPI. A partir del momento en que terminan de recibir la información necesaria de un caso de estudio, ejecutan una instancia del programa original de Fortran y realizan la simulación con el subconjunto de datos.

La simulación llevará a cabo multitud de operaciones de álgebra lineal, entre ellas las multiplicaciones de matrices. El programa original de Fortran ejecutado por **kernel** lleva a cabo una ejecución habitual del mismo con la particularidad de que cada vez que se encuentre con que debe realizar una multiplicación de matrices comenzará un proceso de decisión. Decisión que valorará si la ventaja en potencia de cálculo que ofrece la GPU es suficiente para el *overhead* de envío y recepción de datos. Esta decisión es importante ya que a priori podría pensarse que la ejecución en GPU siempre es mas rápida pero como a continuación explicaremos esto no es siempre cierto. El tiempo necesario para llevar a cabo esta operación esta determinado por dos factores: El envío y la recepción de memoria del dispositivo y el tiempo de cálculo del dispositivo.

En cuanto al primer factor, el tiempo necesario para enviar y recibir la información de la operación. Inicialmente la información del programa reside en memoria principal, la memoria que puede acceder la CPU. Para que la GPU pueda llevar a cabo la operación las dos matrices que van a multiplicarse deben ser enviadas a la memoria de la GPU. El coste temporal del envío de memoria es proporcional con el tamaño de las matrices. Una vez procesada la operación el resultado debe ser enviado de vuelta a memoria principal de CPU para continuar con la ejecución de la simulación. Si la operación es llevada a cabo en CPU la información ya reside en memoria y por tanto no es necesario un envío entre distintas memorias.

En cuanto al segundo factor, el coste de la multiplicación de matrices, aquí es donde la GPU ofrece una ventaja. El coste temporal de la operación también es proporcional con el tamaño de las matrices a multiplicar. Esto significa que a mayor tamaño mayor es la diferencia en coste temporal entre CPU y GPU como veremos mas adelante.

Por tanto es necesario un balance entre el tiempo de envío de información y cálculo de la operación para obtener el mejor resultado. El estudio de esta decisión ha sido llevado a cabo de forma experimental. En la sección de evaluación de este capítulo se muestran los valores obtenidos del estudio del tiempo de multiplicación de matrices de doble precisión en Fortran y en una GPU a través de un proceso en una máquina remota.

El proceso de decisión involucra a los procesos **kernel** y **server**. Es el siguiente: Se han sustituido en el programa original todas las llamadas a la función *matmul* de Fortran. En su lugar se llama a una función Fortran que se encarga de ejecutar la multiplicación mediante la función *matmul* de Fortran o se decide enviar a la GPU perteneciente a algún proceso **server**. Esta función se puede encontrar en el anexo del documento (Fig A.0.4). A continuación encontramos un fragmento de la función completa.

```
function matmul_cuda_m_m(a,b) result(c)
    real*8                              :: a(:,:), b(:,:)
    real*8, dimension(size(a,1),size(b,2)) :: c

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

```
    end if
  end function matmul_cuda_m_m
```

3.3.3. Procesos server

Finalmente describimos el proceso **server**. Este es el receptor de las operaciones matriciales. Mediante un protocolo de mensajes enviados mediante MPI se comunica con cualquier kernel que desee realizar una petición de cálculo matricial.

El protocolo de comunicación entre **server** y **kernel** es el siguiente: En la tabla 3.1 se muestran los tipos de mensajes que son enviados a través de este protocolo. El valor de este mensaje es codificado mediante el parámetro *label* de los mensajes MPI. Lo cual permite ampliar la funcionalidad del protocolo de una manera muy sencilla.

En la figura 3.3 se describe de forma gráfica el protocolo de comunicación que se ocurre entre un proceso **kernel** y **server** cuando el primero desea realizar una petición de cálculo de multiplicación de matrices.

Valor	#def	Significado
0	ACK	Confirmación
1	PETICION_CALCULO	Kernel se comunica con server para pedir un cálculo
2	ENVIO_DIMENSIONES	Kernel envía M N K a server
3	ENVIO_MATRICES	Kernel se comunica con server para el envio de las matrices
4	ENVIO_MATRIZ_A	Kernel envía la matriz a server
5	ENVIO_MATRIZ_B	Kernel envía la matriz b a server
6	ENVIO_RESULTADOS	Server devuelve C a kernel
7	CIERRE	Master envia un mensaje de cierre al server

Tabla 3.1: Descripción del protocolo de comunicación creado entre *kernel* y *server*.

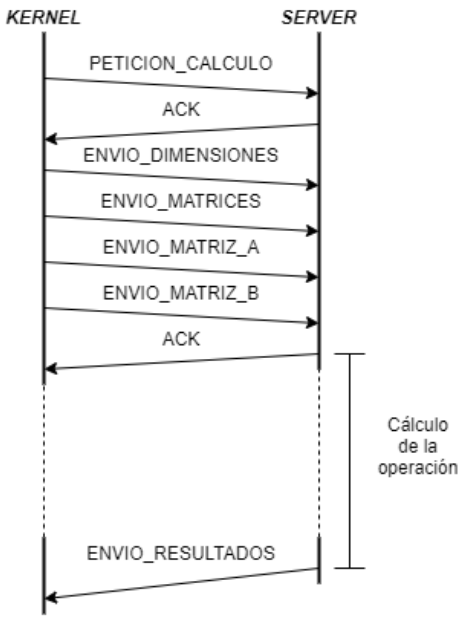


Figura 3.3: Protocolo de comunicación entre un proceso *kernel* y *server*. Donde el proceso *kernel* inicia la comunicación realizando una petición de multipliación de matrices remota.

3.4 Soporte de GPUs con OPENCL

Tal y como se ha mencionado anteriormente, la aplicación esta centrada en acelerar el cálculo matricial. Para ello hemos utilizado la librerias OpenCL y CLBLAS, las cuales nos permiten el uso de GPUs.

La función que lleva a cabo el cálculo matricial es GEMM (general matrix multiplication). Hemos decidido utilizar una implementación ya existente de la librería CLBLAS ya que ofrece un resultado considerablemente superior a cualquier implementación que podríamos haber creado nosotros mismos como se ha demostrado en el ejemplo de la sección de OpenCL.

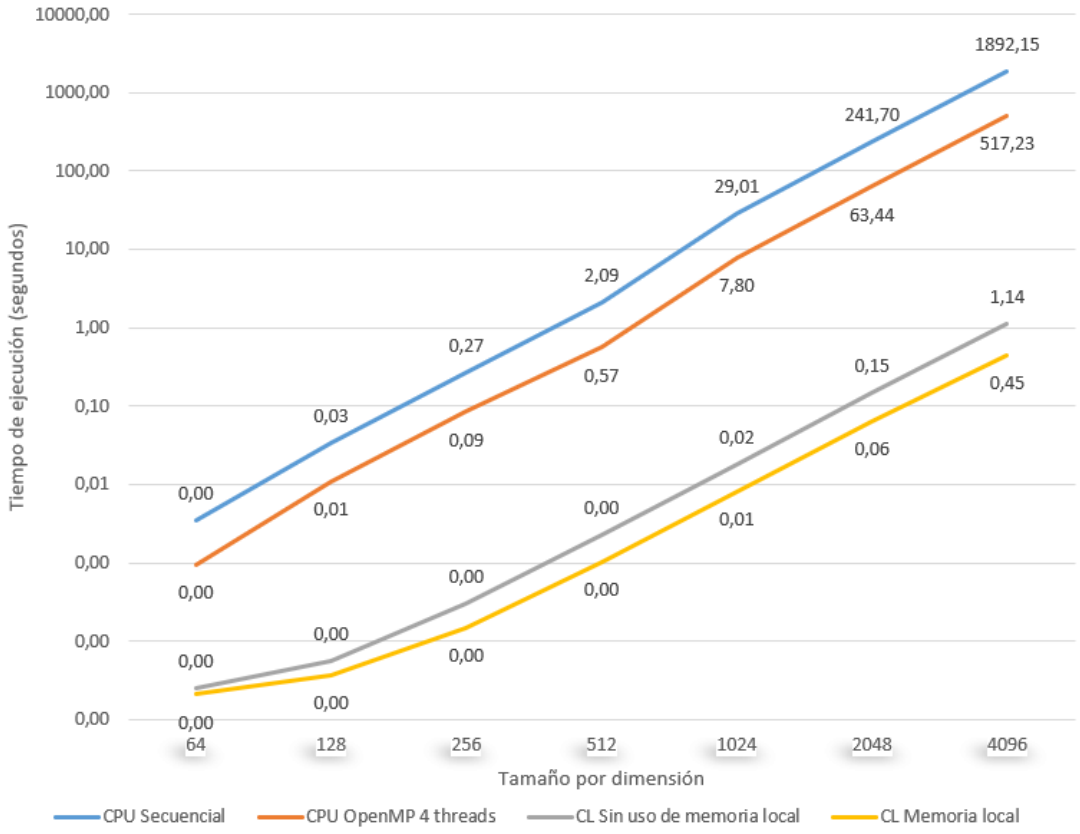


Figura 3.4: Comparativa de tiempos de ejecución entre diferentes implementaciones de multiplicación de matrices. Realizados en la misma GPU y CPU respectivamente. Simple Precisión.

Al comienzo del desarrollo de la aplicación se realizó una comparativa del coste temporal de cálculo de multiplicaciones de matrices cuadradas en diferentes dispositivos y algoritmos para conocer la magnitud de la mejora que se podía llegar a obtener.

A continuación se muestra una tabla y una serie de gráficos que presentan los resultados obtenidos en las mediciones. Se han tenido en cuenta un total de 5 posibles variaciones de hardware y algoritmo: En primer lugar el método que peores resultados ha ofrecido como es lógico ha sido el de CPU secuencial. Este será el caso base contra el cual compararemos los *speed-up's* obtenidos en las posteriores ejecuciones; En segundo lugar se ha ejecutado de nuevo en CPU pero esta vez utilizando OpenMP, una librería paralela especializada en CPU, para llevar a cabo la ejecución en 4 hilos de CPU; En tercer lugar se ha utilizado la GPU comenzando por una implementación simple A.0.1; En

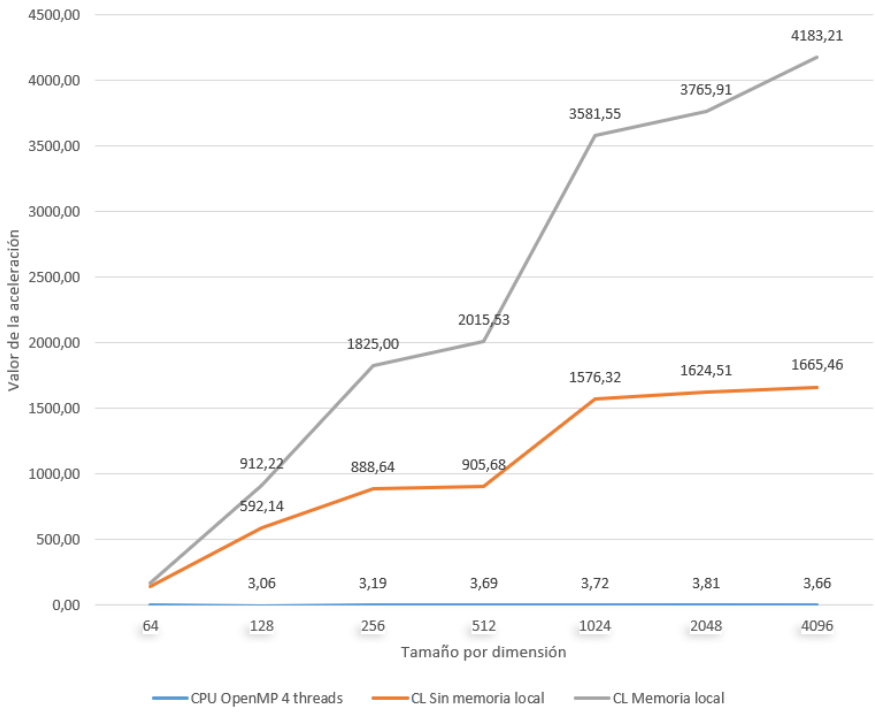


Figura 3.5: Comparativa de valores de la aceleración entre diferentes implementaciones de multiplicación de matrices en comparación al tiempo de ejecución en una CPU secuencial.

cuarto lugar continuando con la GPU se ha utilizado una implementación con una ligera optimización A.0.2, comentada en la sección de OpenCL de este mismo documento.

En el gráfico 3.4 encontramos las medidas de tiempo tomadas para el cálculo de la multiplicación de matrices cuadradas para diferentes tamaños de dimensión. El gráfico 3.5 es un cálculo de la aceleración de cada implementación comparada con el tiempo en CPU ejecutado de manera secuencial. Como aclaración comentar que todas las mediciones han sido llevadas a cabo en la misma CPU y en la misma GPU respectivamente.

3.5 Soporte de comunicación remota con MPI

En primer lugar, el motivo por el que se ha decidido utilizar la herramienta MPI ha sido principalmente que su utilización es relativamente sencilla y se adapta perfectamente a las necesidades de la aplicación. MPI funciona mediante el envío de mensajes entre nodos con una estructura de memoria distribuida. Se ha podido diseñar el esquema en el que un nodo de la red ejecuta un subconjunto del problema *kernel* y otro nodo de la red recibe mensajes con las matrices que tiene que multiplicar de los nodos que están ejecutando los diversos subconjuntos del problema total.

Mediante MPI hemos implementado el protocolo de comunicación mencionado anteriormente utilizando del parámetro *tag* de las funciones MPI_Send y MPI_Recv de la siguiente forma:

```
//SERVER

MPI_Recv(NULL, 0, MPI_INT, MPI_ANY_SOURCE, PETICION_CALCULO, MPI_COMM_WORLD, &status);

...

MPI_Send(NULL, 0, MPI_INT, rank, ACK, MPI_COMM_WORLD);
```

A continuación mostramos las medidas realizadas mediante la herramienta TAU que analiza las comunicaciones MPI de nuestra aplicación. Con el objetivo de conocer la utilización de los envíos de mensajes, principalmente entre **kernels** y **servers**. Las medidas obtenidas han sido tomadas con la siguiente configuración: Un proceso **master**, dos procesos **kernel** y un proceso **server**. Todos ellos situados en la misma máquina.

Para esta medición no se ha utilizado la aplicación de simulaciones ya que en este caso se ha buscado comprobar el funcionamiento de la aplicación para cualquier caso de uso. En su lugar se ha decidido que cada kernel realize cien peticiones de multiplicaciones de tamaño 2048 por 2048 a server. Como se puede observar en la figura 3.7, hemos creado dos procesos kernel y un proceso server.

Los resultados que hemos obtenido de TAU son los siguientes. Visualizados mediante la herramienta de TAU: ParaProf. Se ha generado la gráfica de la figura 3.8. En el caso de nuestra aplicación nos centramos en dos funciones fundamentales en la comunicación MPI: MPI_Send y MPI_Recv. Nuestro proyecto no incluye funciones de comunicación colectiva por tanto todo el tráfico de datos MPI circula a través de estas dos funciones y puede ser analizado al completo.

A primera vista encontramos una función con mayor peso que MPI_Send y MPI_Recv en *node 2*. Esta función es la encargada de gestionar todo el protocolo de comunicación de server e incluye el tiempo a las llamadas MPI_Send y MPI_Recv, por lo que analizando MPI_Send y MPI_Recv no hace falta centrarse en ella.

Antes de valorar los resultados medidos, el resultado esperado debería ser el siguiente: Por lo que respecta a MPI_Send los tres procesos deberían tener una carga similar. Mientras que **server** debería tener una carga de MPI_Recv cuatro veces mayor a la de ambos procesos **kernel**. Además las matrices son del mismo tamaño por lo que estas estimaciones son coherentes. Estimación gráficamentre mostrada en la figura 3.6.

La función MPI_Recv presenta un resultado contraintuitivo. Mientras que server debería ocupar mas tiempo que ambos **kernels** combinados, los resultados de las mediciones muestran que este toma menos tiempo en MPI_Recv que ambos por separado. Este resultado muestra un cuello de botella en nuestra aplicación.

Lo que está ocurriendo es que la medición de TAU también tiene en cuenta el tiempo en el que la llamada esta bloqueada esperando al comienzo del envío, ya que las llamadas MPI_Send y MPI_Recv son bloqueantes. Server solo puede atender a un kernel a la vez hasta terminar de gestionar la petición. Para solucionar este problema pueden tomarse diversas medidas que comentaremos en la siguiente sección.

Lo mismo sucede con MPI_Send para el caso de los procesos **kernel**. El envío del primer mensaje del protocolo (PETICION_CALCULO) queda bloqueado hasta que server haya finalizado la petición del otro **kernel**. De nuevo, las soluciones a este cuello de botella son comentadas en una sección posterior de este capítulo.

3.6 Evaluación

Como conclusión exponemos el balance final de los resultados de la aplicación. El resultado general se puede concluir que es favorable, ya que de la figura 3.9 se observa que se ha conseguido una mejora respecto a las condiciones iniciales de la aplicación. También se aprecia que, debido a que la máquina donde se ha ejecutado cuenta con 4 *cores*, el máximo paralelismo en una aplicación completamente eficiente que puede llegar a ser alcanzado es de 4 procesos: Un proceso **master**, un proceso **server** y dos procesos **kernel**. Sin embargo a causa de la comunicación síncrona de los envíos de mensajes MPI,

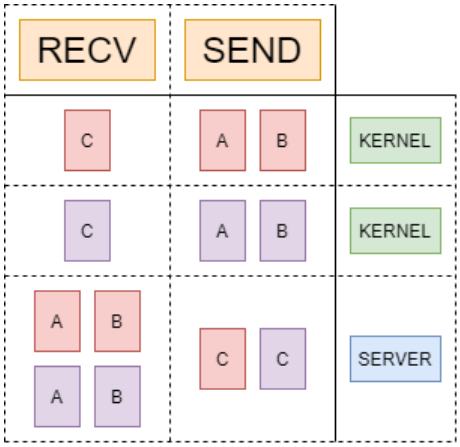


Figura 3.6: Tráfico esperado de comunicación MPI.

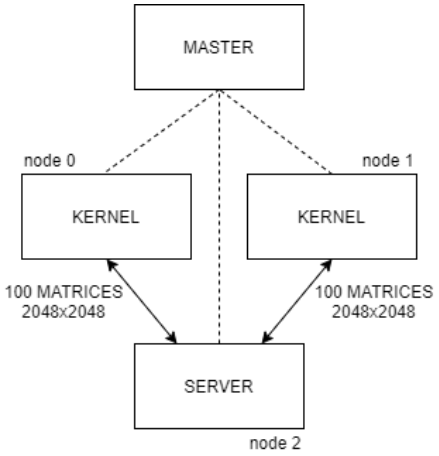


Figura 3.7: Esquema del entorno de ejecución de la medición con TAU.

un mayor número de procesos **kernel** puede llegar a mejorar el rendimiento tal y como se observa en la gráfica.

Por lo tanto se concluye que la utilidad de la aplicación es favorable y se han conseguido resultados satisfactorios respecto al punto de partida. Principalmente gracias a la división de la carga de trabajo entre diversos **kernel**, lo cual ofrece una ventaja directa.

Sin embargo los resultados podrían ser mejores si no fuera por el hecho de que hemos encontrado un problema al realizar las mediciones temporales. Este problema esta causado por el tiempo de envío de los datos a través de las distintas memorias de nuestros nodos. La información de la resolución del problema reside en memoria principal y en el momento de realizar una petición de cálculo, esta se envía a un proceso **server**. Este envío tiene dos fases que consumen tiempo las cuales procedemos a explicar.

La primera de ellas es el envío a través de la red, este envío depende del ancho de banda de la conexión de red entre las dos máquinas que se comunican y otros factores adicionales como el estado de carga de la red. Los cuales no serán explicados en detalle en este documento ya que no es el objetivo del mismo. Pese a todo, causan que el envío de datos entre dos máquinas sea mas costoso que la lectura de memoria principal de una máquina. El coste temporal de esta fase puede ser reducido enormemente si el proceso **server** y **kernel** que se estan comunicando residen en la misma máquina. Pese a todo el tiempo no será nulo en ningún caso ya que la comunicación MPI entre dos procesos siempre tiene asociada un *overhead*.

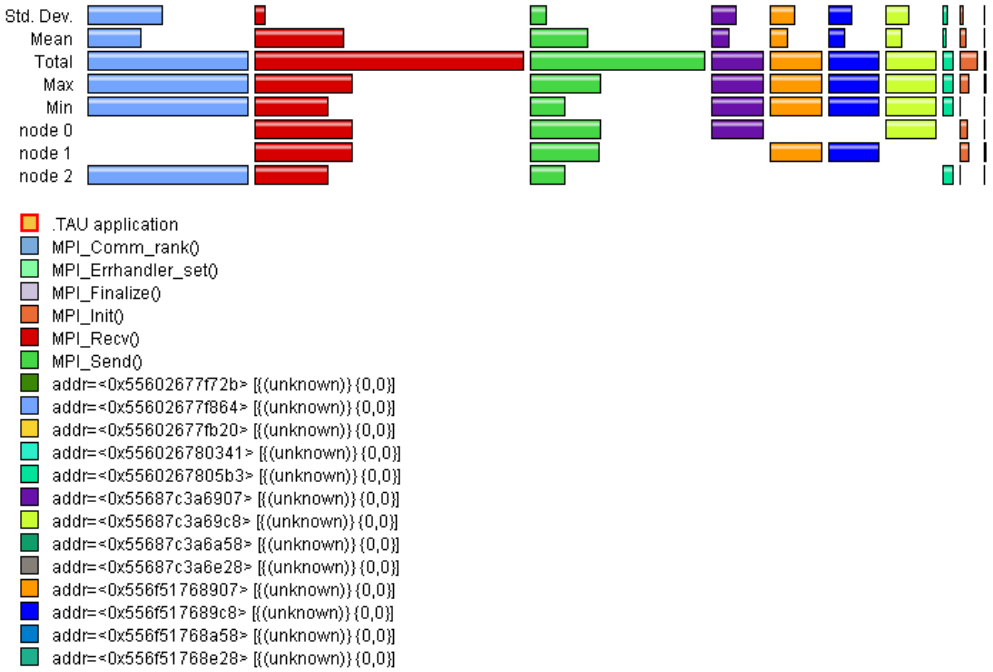


Figura 3.8: Resultados obtenidos con TAU en la medición de ejecución de la aplicación. *Node 0* y *Node 1* son procesos kernel mientras que *Node 2* es un proceso server

La segunda fase es el envío de memoria principal a memoria de la GPU. Este coste es necesario asumirlo siempre que busquemos realizar una operación en la GPU. El ancho de banda es mayor que el que encontramos en la comunicación entre dos máquinas a través de internet pero no es despreciable. Además por motivos del estado actual de la tecnología, siempre vamos a encontrar un ancho de banda menor que el que existe entre memoria principal y la CPU.

En la figura 3.10 mostramos los tiempos de ejecución del cálculo matricial de manera directa, es decir, sin utilizar nuestra aplicación. Simplemente creando las matrices y multiplicandolas en Fortran y CLBLAS sin realizar comunicación entre procesos ni envios MPI. El único coste adicional es el de la copia de memoria principal a memoria de GPU para el caso de CLBLAS. De estos datos podemos extraer la conclusión de que el sobrecoste de las comunicaciones es un factor clave.

Por lo tanto hemos de encontrar un balance entre la ventaja obtenida al multiplicar en GPU y el tiempo que tardamos en enviar los datos de las dos matrices a multiplicar y recibir la matriz resultado. Hemos realizado una medición bajo las siguientes condiciones: *Uso de dos máquinas donde una ejecuta un proceso **server** y la otra ejecuta un proceso **kernel** y **master**. El envío de datos es a través de la red, desconocemos el ancho de banda y la carga de la misma en el momento de la medición.*

En la siguiente tabla se observa que hasta un tamaño de dimensión de matrices cuadradas de 2048, el sistema de envíos y cálculo en GPU es mas lento. Esa cifra es ilustrativa del problema, el valor 2048 es el óptimo para estas condiciones (teniendo en cuenta que no hemos buscado valores intermedios en el incremento en potencias de dos de los tamaños). En otro entorno de ejecución el valor óptimo deberá ser recalculado, en la sección de posibles mejoras de la aplicación hacemos referencia a una posible solución a este cálculo manual de los valores óptimos.

Tamaño por dimensión	Fortran (segundos)	Server (segundos)
32	0.00002	0.22335
64	0.00010	0.21265
128	0.00073	0.41777
256	0.00540	0.22916
512	0.04300	0.27574
1024	0.33600	0.89736
2048	2.87800	1.84342
4096	51.93200	5.76084
8192	405.92186	29.38560

Como conclusión, los objetivos que han sido propuestos para esta aplicación son: La implementación de una comunicación MPI entre diversos computadores interconectados a través de la red y el de optimizar la aplicación de simulación numérica.

Respecto al primer objetivo se puede concluir que ha sido conseguido satisfactoriamente ya que la ventaja principal que se obtiene de nuestra aplicación es gracias al paralelismo ofrecido por la ejecución simultánea de diversos procesos en un numero determinado de computadores interconectados.

Respecto al segundo objetivo, la optimización obtenida es significativa como se ha demostrado y por ello el coste temporal de la simulación es menor que sin el uso de nuestra aplicación. Sin embargo es una ventaja inferior a la mejora esperada debido al problema del coste de las comunicaciones. Por ello todavía existen multitud de optimizaciones que podrían llevarse a cabo sobre la aplicación que sin duda conseguirían explotar los recursos disponibles de una forma mas eficiente para obtener mejores resultados. A continuación procedemos a comentar las optimizaciones mas interesantes que surgen de las conclusiones obtenidas experimentalmente.

3.7 Posibles mejoras

La implementación actual de la aplicación esta abierta a una serie de mejoras que podrían mejorar la eficiencia del cálculo y el tiempo de ejecución.

- Actualmente, por motivos de simplificar la complejidad de la programación del proceso **server**, las peticiones son atendidas de forma secuencial. Por lo que los **kernels** que esperan a que sus peticiones sean atendidas quedan bloqueados hasta que son atentidos. Si **server** utilizara diversos hilos para atender las peticiones entrantes el potencial de paralelismo de la estructura en diversos **kernel** sería aprovechado de una forma mas óptima.
- Distribución de la carga. Actualmente el proceso **master** reparte la carga a los kernels siguiendo una planificación Round-robin. Pero **master** no conoce la carga actual del proceso al que asigna el siguiente caso. Si **master** conociera la carga actual de cada proceso **kernel**, sería capaz de repartir la carga de una forma mas homogénea y de nuevo aprovechando el paralelismo de una forma mas óptima
- Por último, debido a la naturaleza del cálculo algebraico, es habitual que las matrices a operar no sean completamente distintas entre multiplicaciones. Es decir que es probable que la misma matriz aparezca en diversas operaciones a lo largo de la resolución de un sistema de ecuaciones. Por lo tanto nuestra aplicación debería ofrecer la funcionalidad de reutilizar las matrices para ahorrar el tiempo de envío de los datos.

Esta mejora podría realizarse de manera relativamente sencilla gracias al protocolo de comunicación existente. Ya que bastaría con extender su funcionalidad y utilizar una estructura de datos como un diccionario en el proceso **server** para guardar las matrices asignándoles un identificador.

- La operación mas costosa en la resolución de los sistemas de ecuaciones que son necesarios para realizar las simulaciones físicas de nuestro caso de uso son las multiplicaciones. Pero ello no significa que estas constituyan el total del tiempo de ejecución de la aplicación, hay otras operaciones que podrían ser aceleradas en GPU y por tanto ofrecer un mejor resultado. El protocolo de comunicación podría ser extendido para ofrecer esta nueva característica de nuestra aplicación.
- Cálculo automático del valor óptimo de cálculo en Fortran localmente frente al cálculo en una GPU en un proceso **server**. Este cálculo podría realizarse en el proceso **master** al inicio de la ejecución , antes de la creación de los procesos **server** y **kernel**. O incluso dinámicamente cada cierto periodo de tiempo dependiendo de la carga de la red en cada momento.

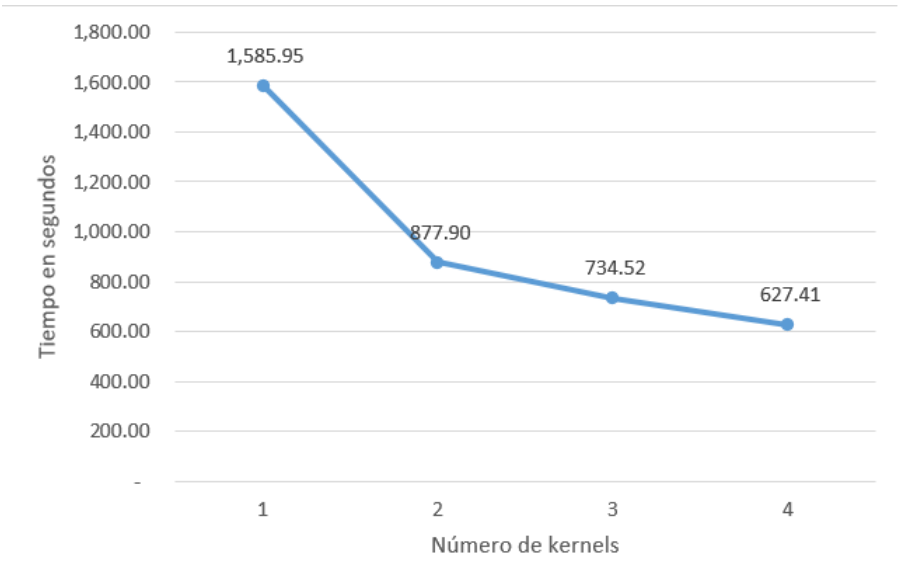


Figura 3.9: Tiempo de ejecución del cálculo de 2000 casos de simulación para diversos numeros de kernels. Ejecutados sobre una sola máquina de 4 *cores*.

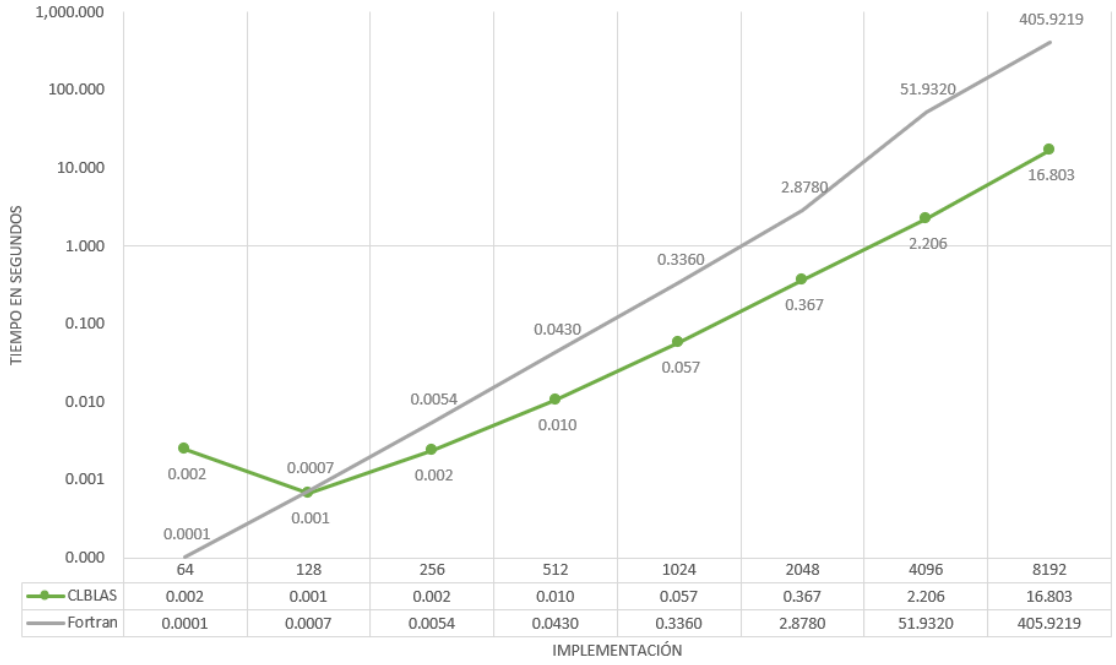


Figura 3.10: Comparativa de tiempos entre Fortran (CPU secuencial), CLBLAS (GPU GTX 1050ti) y CUBLAS (GPU GTX 1050ti). Doble precisión.

CAPÍTULO 4
Aplicación de entrenamiento de redes neuronales

En este capítulo describimos el trabajo realizado sobre una aplicación existente de entrenamiento de redes neuronales. Dicha aplicación, HELENNA, ofrece la posibilidad de entrenar una red neuronal seleccionado el conjunto de datos a utilizar para el entrenamiento así como la topología mediante la cual se va a llevar a cabo. Este proyecto, HELENNA, ha sido desarrollado en el grupo de arquitecturas paralelas de la Universitat Politècnica de València.

4.1 Descripción de la Aplicación

HELENNA (HEterogeneous LEarning Neural Network Application) es una aplicación para el entrenamiento e inferencia de redes neuronales. El ámbito de uso y aplicación es en docencia y en investigación ligado con proyectos Europeos actualmente en ejecución (H2020 RECIPE [9]). HELENNA es un proyecto que se ha iniciado en el Departamento de Informática de Sistemas y Computadores (DISCA) de la UPV por parte de investigadores de dicho departamento, todos ellos en el ámbito del grupo de investigación GAP.

Una de las características principales de HELENNA es su capacidad de llevar a cabo el entrenamiento en diferentes arquitecturas. Tales como CPUs, GPUs y FPGAs. Además, debido a que su desarrollo sigue en marcha, busca expandir su funcionalidad a nuevas arquitecturas tales como las TPU de Google [4] o las plataformas embarcadas de NVIDIA como es JETSON XAVIER [7]. La búsqueda de implementar la funcionalidad de HELENNA en un conjunto tan diverso de arquitecturas tiene como objetivo conseguir realizar el entrenamiento de redes neuronales precisando de un consumo energético y un coste temporal más bajo.

En la tabla 4.1 se muestran las arquitecturas que actualmente están soportadas por HELENNA. Entre ellas la arquitectura la cual hemos añadido al soporte de HELENNA, GPU-clBLAS. Estas arquitecturas se seleccionan para su utilización en el entrenamiento en la configuración del mismo junto con el tipo de topología, el conjunto de datos de entrenamiento y otras opciones adicionales.

Extender el soporte de HELENNA a otros dispositivos resulta sencillo. Ya que el desarrollo de este proyecto esta enfocado en facilitar la programación del soporte de nuevas arquitecturas de manera modular. Esto es así debido a que la codificación relativa a una arquitectura se concentra en un fichero de código y su respectivo *header*. La implementación de las capas que componen las topologías es independiente de la arquitectura sobre

CPU	Soportado
CPU-AVX	soportado
CPU-AVX512	en desarrollo
CPU-MKL	soportado
GPU-clBLAS	desarrollado en este TFG
GPU-cuBLAS	desarrollado en otro TFG
FPGA-OpenCL	en desarrollo

Tabla 4.1: Dispositivos soportados en HELENNA.

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

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

la que va a ser realizado y en última instancia mediante la sobrecarga de funciones se lleva a cabo la ejecución del programa en la arquitectura seleccionada. El lenguaje de programación sobre el que ha sido desarrollado HELENNA es exclusivamente *C*.

El análisis de la ejecución de la aplicación en lo que respecta al coste temporal de las funciones es una utilidad que permite HELENNA. Este análisis estadístico tiene un enfoque docente ya que permite reflejar los problemas que pueden surgir en la codificación de los módulos que añaden soporte a arquitecturas. Gracias a esta característica, el desarrollo del módulo GPU-clBLAS que se ha llevado a cabo en este proyecto ha sido mas eficiente, ya que ha permitido detectar problemas de codificación de manera mas sencilla.

De manera similar al desarrollo modular de las arquitecturas, la implementación de nuevas topologías también sigue un modelo de desarrollo de estas características. Las diferentes capas se implementan en ficheros en C diferentes y tienen una interfaz definida e uniforme. Las capas actuales que soporta HELENNA se muestran en la tabla 4.2. Además, las topologías que emplean todas estas capas se pueden encontrar en el apéndice de este documento A.0.6.

La extensión GPU-clBLAS permite el entrenamiento en cualquier tipo de topología que soporta HELENNA. Debido a que se han codificado todas las capas que forman parte de estas topologías. Pese a que sería interesante describir al completo todas las implementaciones de estas capas nos centraremos en un subconjunto de las mismas que sea representativo del trabajo realizado.

4.2 Soporte de GPU con CLBLAS

Al comienzo de este proyecto el soporte ofrecido por HELENNA incluía únicamente a la implementación CPU y MKL. El proyecto ha permitido el entrenamiento en GPU mediante apoyándonos en dos librerías, CLBLAS y OpenCL. La mayor parte de las funciones pertenecientes a las topologías a las que hemos dado soporte consisitían en una combinación de funciones de álgebra lineal por lo que han sido codificadas fácilmente mediante CLBLAS. Sin embargo algunas de ellas necesitaban una mayor complejidad en su implementación o simplemente no correspondían con ninguna rutina BLAS. Debido a esto también hemos tenido que implementar nuestros propios *Kernels* OpenCL para dar un soporte completo. Las figuras 4.1 y 4.2 muestran todas las funciones implementadas en este proyecto, indicando para cada una de ellas una breve descripción.

Función	Descripción	Soporte clBLAS	Función	Descripción	Soporte clBLAS
maxpooling	Explicado en este documento	kernel_OpenCL	**GEMM**	C = a * AB + b * C	clblasSgemm
demaxpooling	Explicado en este documento	kernel_OpenCL	**matmul**	C = AB	clblasSgemm
vect_step	Se asigna 1 a los valores mayores que cero y cero a los demás.	kernel_OpenCL	**matmul_bt**	matmul (B transpuesta)	clblasSgemm
vect_mult	Multiplicar elemento a elemento dos vectores.	kernel_OpenCL	**matmul_at**	matmul (A transpuesta)	clblasSgemm
vect_mult_add	Producto escalar de dos vectores	clblasSdot	**matadd_col**	Sumar un vector columna a la matriz	clblasSger
vect_mult_add_with_offset	Producto escalar de dos vectores pero el punto inicial comienza en *offset*.	clblasSdot	**matmul_elwise**	Multiplicar elemento a elemento dos matrices.	kernel_OpenCL
vect_scalar_product	A[x] = A[x] * b	clblasSscal	**matrix_transpose**	Transponer matriz	kernel_OpenCL
vect_max	C[x] = max(A[x], B[x])	kernel_OpenCL	**vector_to_matrix**	Copia el vector por columnas tantas veces como elementos haya en el vector.	clblasSgemm
vect_mult_add_with_stride	Producto escalar pero los elementos se encuentran a distancia de *stride*.	clblasSdot	**matsub**	$C[x] = A[x] - b * B[x]$	kernel_OpenCL
vect_V2subV1xK	B[x] = a * A[x] + b	clblasSaxpy	**matadd**	$C[x] = A[x] + b * B[x]$	kernel_OpenCL
matrix_softmax	Función activación softmax	kernel_OpenCL	**matrix_sigmoid_der**	Derivada función sigmoid.	kernel_OpenCL

Figura 4.1: Listado y descripción de las funciones implementadas en el proyecto HELENNA. Mediante un kernel OpenCL o utilizando una función CLBLAS.

También se han añadido ciertas funcionalidades al proyecto HELENNA que no estan directamente relacionadas con código OpenCL o clBLAS:

- Opción *-force_sync* para alternar entre una ejecución síncrona o asíncrona de las funciones encoladas en el contexto de OpenCL. Esta opción se selecciona en tiempo de ejecución al iniciar la aplicación.

- Extensión de los archivos que controlan la compilación automática del simulador para añadir mi módulo OpenCL/clBLAS.

A continuación se describe con mas detalle el soporte para la capa de normalización y la capa maxpooling, por tener un mayor detalle y complejidad y ser mas representativos del trabajo realizado.

Función	Descripción	Soporte clBLAS	Función	Descripción	Soporte clBLAS
vec_copy_with_stride_from_cpu_mem	Similar vec_copy_with_stride pero la matriz a ser copiada reside en memoria principal	clblasScopy	mat_copy	Copia una matriz en otra del mismo tamaño.	clblasScopy
zero_vec	Escribe todos los valores a 0.	kernel_OpenCL	mat_set	Se asigna el mismo valor a todos los elementos de la matriz	kernel_OpenCL
set_vec	Escribe todos los valores a un valor dado.	kernel_OpenCL	mat_reduce_rows	Conversión de matriz a vector sumando las filas.	kernel_OpenCL
vec_quantize	Función de cuantificación a un vector.	kernel_OpenCL	vec_copy	Copia de un vector en otro	clblasScopy
mat_quantize	Función de cuantificación a una matriz.	kernel_OpenCL	vec_copy_with_stride	Copia de un vector en otro pero los valores están a distancia *stride* .	clblasScopy
im2col	Conversión de una imagen digital en un vector.	kernel_OpenCL	vec_copy_with_offset	Copia de un vector en otro pero comenzando en la posición *stride* .	clblasScopy
im2col_bp_ant	No es necesaria en el entrenamiento	No	mat_A_plus_b_L1	D = A + b * signo(B)	kernel_OpenCL
cat_cross_entpy	No es utilizada en el entrenamiento	No	vec_axpy	B[x] = a * A[x] + B[x] (vector)	clblasSaxpy
col2im	Conversión de un vector en una imagen digital.	kernel_OpenCL	matrix_relu	Función activación. Se truncan a cero los valores menores a cero.	kernel_OpenCL
matset_random_ones	Asignación pseudo-aleatoria de 1's y 0's en una matriz.	kernel_OpenCL	batch_normalization_backward	Explicado en este documento	kernel_OpenCL
matrix_sigmoid	Función activación sigmoid.	kernel_OpenCL	batch_normalization_compute_gradients	Explicado en este documento	kernel_OpenCL
matrix_relu_der	Derivada función Relu. Similar pero los valores mayores a cero se igualan a uno.	kernel_OpenCL	batch_normalization_forward	Explicado en este documento	kernel_OpenCL

Figura 4.2: Listado y descripción de las funciones implementadas en el proyecto HELENNA. Mediante un kernel OpenCL o utilizando una función CLBLAS.

4.3 Implementación funciones utilizando CLBLAS/OpenCL

En esta sección comentaremos el trabajo de codificación del módulo OpenCL/CLBLAS para GPUs que ha supuesto nuestra aportación en el proyecto HELENNA. Para ilustrarlo, mostraremos a modo de ejemplo dos funciones que ponen de manifiesto las técnicas aprendidas y empleadas para la programación heterogénea y paralela que encontramos como eje principal de este proyecto.

Los conjuntos de datos de entrenamiento utilizados con el fin de observar el comportamiento de las topologías empleadas y las implementaciones realizadas son CIFAR-10 y MNIST. Los cuales son ampliamente conocidos en la comunidad de ML y habitualmente se utilizan para estudiar nuevas topologías y técnicas de entrenamiento. La ventaja que presenta utilizar estas bases de datos es que nos ahorran el trabajo de preprocesado y formateo de los datos y nos permiten centrar nuestro trabajo en la fase de entrenamiento y clasificación. Además nos permiten comparar nuestro trabajo con el de otros investigadores y desarrolladores.

- CIFAR-10 consiste en 60000 imágenes en color de 32x32 en 10 clases, con 6000 imágenes por clase. Hay 50000 imágenes de entrenamiento y 10000 imágenes de prueba. Las clases son las siguientes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. [1]

- MNIST consiste en un conjunto de imagenes que representan dígitos manuscritos con valores de 0 a 9. Los datos están formados por un conjunto de entrenamiento de 60.000 muestras, y un conjunto de pruebas de 10.000 muestras. [31]

4.3.1. Maxpooling y Demaxpooling

En primer lugar encontramos la rutina *maxpooling* y su inversa *demaxpooling* que han sido desarrolladas utilizando únicamente la librería OpenCL.

Maxpooling es un proceso de discretización utilizado en redes neuronales convolucionales [3]. En el propio proceso se lleva a cabo una disminución del número de parámetros de los que aprender, consiguiendo simplificar el problema y disminuir la complejidad espacial y temporal del mismo.

El punto de partida es una matriz bidimensional la cual, al tratarse de redes neuronales convolucionales, representa los valores de los píxeles de una imagen. Establecemos una ventana o *Kernel* de tamaño WxH y un salto de ventana o stride, que habitualmente es empleado utilizando un parámetro por dimensión, el cual determina el movimiento de la misma tanto en el eje vertical como en el horizontal. Para cada ventana se calcula el máximo de los valores que la forman y ese máximo se escribe en la matriz resultado. En la figura 4.3 se puede observar fácilmente el funcionamiento de esta función.

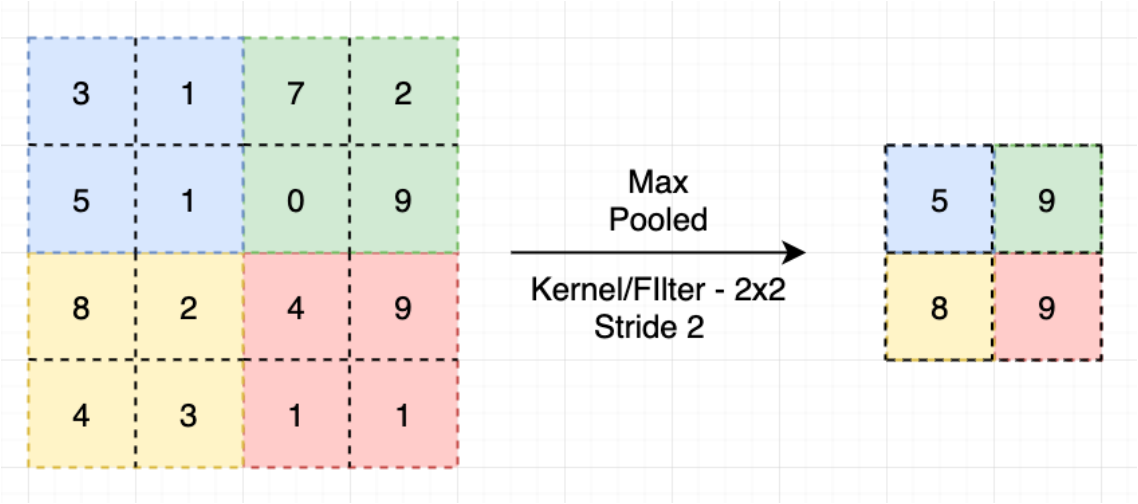


Figura 4.3: Gráfico conceptual del funcionamiento de maxpooling.

En la sección A.0.3 se puede observar el código secuencial que realiza el cálculo de maxpooling en CPU. Es el código sobre el que está basado la implementación paralela en GPU.

Estos son los parámetros que recibe la función que inicializa la rutina OpenCL.

```
int fn_maxpooling_clblas(int I, int WI, int HI, int SW,
                int SH, int KW, int KH, int O, int WO, int HO, int B,
                cl_mem mid_in, cl_mem mid_out, cl_mem mid_addr)
```

Los parámetros están descritos de forma gráfica en la figura 4.5. De manera adicional, hay tres parámetros que no estan descritos en la figura debido a que su representación gráfica es más compleja:

- I: Número de capas de la imagen input. Por ejemplo las capas RGB de la representación habitual de imagen digital.

- B: *Batch size*, número de imagenes con las que se trabaja de manera simultánea para explotar la capacidad de cómputo paralelo de la arquitectura.
- O: Número de capas de la imagen output. En el caso de la función maxpooling, este valor siempre es igual al número de capas de la imagen input.

A partir de estos parámetros se inicializa el cálculo en OpenCL. El número de elementos concurrentes que se crean es el siguiente: $O * B * WO * HO$. Es decir, cada elemento resultado de la matriz output calcula dentro de su propio *kernel size*, $KW * KH$, cual es el valor máximo y lo escribe en la matriz output. Para esto, como es lógico, su realiza una búsqueda del valor máximo recorriendo todos los elementos de kernel.

```
for (int kw = 0; kw < KW; kw++) {
    for (int kh = 0; kh < KH; kh++) {
        int addr_in_2 = addr_in + (kw * B) + (kh * WI * B);
        float value = ptr_in[addr_in_2];
        if (value > max_value) {
            max_value = value;
            max_addr = addr_in_2;
        }
    }
}
ptr_addr[addr_out] = max_addr;
ptr_out[addr_out] = max_value;
```

Tras llevar a cabo la implementación y utilizarla en diversas topologías para entrenar una red neuronal, los resultados que hemos obtenido entrenando el conjunto de entrenamiento de diez clases CIFAR-10 es el mostrado en la figura 4.4. Cada uno de los valores resultado corresponde al tiempo medio de ejecución de la función Maxpooling en cada una de las topologías presentadas. La diferencia de tiempos entre ambas plataformas muestra de forma evidente la ventaja que ofrece OpenCL al utilizar una GPU.

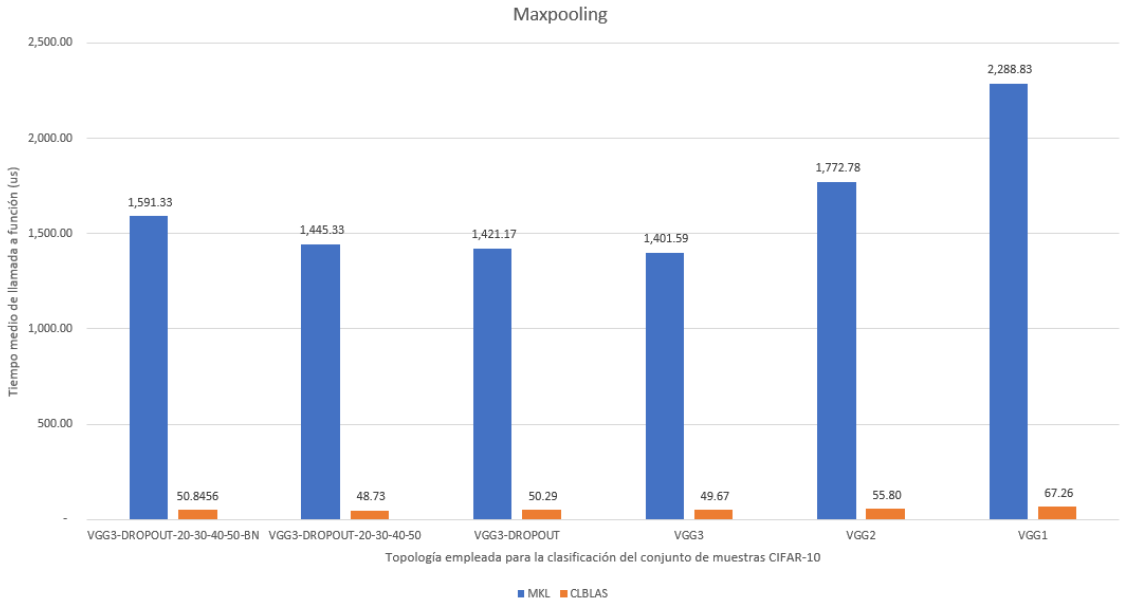


Figura 4.4: Comparativa de tiempos OpenCL (GPU) frente a MKL (CPU)

Podemos observar del código implementado que los valores máximos que se seleccionan de la matriz de entrada se guardan en cierta forma en la matriz auxiliar *ptr_addr*. De hecho, guardamos por cada pixel seleccionado su dirección en esta matriz. Esto es necesario ya que en el proceso de entrenamiento de la red neuronal necesitamos propagar el error en el sentido inverso a la topología de red. En el caso de la capa *maxpooling* debemos

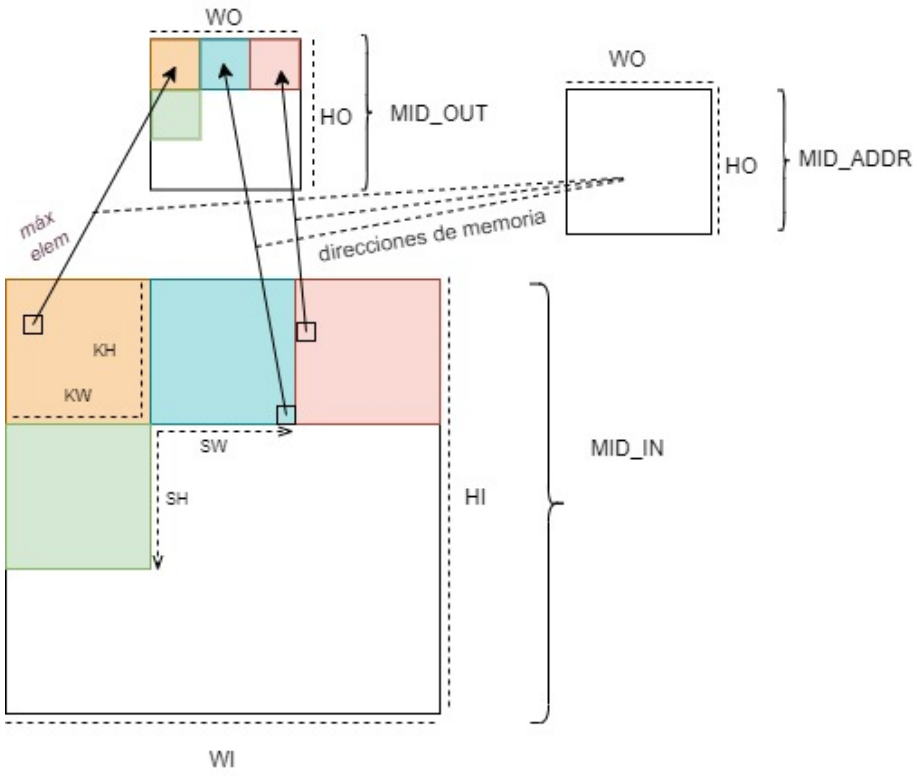


Figura 4.5: Diagrama que ilustra los parámetros de la función maxpooling

propagar el error de entrada a la capa solamente a los píxeles de entrada seleccionados. Para evitar realizar el cálculo de maxpooling de nuevo nos anotamos las direcciones en la matriz *ptr_addr* para utilizarla directamente en la fase de propagación del error.

En concreto, la función *demaxpooling* es muy sencilla ya que solamente debemos propagar el error de entrada en los píxeles apuntados por *ptr_addr*. Inicializamos la llamada al kernel OpenCL que lleva a cabo la función demaxpooling con el número de elementos concurrentes igual al número de elementos de la matriz output de demaxpooling. La tarea que debe llevar a cabo cada uno de ellos es la de propagar el error a la capa de entrada.

```
__kernel void demaxpooling(int SW, int SH, int O, int WO, int HO, int B,
    __global float *ptr_out, __global float *ptr_in, __global int *ptr_addr) {
   int tid = get_global_id(0);
    // calculo de direcciones para obtener addr_out
   int addr_in = ptr_addr[addr_out];
   ptr_in[addr_in] += ptr_out[addr_out];
}
```

Estos son los resultados del tiempo de ejecución que hemos obtenido con nuestra implementación OpenCL de la función demaxpooling comparados con la implementación en CPU 4.6.

4.3.2. Batch Normalization

En segundo lugar encontramos la función *Batch Normalization* (BN) que ha sido desarrollada utilizando conjuntamente rutinas CLBLAS y funciones específicas que no estaban cubiertas por las rutinas BLAS, codificadas expresamente para esta tarea utilizando OpenCL. Esta función es, en diferencia, la más compleja ya que precisa de un número elevado de cálculo con vectores y matrices.

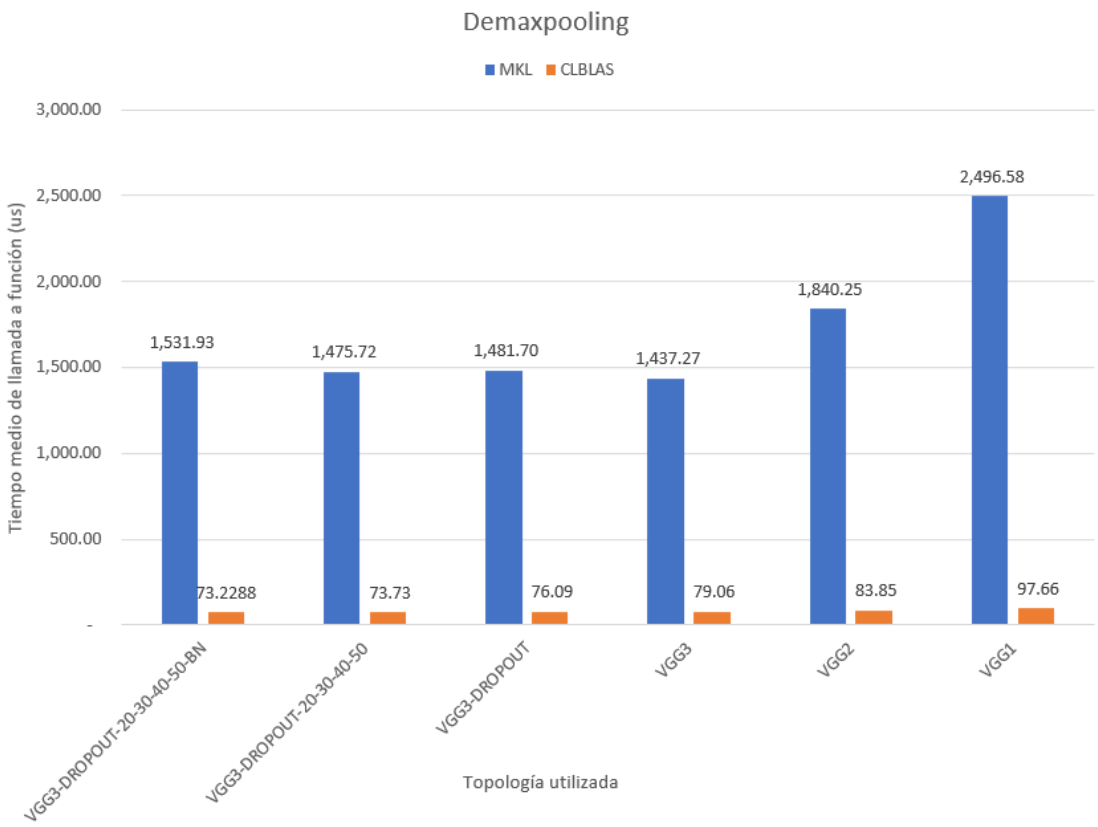


Figura 4.6: Resultados de la función demaxpooling para el conjunto de datos CIFAR-10 empleando distintos tipos de topologías.

BN es una técnica de ML la cual consigue reducir el tiempo en el que se alcanza la convergencia. La veracidad de esta afirmación está demostrada en el artículo original que introdujo esta técnica por primera vez en 2015 [20]. Batch normalization consiste en la normalización para cada dimensión de las muestras de entrenamiento que recibe cada capa de una red neuronal.

En concreto, BN centra y escala los datos de entrada de la capa neuronal, ofreciendo a la siguiente capa los datos normalizados. Los datos se normalizan para que tengan una media de 0 y una varianza de 1.

De manera adicional BN utiliza dos parámetros: γ y β. Los cuales se entrenan en cada iteración, para mejorar la efectividad de la técnica. Estos parámetros se utilizan solamente en el proceso de inferencia. Se puede observar de manera gráfica el resultado de aplicar BN a un conjunto de muestras. En la figura 4.9 se observa gráficamente el proceso de normalización en muestras de dos dimensiones.

La técnica de BN esta compuesta de dos fases. La fase *forward*, la cual aplica las normalizaciones a las muestras. Y la fase *backward*, la cual realiza la propagación del error y la actualización de los parámetros de las capas de la red neuronal. En la figura 4.8 se puede observar de manera esquematizada en forma de pseudocódigo el algoritmo de BN en su fase *forward* propuesto por Ioffe y Szegedy en la publicación previamente mencionada y sobre el que está basado nuestra implementación del mismo.

La implementación CLBLAS que se ha llevado a cabo del algoritmo BN está descompuesta en tres funciones. En la figura 4.7 se observa el funcionamiento de todas ellas, explicado a continuación:

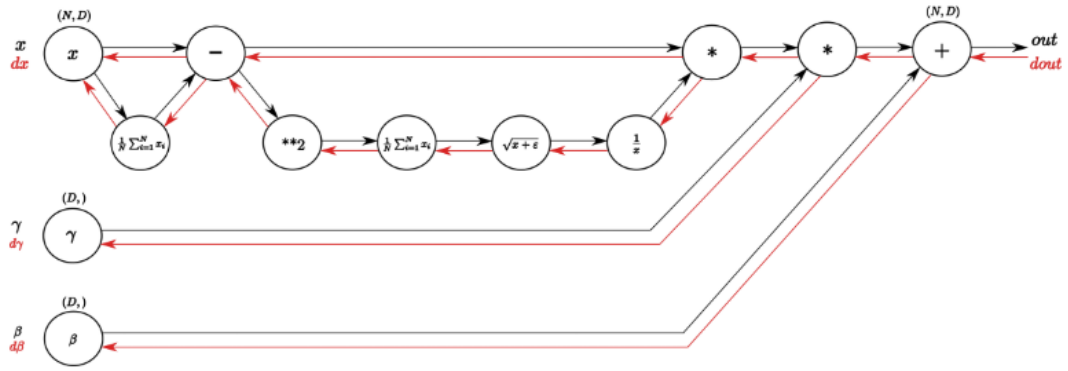


Figura 4.7: Esquema que muestra las tres funciones que forman BN. [21]

- fn_batch_normalization_clblas: Su algoritmo consiste en el recorrido de las flechas negras de izquierda a derecha del esquema. Coincide con el esquema de la figura 4.8.
- fn_batch_normalization_compute_gradients_clblas: Su algoritmo esta representado en el recorrido de derecha a izquierda en rojo que termina en los dos nodos que calculan $d\gamma$ y $d\beta$. De hecho, esta función calcula los gradientes a aplicar para actualizar los dos parámetros (γ y β).
- fn_batch_normalization_backward_clblas: Su algoritmo esta representado en el recorrido de derecha a izquierda en rojo que termina en el nodo que calcula dx. Estga función realiza la propagación del error a través de la capa BN.

Nuestra implementación de BN se ha servido de funciones tanto CLBLAS como funciones propias codificadas en OpenCL. En la figura A.0.5 se puede observar la implementación realizada de una de las tres funciones, en este caso fn_batch_normalization_clblas. Además de una función (step789_BN) en código OpenCL.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figura 4.8: Ioffe y Szegedy: "Batch Normalizing Transform, applied to activation x over a mini-batch". [21]

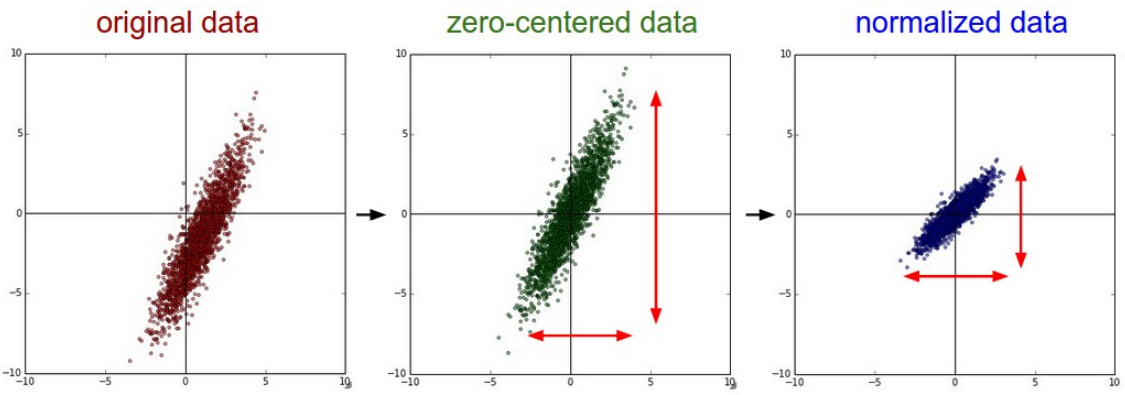


Figura 4.9: Preprocesamiento de los datos que lleva a cabo BN. Restando la media y escalando por la desviación típica cada dimensión. [3]

Hemos realizado mediciones del tiempo medio de ejecución de la función batch_normalization comparando la implementación en OpenCL que combina uso de GPU y CPU frente a la implementación en CPU la cual utiliza MKL. Estas mediciones han sido realizadas en un entrenamiento del conjunto de datos CIFAR-10 haciéndonos servir de una topología VGG3. En la figura 4.10 se encuentran los resultados, los cuales demuestran nuevamente la ventaja que ofrece una implementación heterogénea que combina el uso de GPU y CPU.

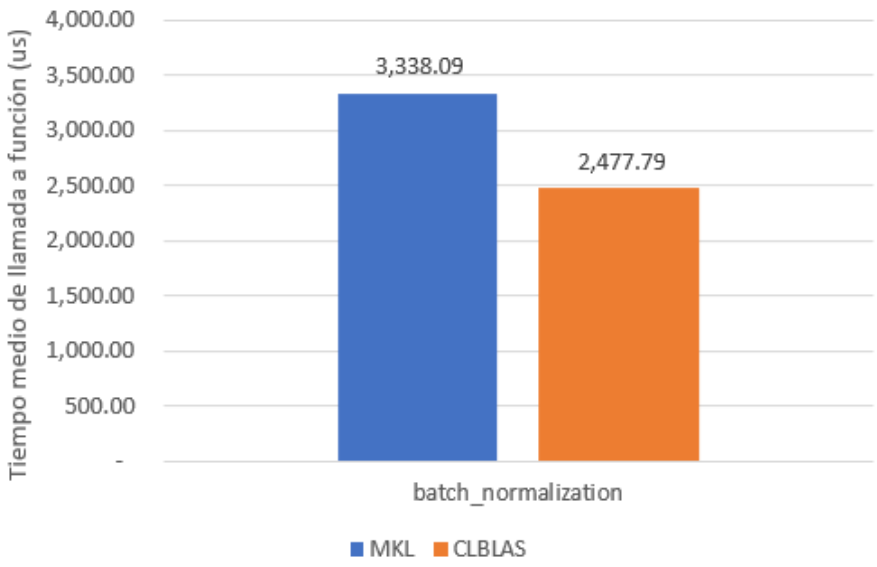


Figura 4.10: Comparativa del tiempo medio de ejecución en microsegudos entre la implementación MKL y la implementación CLBLAS en la llamada a la función batch_normalization.

4.4 Evaluación

Finalmente, en esta sección mostraremos los resultados obtenidos al realizar mediciones de los tiempos de ejecución de varias topologías de redes neuronales aplicadas sobre diversos conjuntos de datos explicados anteriormente: CIFAR-10 y MNIST.

En primer lugar comentaremos el entrenamiento de CIFAR-10. Mostramos los resultados que hemos obtenido respecto al tiempo de ejecución del mismo. Comparando las

prestaciones que ofrece una implementación MKL basada enteramente en CPU y una implementación OpenCL la cual se ayuda de la librería CLBLAS.

Los resultados que hemos obtenidos se encuentran en en la figura 4.11. Se puede observar que para el entrenamiento del conjunto de datos CIFAR-10 el coste temporal es significativamente menor para la implementación en OpenCL/CLBLAS. Consiguiendo esta ventaja para todas las topologías.

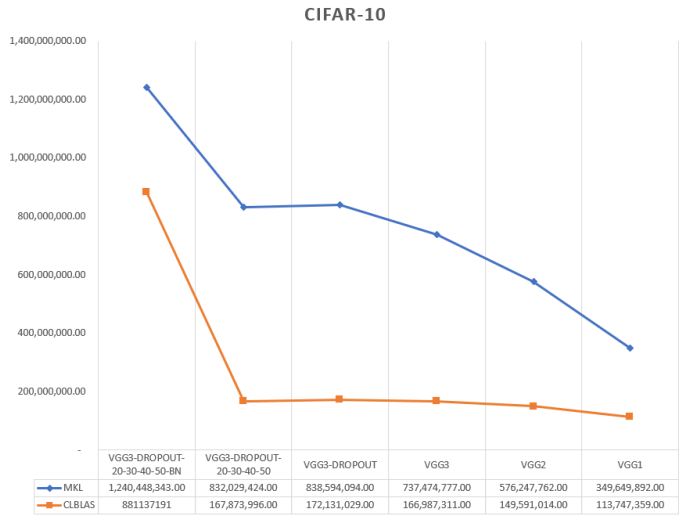


Figura 4.11: Comparativa del tiempo de entrenamiento en función de diferentes topologías sobre el conjunto de datos CIFAR-10. Comparando la implementación MLK y CLBLAS.

De manera adicional hemos incluido un figura que muestra la precisión del clasificador que se ha obtenido para ambos tipos de implementaciones en cada una de las topologías 4.12. Lo cual demuestra que la precisión obtenida es relativamente similar.

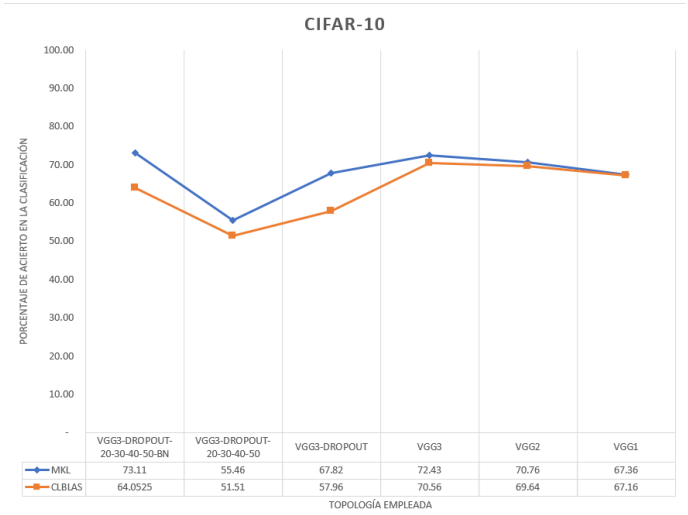


Figura 4.12: Precisión alcanzada en el entrenamiento del conjunto de datos CIFAR-10. Comparando la implementación CLBLAS y MKL.

Sin embargo al realizar esta comparativa para el conjunto de datos MNIST (Figura 4.13), encontramos unos valores en la que son llamativos. La implementación en MKL es más rápido que CLBLAS para el entrenamiento de las topologías convolucionales. Con tal de buscar el motivo que causa esta anomalía hemos comenzado estudiando detalladamente los costes de ejecución de las funciones que intervienen en los entrenamientos. En el anexo hemos incluido la figura A.2 la cual muestra en detalle un estudio del uso

de las funciones que son utilizadas en el entrenamiento de MNIST en tres tipos de topologías: Conv y Conv16, por ser las que presentan esta anomalía y MLP_Large por ser la que mayor ventaja obtenía respecto a MKL. Buscando comparar cual es la diferencia que causa este problema.

Figura 4.13: Comparativa del tiempo de entrenamiento en función de diferentes topologías sobre el conjunto de datos MNIST. Comparando la implementación MLK y CLBLAS.

Se puede observar que la función de multiplicación de matrices *matmul* consume gran parte del porcentaje del tiempo de entrenamiento en CLBLAS. 70 % para el caso de la topología Conv. Mientras que la misma función en la topología MLP_Large toma un 22 % del tiempo de entrenamiento. Además si comparamos el tiempo medio de llamada a esa función en CLBLAS entre ambas topologías observamos que es cuatro veces mas lenta en la topología convolucional. Como estamos utilizando la misma implementación de la función en ambos casos, la diferencia ha de deberse a las características de las matrices a multiplicar.

Hemos comparado los tamaños de las matrices que aparecen con mas frecuencia en las multiplicaciones de las tres topologías (Figura 4.14). Encontramos que las matrices que son multiplicadas en la topología MLP_Large son mucho mas cuadradas que en el caso de las topologías convolucionales. De esta manera extraemos la conclusión de que la implementación en MKL de la multipliación de matrices es capaz de tratar de manera mas eficiente las multiplicaciones de matrices rectangulares.

	CONV_16				CONV				MLP_LARGE			
	A		B		A		B		A		B	
	Dim 1	Dim2	Dim 1	Dim2	Dim 1	Dim2	Dim 1	Dim2	Dim 1	Dim2	Dim 1	Dim2
Matriz 1	16	9	9	50176	16	9	9	50176	2000	2000	2000	64
Matriz 2	10	3136	3136	64	10	196	196	64	2000	784	784	64

Figura 4.14: Tamaño de las dos matrices mas comunes que aparecen en las topolgías CONV_16, CONV y MLP_Large.

Finalmente, de manera similar a la información relativa a la precisión que hemos mostrado de CIFAR-10 también incluimos la precisión alcanzada para MNIST (Figura 4.15). La cual muestra unos resultados de precisión en los que la implementación CLBLAS supera en algunas topologías a los resultados ofrecidos por MKL.

Como conclusión, recordamos que los objetivos que se pretendían alcanzar tras el trabajo realizado en esta aplicación son: Desarrollo de una interfaz OpenCL que satisfaga todas las necesidades del cálculo para GPUs utilizando la librería matemática CLBLAS; Soporte OpenCL de las capas utilizadas en el entrenamiento de redes neuronales presentes en HELENNA.

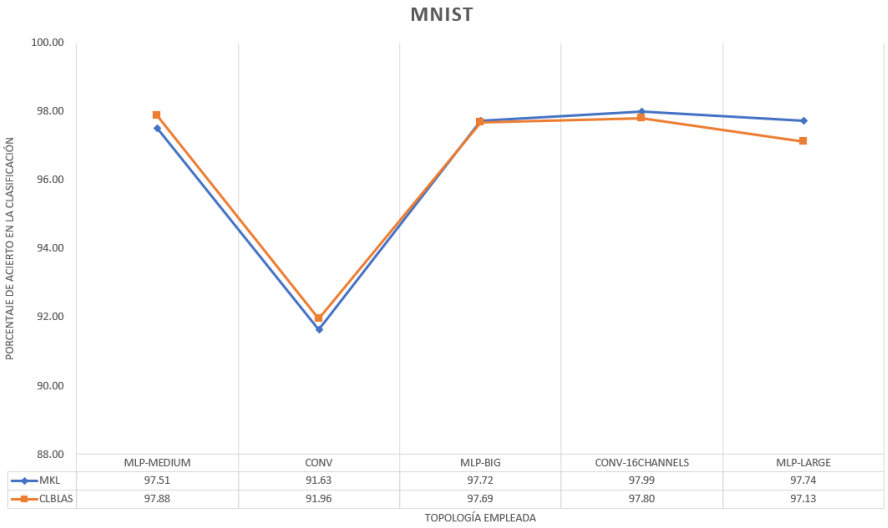


Figura 4.15: Precisión alcanzada en el entrenamiento del conjunto de datos MNIST. Comparando la implementación CLBLAS y MKL.

Ambos objetivos han sido cumplidos satisfactoriamente como se ha demostrado. Ya que hemos conseguido extender la funcionalidad del la aplicación de entrenamiento de redes neuronales para OpenCL en todas las topologías implementadas en HELENNA. Consiguiendo ofrecer resultados del coste temporal del entrenamiento considerablemente mas rápido que el realizado en CPU.

Como puntos negativos a destacar y en los que podemos continuar mejorando nuestra extensión para OpenCL son: La precisión del entrenamiento obtenido, la cual tiende a ser inferior a la precisión obtenida en MKL; Optimización de las implementaciones de las funciones. Las cuales tienen un amplio margen de mejora si comparamos nuestros resultados con las herramientas presentes en el mercado de entrenamiento de redes neuronales en GPU como Keras [5] o PyTorch [8] entre otras.

CAPÍTULO 5
Conclusiones

En este capítulo de cierre se presenta una recapitulación de las principales metas alcanzadas durante la realización de este proyecto, el cual cubre diversas aplicaciones enmarcadas en un contexto común: explorar el uso de programación para arquitecturas heterogéneas.

En primer lugar, en lo relativo a la aplicación para el modelado y simulación del comportamiento dinámico de puentes, pueden extraerse las siguientes conclusiones generales:

- Se ha tomado contacto directo con las necesidades de la aplicación, comprendiendo sus problemáticas, interés y objetivos principales, relacionados todos ellos con cuestiones de ingeniería civil de importante repercusión social, relativas al proyecto y conservación de puentes ferroviarios.
- Se ha realizado una labor de estudio y práctica de varios meses de duración para familiarizarse con la programación mixta en lenguajes C y Fortran.
- Se han conseguido mejoras apreciables en la velocidad de cálculo mediante la paralelización basada en MPI. Principalmente debido a la posibilidad que ofrece nuestra aplicación de instanciar procesos en diversas máquinas remotas o locales, con los que dividir la carga de trabajo.
- Se han conseguido también mejoras sensibles en la velocidad de cálculo mediante el uso de clBLAS y OpenCL, a partir de determinado tamaño de matrices que viene determinado por la transferencia de información de memoria principal a la GPU. Donde hemos demostrado que la aceleración del cálculo obtenida por una GPU es determinante.
- El desarrollo de la aplicación para simulación física ha sido llevado gracias a una colaboración entre la facultad de Ingeniería informática (ETSINF) y la facultad de Ingeniería de Caminos, Canales y Puertos (ETSICCP) de la UPV.
- **Como resultado a destacar, se ha presentado una ponencia en el congreso internacional IRAS (International Symposium on Risk Analysis and Safety of Complex Structures and Components), celebrado en Oporto en Julio de 2019** [22].
- Posibles mejoras de cara al futuro se han descrito detalladamente en el capítulo 3.

En segundo lugar, y en lo relativo a la aplicación para entrenamiento de redes neuronales, pueden extraerse las siguientes conclusiones generales:

- Se ha tomado contacto directo con las necesidades de la aplicación, comprendiendo sus problemáticas, interés y objetivos principales, relacionados con el entrenamiento de redes neuronales y sus necesidades derivadas del alto coste temporal de los entrenamientos para las topologías y conjuntos de datos actuales.
- Se ha conseguido ampliar la funcionalidad de HELENNA, permitiendo el entrenamiento de redes neuronales en arquitecturas heterogéneas, dando soporte a diversas topologías y las capas de las que están formadas. Además se ha conseguido ofrecer una implementación que reduce el coste temporal del entrenamiento de un número considerable de topologías.
- La participación en proyectos de gran complejidad organizativa, como en el caso de HELENNA, ha supuesto un aprendizaje y un refuerzo de las técnicas de programación adquiridas durante la carrera, ya que era necesario una correcta utilización de herramientas de programación que permitan el desarrollo de software de manera escalable y organizada (como por ejemplo el uso de un sistema de control de versiones como Git, división del código fuente en headers, o uso de la herramienta de compilación automática GNU Make y CMake).
- Posibles mejoras de cara al futuro se han descrito detalladamente en el capítulo 4.

Se quiere destacar también que, tanto en el proyecto de simulación física como durante mi participación en el proyecto HELENNA, ha sido necesaria la colaboración con diversos desarrolladores. Esta colaboración ha supuesto la puesta en práctica de habilidades de trabajo en equipo y, personalmente, la considero una provechosa experiencia.

Todo el transcurso de proyecto ha sido llevado a cabo en el laboratorio del *Grupo de arquitecturas paralelas* perteneciente al departamento DISCA. Esto ha servido para hallarse en un entorno de desarrollo de proyectos con un enfoque común al nuestro: el de la computación de altas prestaciones y las arquitecturas paralelas. Esta circunstancia ha supuesto una valiosa oportunidad para conocer más en profundidad el contexto actual de las herramientas y de las tecnologías utilizadas en este campo de la informática.

Bibliografía

[1] CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html.

[2] clMathLibraries. https://github.com/clMathLibraries.

[3] CS231n Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/neural-networks-2/.

[4] Google TPU. https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[5] Keras website. https://keras.io/.

[6] MPI - C Examples. https://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html.

[7] Nvidia Jetson Xavier. https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[8] PyTorch website. https://pytorch.org/.

[9] REliable Power and time-ConstraInts-aware Predictive management ofheterogeneous Exascale syster http://www.recipe-project.eu/.

[10] TAU - Tuning and Analysis Utilities. https://www.cs.uoregon.edu/research/tau/home.php.

[11] TAU by example. https://wiki.mpich.org/mpich/index.php/TAU_by_example.

[12] The Khronos Group. https://www.khronos.org/, Jun 2020.

[13] Blaise Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/.

[14] Vaughn Betz. Fpga architecture for the challenge.

[15] Jack Dongarra, Jeremy Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16:1–17, 03 1990.

[16] Jack Dongarra, Jeremy Croz, Sven Hammarling, and Richard Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software - TOMS*, 14:1–17, 03 1988.

[17] Jack Dongarra, Jeremy Croz, Sven Hammarling, and Richard Hanson. A proposal for an extend set of fortran basic linear algebra subprograms. *ACM SIGNUM Newsletter*, 20:2–18, 11 1992.

[18] Stephen A. Dyer and Brian K. Harms. Digital signal processing. volume 37 of *Advances in Computers*, pages 59 – 117. Elsevier, 1993.

[19] Matthew Gilbert, Clive. Melbourne, C. Smith, L Augusthus Nelson, and Gm Swift. Proposed permissible limit state assessment criteria for masonry arch bridges. 2016.

[20] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[21] Frederik Kratzert. Understanding the backward pass through Batch Normalization Layer. https://kratzert.github.io/2016/02/12/understanding-the-gradient-\flow-through-the-batch-normalization-layer.html.

[22] P. Museros L. Medina, M. Gavilán, R. Palma, and J. Flich. Updated parallel computing strategies for nondeterministic dynamic analysis of railway bridges. *First International Symposium on Risk Analysis and Safety of Complex Structures and Componen*, pages 199–200, 2019.

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

[24] Rasa Remenyte-Prescott Matteo Vagnoli and John Andrews. Railway bridge structural health monitoring and fault detection: State-of-the-art methods and future challenges. *Structural Health Monitoring*, 17(4):971–1007, 2018.

[25] Cedric Nugteren. Publications. https://cnugteren.github.io/publications.html.

[26] Pedro Museros Romero. *Interacción vehículo-estructura y efectos de resonancia en puentes isostáticos de ferrocarril para líneas de alta velocidad*. PhD thesis, 2002.

[27] Karl Rupp, Andreas Will August 19, Karl Rupp Post authorAugust 19, and Freddie September 8, Jun 2013.

[28] CORPORATE The MPI Forum. Mpi: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, page 878–883, New York, NY, USA, 1993. Association for Computing Machinery.

[29] Neil Trevett, Andrew Richards, Mark Butler, Jeff McVeigh, Anshuman Bhat, Balaji Calidas, Vincent Hindriksen, and Weijin Dai. Opencl - the open standard for parallel programming of heterogeneous systems, Jul 2013.

[30] Matteo Vagnoli, Rasa Remenyte-Prescott, and John Andrews. Railway bridge structural health monitoring and fault detection: State-of-the-art methods and future challenges. *Structural Health Monitoring*, 17(4):971–1007, 2017.

[31] Corinna Cortes Yann LeCun and Chris Burges. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/.

APÉNDICE A
Código utilizado

A.0.1. Implementación simple de la función GEMM.

```
// First naive implementation
__kernel void myGEMM1(const int M, const int N, const int K,
                      const __global float* A,
                      const __global float* B,
                      __global float* C) {

    // Thread identifiers
    const int globalRow = get_global_id(0); // Row ID of C (0..M)
    const int globalCol = get_global_id(1); // Col ID of C (0..N)

    // Compute a single element (loop over K)
    float acc = 0.0f;
    for (int k=0; k<K; k++) {
        acc += A[k*M + globalRow] * B[globalCol*K + k];
    }

    // Store the result
    C[globalCol*M + globalRow] = acc;
}
```

A.0.2. Implementación con uso de memoria local basada en tiles de la función GEMM.

```
// Tiled and coalesced version
__kernel void myGEMM2(const int M, const int N, const int K,
                      const __global float* A,
                      const __global float* B,
                      __global float* C) {

    // Thread identifiers
    const int row = get_local_id(0); // Local row ID (max: TS)
    const int col = get_local_id(1); // Local col ID (max: TS)
    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    // Initialise the accumulation register
    float acc = 0.0f;

    // Loop over all tiles
```

```
const int numTiles = K/TS;
for (int t=0; t<numTiles; t++) {

    // Load one tile of A and B into local memory
    const int tiledRow = TS*t + row;
    const int tiledCol = TS*t + col;
    Asub[col][row] = A[tiledCol*M + globalRow];
    Bsub[col][row] = B[globalCol*K + tiledRow];

    // Synchronise to make sure the tile is loaded
    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the computation for a single tile
    for (int k=0; k<TS; k++) {
        acc += Asub[k][row] * Bsub[col][k];
    }

    // Synchronise before loading the next tile
    barrier(CLK_LOCAL_MEM_FENCE);
}

// Store the final result in C
C[globalCol*M + globalRow] = acc;
}
```

A.0.3. Implementación en CPU de la función Maxpooling.

```
int fn_maxpooling_cpu(int I, int WI, int HI, int SW, int SH, int KW, int KH,
    int O, int WO, int HO, int B, type * ptr_in, type *ptr_out, int *ptr_addr)
    {

    int i, wi, hi, kw, kh, b;
    int addr_in;
    int addr_out;
    int channel_in_size = WI * HI * B;
    int row_in_size = WI * B;
    int point_in_size = B;
    int channel_out_size = WO * HO * B;
    int row_out_size = WO * B;
    int point_out_size = B;

    // in the forward we annotate the input pixel that conforms each output
        pixel
    memset(ptr_addr, 0, O * WO * HO * B * sizeof (int));

    int n = 5;
    for (i = 0; i < I; i++) {
        int addr_in1 = i * channel_in_size;
        int addr_out1 = i * channel_out_size;
        for (wi = 0; wi < WI; wi = wi + SW) {
            int addr_in2 = addr_in1 + (wi * point_in_size);
            int addr_out2 = addr_out1 + (wi / KW * point_out_size);
            for (hi = 0; hi < HI; hi = hi + SH) {
                int addr_in3 = addr_in2 + (hi * row_in_size);
                int addr_out3 = addr_out2 + (hi / KH * row_out_size);
                for (b = 0; b < B; b++) {
                    int addr_in4 = addr_in3 + b;
                    int max_addr = addr_in4;
                    type max_value = ptr_in[max_addr];
                    for (kw = 0; kw < KW; kw++) {
                        int addr_in5 = addr_in4 + (kw * point_in_size);
                        for (kh = 0; kh < KH; kh++) {
                            addr_in = addr_in5 + (kh * row_in_size);
```

```
                                type value = ptr_in[addr_in];
                                if (value > max_value) {
                                    max_value = value;
                                    max_addr = addr_in;
                                }
                            }
                        }
                        addr_out = addr_out3 + b;
                        ptr_out[addr_out] = max_value;
                ptr_addr[addr_out] = max_addr;

                    }
                }
            }
        }
        return 1;
    }
```

A.0.4. Modulo Fortran que realiza la llamada a C.

```
module cuda_matmul
implicit none

!integer, parameter :: ind1 = 2560, ind2 = 2560, ind3 = 256
integer, parameter :: ind1 = 1, ind2 = 1, ind3 = 1

  interface ! declaracion de la funcion externa en C, para el compilador fortran
  subroutine cfun(c, a, b, af, ac, bc)
      integer*4 :: af, ac, bc
      real*8   :: a(af,ac), b(ac,bc), c(af,bc)
  end subroutine cfun
  end interface

interface matmul_cuda
  procedure matmul_cuda_m_m, matmul_cuda_m_vc,
+            matmul_cuda_vf_m, matmul_cuda_vf_vc
end interface matmul_cuda

contains

function matmul_cuda_m_m(a,b) result(c) ! - variante para hacer matriz por matriz -
    - - - - - - - - - - - - -

real*8                              :: a(:,:), b(:,:)
real*8, dimension(size(a,1),size(b,2)) :: c

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

end function matmul_cuda_m_m

function matmul_cuda_m_vc(a,b) result(c) ! - variante para hacer matriz por vector
    columna - - - - - - - - - - - - - -

real*8                    :: a(:,:), b(:)
real*8, dimension(size(a,1)) :: c

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

```
  c = matmul(a,b)
endif
end function matmul_cuda_m_vc

function matmul_cuda_vf_m(a,b) result(c) ! - variante para hacer vector fila por
    matriz - - - - - - - - - - - - - -

real*8                    :: a(:), b(:,:)
real*8, dimension(size(b,2)) :: c

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

end function matmul_cuda_vf_m

function matmul_cuda_vf_vc(a, b) result(c) ! - variante para hacer vector fila por
    vector columna - - - - - - - - - - - - - -

real*8               :: a(:), b(:)
real*8               :: c
real*8, dimension(1,1) :: c_matrix

if (size(a,1)*size(a,1).gt.(ind1*ind2*ind3)) then
call cfun(c_matrix,a,b,1,size(a,1),1)
c = c_matrix(1,1)
else
c = dot_product(a,b)
end if

end function matmul_cuda_vf_vc

end module cuda_matmul
```

A.0.5. Implementación fn_batch_normalization_clblas

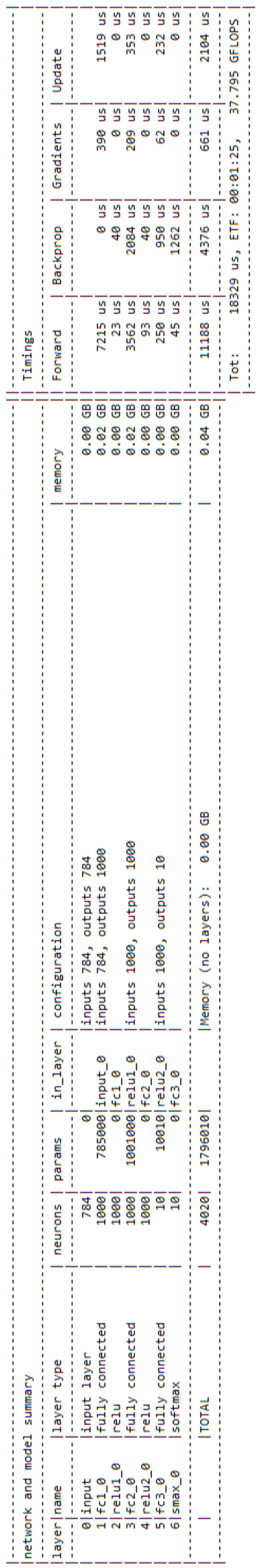
A.0.6. Simulador HELENNA

```
void fn_batch_normalization_clblas(int n, int B, cl_mem ptr_gamma,
                                   cl_mem ptr_beta, float eps, cl_mem ptr_mu,
                                   cl_mem ptr_xmu, cl_mem ptr_sq, cl_mem ptr_var,
                                   cl_mem ptr_sqrtvar, cl_mem ptr_ivar,
                                   cl_mem ptr_xhat, cl_mem ptr_gammax,
                                   cl_mem ptr_in, cl_mem ptr_out)
{
    // n parameter is the size of the input data, B is the batch size
    // n represents the first dimension and B the second dimension

    // step 1. Calculate mean: mu = 1./N * np.sum(x, axis = 0)
    int step = 0;
    reservar_one_buffer(B);
    fn_set_vec_clblas(0.0, ptr_mu, n);
    fn_sgemv_clblas(ptr_in, clblas_one_buffer, n, B, ptr_mu);
    fn_vect_scalar_product_clblas(1.0 / B, ptr_mu, n);

    // step 2: susbstract mean vector of every training example: xmu = x - mu
    fn_mat_vector_scale_clblas(-1.0, ptr_in, n, B, ptr_mu, ptr_xmu);

    // step 3: sq = xmu^2
    fn_elemwise_pow2(ptr_xmu, ptr_sq, n * B);

    // step 4: calculate variance: var = 1./N * np.sum(sq, axis = 0)
    fn_sgemv_clblas(ptr_sq, clblas_one_buffer, n, B, ptr_var);
    fn_vect_scalar_product_clblas(1.0 / B, ptr_var, n);

    // step 5: add eps for numerical stability, then sqrt: sqrtvar = np.sqrt(var +
        eps)
    //  sqrtvar = var + eps
    fn_vec_copy_clblas(ptr_var, ptr_sqrtvar, n);
    fn_vector_plusadd(eps, ptr_sqrtvar, n);
    fn_elemwise_sqrt(ptr_sqrtvar, ptr_sqrtvar, n);

    // step 6: invert sqrtvar
    fn_elemwise_invert(ptr_sqrtvar, ptr_ivar, n);

    // step 7: execute normalization
    // step 8: nor the two transformations steps (apply gamma)
    // step 9: out (apply beta)
    fn_step789_BN(ptr_ivar, ptr_xhat, ptr_xmu, ptr_gamma,
                  ptr_gammax, ptr_beta, ptr_out, n, B);
}

__kernel void step789_BN(
    __global float *ptr_ivar,
    __global float *ptr_xhat,
    __global float *ptr_xmu,
    __global float *ptr_gamma,
    __global float *ptr_gammax,
    __global float *ptr_beta,
    __global float *ptr_out,
    int n, int B)
{
    int global_row_id = get_global_id(0);
    int global_col_id = get_global_id(1);
    int element_id = global_col_id + global_row_id * B;

    ptr_xhat[element_id] = ptr_xmu[element_id] * ptr_ivar[global_row_id];
    ptr_gammax[element_id] = ptr_gamma[global_row_id] * ptr_xhat[element_id];
    ptr_out[element_id] = ptr_gammax[element_id] + ptr_beta[global_row_id];
}
```

Figura A.1: Función BN

	CONV				CONV16				MLP_LARGE			
OPENCL/CLBLAS	total calls	total time	%	us/call	total calls	total time	%	us/call	total calls	total time	%	us/call
im2col	5,466.00	401,061.00	0.97	73.37	5,466.00	487,067.00	0.69	89.11				
maxpooling	5,466.00	127,929.00	0.31	23.40	5,466.00	224,048.00	0.32	40.99				
demaxpooling	4,686.00	227,044.00	0.55	48.45	4,686.00	283,862.00	0.40	60.58				
matmul	24,990.00	30,162,672.00	73.27	1,206.99	24,990.00	33,705,334.00	47.88	1,348.75	143,694.00	35,982,381.00	22.75	250.41
matadd_col	10,932.00	989,781.00	2.40	90.54	10,932.00	1,211,309.00	1.72	110.80	54,660.00	11,028,023.00	6.97	201.76
matmul_elwise	4,686.00	105,654.00	0.26	22.55	4,686.00	208,736.00	0.30	44.54	42,174.00	1,264,772.00	0.80	29.99
vect_scalar_prod	37,488.00	2,110,040.00	5.13	56.29	37,488.00	2,321,650.00	3.30	61.93	187,440.00	17,015,771.00	10.76	90.78
matsub	23,429.00	801,767.00	1.95	34.22	23,429.00	686,758.00	0.98	29.31	98,405.00	8,298,598.00	5.25	84.33
matadd	18,744.00	677,958.00	1.65	36.17	18,744.00	571,178.00	0.81	30.47	93,720.00	8,172,081.00	5.17	87.20
mat_reduce_rows	9,372.00	4,446,811.00	10.80	474.48	9,372.00	29,497,130.00	41.90	3,147.37	46,860.00	2,073,681.00	1.31	44.25
matrix_relu	5,466.00	110,805.00	0.27	20.27	5,466.00	186,145.00	0.26	34.06	49,194.00	1,818,505.00	1.15	36.97
matrix_relu_der	4,686.00	102,609.00	0.25	21.90	4,686.00	159,663.00	0.23	34.07	42,174.00	1,243,845.00	0.79	29.49
matrix_softmax	5,466.00	280,869.00	0.68	51.38	5,466.00	233,669.00	0.33	42.75	5,466.00	3,791,188.00	2.40	693.59
xentropy	5,465.00	262,372.00	0.64	48.01	5,465.00	282,749.00	0.40	51.74	5,465.00	52,471,918.00	33.18	9,601.45
mat_copy	4,686.00	356,057.00	0.86	75.98	4,686.00	334,555.00	0.48	71.39	4,686.00	436,295.00	0.28	93.11
init_params	2.00	263.00	-	131.50	2.00	3,417.00	-	1,708.50	10.00	14,567,971.00	9.21	1,456,797.13
zero_vec	10.00	81.00	-	8.10	10.00	73.00	-	7.30	40.00	620.00	-	15.50
CPU/MKL												
im2col	5,466.00	3,049,667.00	18.48	557.93	5,466.00	4,749,413.00	12.28	868.90				
maxpooling	5,466.00	4,869,814.00	29.50	890.93	5,466.00	4,252,335.00	10.99	777.96				
demaxpooling	4,686.00	279,471.00	1.69	59.64	4,686.00	3,937,187.00	10.18	840.20				
matmul	24,990.00	1,139,077.00	6.90	45.58	24,990.00	8,429,056.00	21.79	337.30	143,694.00	263,837,117.00	45.18	1,836.10
matadd_col	10,932.00	324,063.00	1.96	29.64	10,932.00	4,458,462.00	11.52	407.84	54,660.00	905,231.00	0.16	16.56
matmul_elwise	4,686.00	91,542.00	0.55	19.54	4,686.00	726,658.00	1.88	155.07	42,174.00	2,156,886.00	0.37	51.14
vect_scalar_prod	37,488.00	20,817.00	0.13	0.56	37,488.00	239,046.00	0.62	6.38	187,440.00	70,102,834.00	12.00	374.00
matsub	23,429.00	82,178.00	0.50	3.51	23,429.00	205,381.00	0.53	8.77	98,405.00	49,363,705.00	8.45	501.64
matadd	18,744.00	70,396.00	0.43	3.76	18,744.00	335,671.00	0.87	17.91	93,720.00	49,927,001.00	8.55	532.73
mat_reduce_rows	9,372.00	1,899,966.00	11.51	202.73	9,372.00	1,003,812.00	2.59	107.11	46,860.00	2,831,094.00	0.48	60.42
vec_copy	649,600.00	2,650,427.00	16.06	4.08	649,600.00	2,861,074.00	7.39	4.40	649,600.00	6,153,073.00	1.05	9.47
matrix_relu	5,466.00	406,772.00	2.46	74.42	5,466.00	3,938,392.00	10.18	720.53	49,194.00	8,196,095.00	1.40	166.61
matrix_relu_der	4,686.00	233,967.00	1.42	49.93	4,686.00	2,015,324.00	5.21	430.07	42,174.00	7,559,920.00	1.29	179.26
matrix_softmax	5,466.00	380,352.00	2.30	69.59	5,466.00	404,612.00	1.05	74.02	5,466.00	38,037,052.00	6.51	6,958.85
xentropy	10,930.00	982,911.00	5.95	89.93	10,930.00	1,095,782.00	2.83	100.25	10,930.00	79,827,534.00	13.67	7,303.53
mat_copy	4,686.00	24,973.00	0.15	5.33	4,686.00	33,474.00	0.09	7.14	4,686.00	619,890.00	0.11	132.29
init_params	2.00	449.00	-	224.50	2.00	4,424.00	0.01	2,212.00	10.00	4,167,939.00	0.71	416,793.91
zero_vec	10.00	18.00	-	1.80	10.00	294.00	-	29.40	40.00	283,156.00	0.05	7,078.90

Figura A.2: Estudio del problema encontrado en el tiempo de ejecución del entrenamiento de topologías convolucionales. Comparados MKL y CLBLAS.



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

Timings

Forward	Backprop	Gradients	Update
7215 us	0 us	390 us	1519 us
23 us	40 us	0 us	0 us
3562 us	2084 us	209 us	353 us
93 us	40 us	0 us	0 us
250 us	950 us	62 us	232 us
45 us	1262 us	0 us	0 us
11188 us	4376 us	661 us	2104 us

Tot: 18329 us, ETF: 00:01:25, 37.795 GFLOPS

Figura A.3: Topología MLP_Big

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

Timings

layer	Forward	Backprop	Gradients	Update
0				
1	7281 us	0 us	8454 us	1558 us
2	24 us	120 us	0 us	0 us
3	475 us	523 us	2208 us	649 us
4	95 us	39 us	0 us	1 us
5	494 us	375 us	277 us	651 us
6	21 us	37 us	0 us	1 us
7	538 us	301 us	274 us	636 us
8	20 us	112 us	0 us	0 us
9	471 us	302 us	273 us	697 us
10	22 us	128 us	0 us	0 us
11	481 us	301 us	274 us	656 us
12	100 us	39 us	0 us	0 us
13	472 us	376 us	276 us	646 us
14	21 us	39 us	0 us	0 us
15	554 us	301 us	278 us	640 us
16	21 us	117 us	0 us	0 us
17	472 us	302 us	271 us	654 us
18	22 us	118 us	0 us	0 us
19	476 us	304 us	275 us	664 us
20	867 us	1230 us	0 us	0 us
TOTAL	12927 us	5064 us	12860 us	7453 us

Tot: 38304 us, ETF: 00:02:59, 378.650 GFLOPS

Figura A.4: Topología MLP_Large

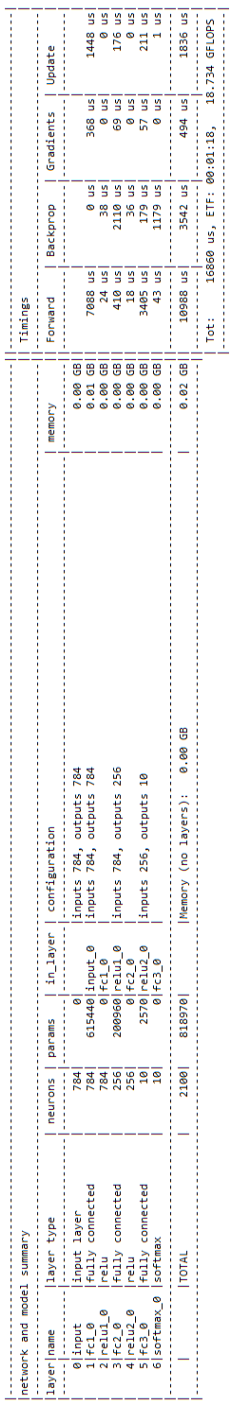


Figura A.5: Topología MLP_Medium

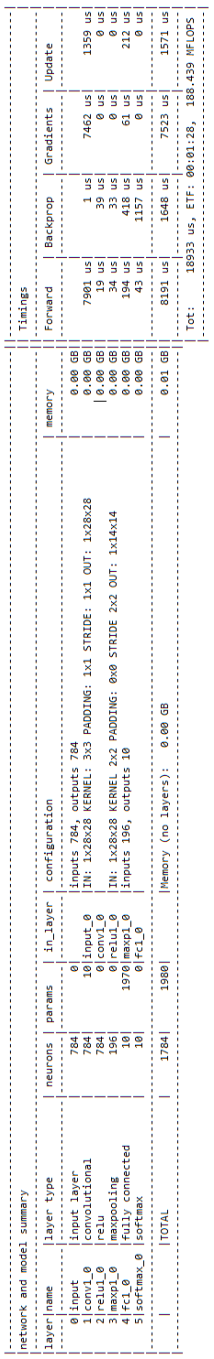


Figura A.6: Topología MNIST_CONV

Figura A.7: Topología MNIST_CONV16channels

```
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 |

Timings
| Forward | Backprop | Gradients | Update |
|---|---|---|---|
| 7920 us | 1 us | 13572 us | 1305 us |
| 154 us | 296 us | 0 us | 0 us |
| 3403 us | 2871 us | 9991 us | 195 us |
| 144 us | 283 us | 0 us | 0 us |
| 213 us | 142 us | 0 us | 0 us |
| 1111 us | 360 us | 443 us | 366 us |
| 18 us | 35 us | 0 us | 0 us |
| 3557 us | 186 us | 59 us | 221 us |
| 45 us | 1197 us | 0 us | 0 us |
| 16565 us | 5371 us | 24065 us | 2087 us |

Tot: 48088 us, ETF: 00:03:07, 91.027 GFLOPS
```

Figura A.8: Topología VGG1

layer	name	layer type	neurons	params	in_layer	configuration	memory	Forward	Backprop	Gradients	Update
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	7833 us	0 us	13557 us	1292 us
2	relu1_0	relu	32768	0	conv1_0		0.02 GB	153 us	284 us	0 us	0 us
3	conv2_0	convolutional	32768	9248	relu1_0	IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32	0.16 GB	3369 us	2872 us	9960 us	176 us
4	relu2_0	relu	32768	0	conv2_0		0.02 GB	144 us	274 us	0 us	0 us
5	maxp1_0	maxpooling	8192	0	relu2_0	IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16	0.01 GB	203 us	138 us	0 us	0 us
6	conv3_0	convolutional	16384	18496	maxp1_0	IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16	0.05 GB	2813 us	741 us	2488 us	174 us
7	relu3_0	relu	16384	0	conv3_0		0.01 GB	115 us	236 us	0 us	0 us
8	conv4_0	convolutional	16384	36928	relu3_0	IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16	0.08 GB	1279 us	1535 us	2698 us	169 us
9	relu4_0	relu	16384	0	conv4_0		0.01 GB	107 us	221 us	0 us	0 us
10	maxp2_0	maxpooling	4096	0	relu4_0	IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8	0.00 GB	122 us	51 us	0 us	0 us
11	fc1_0	fully connected	128	524416	maxp2_0	inputs 4096, outputs 128	0.01 GB	769 us	62 us	434 us	277 us
12	relu5_0	relu	128	0	fc1_0		0.00 GB	18 us	35 us	1 us	0 us
13	fc2_0	fully connected	10	1290	relu5_0	inputs 128, outputs 10	0.00 GB	3444 us	184 us	60 us	205 us
14	softmax_0	softmax	10	0	fc2_0		0.00 GB	44 us	1205 us	0 us	0 us
		TOTAL	209172	591274		Memory (no layers): 0.00 GB	0.43 GB	20413 us	7838 us	29198 us	2293 us

Table groups: network and model summary (layer – memory); Timings (Forward – Update).

Tot: 59742 us, ETF: 00:03:53, 160.943 GFLOPS

Figura A.9: Topología VGG2

```
|---------------------------------------------------------------------------------------------------------------------------------------------------||-------------------------------------------------------|
|network and model summary                                                                                                                          || Timings                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------||-------------------------------------------------------|
|layer|name       |layer type          | neurons | params  | in_layer | configuration                                               | memory     || Forward    | Backprop   | Gradients  | Update     |
|-----|-----------|--------------------|---------|---------|----------|-------------------------------------------------------------|------------||------------|------------|------------|------------|
|    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||     7965 us|        1 us|    13586 us|     1322 us|
|    2|relu1_0    |relu                |    32768|        0|conv1_0   |                                                             |     0.02 GB||      155 us|      303 us|        1 us|        0 us|
|    3|conv2_0    |convolutional       |    32768|     9248|relu1_0   |IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32 |     0.16 GB||     3397 us|     2850 us|     9964 us|      189 us|
|    4|relu2_0    |relu                |    32768|        0|conv2_0   |                                                             |     0.02 GB||      143 us|      279 us|        0 us|        0 us|
|    5|maxp1_0    |maxpooling          |     8192|        0|relu2_0   |IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16 |     0.01 GB||      207 us|      143 us|        0 us|        0 us|
|    6|conv4_0    |convolutional       |    16384|    18496|maxp1_0   |IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16 |     0.05 GB||     1070 us|      747 us|     2488 us|      174 us|
|    7|relu3_0    |relu                |    16384|        0|conv4_0   |                                                             |     0.01 GB||      109 us|      222 us|        0 us|        0 us|
|    8|conv4_0    |convolutional       |    16384|    36928|relu3_0   |IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16 |     0.08 GB||     1271 us|     1547 us|     2700 us|      177 us|
|    9|relu4_0    |relu                |    16384|        0|conv4_0   |                                                             |     0.01 GB||      109 us|      222 us|        0 us|        0 us|
|   10|maxp2_0    |maxpooling          |     4096|        0|relu4_0   |IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8   |     0.00 GB||      141 us|       55 us|        0 us|        0 us|
|   11|conv5_0    |convolutional       |     8192|    73856|maxp2_0   |IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8  |     0.02 GB||      604 us|      532 us|     1271 us|      173 us|
|   12|relu5_0    |relu                |     8192|        0|conv5_0   |                                                             |     0.01 GB||       99 us|      190 us|        0 us|        0 us|
|   13|conv6_0    |convolutional       |     8192|   147584|relu5_0   |IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8 |     0.04 GB||      978 us|      838 us|     1941 us|      282 us|
|   14|relu6_0    |relu                |     8192|        0|conv6_0   |                                                             |     0.01 GB||      104 us|      196 us|        0 us|        1 us|
|   15|maxp3_0    |maxpooling          |     2048|        0|relu6_0   |IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4   |     0.00 GB||       43 us|      111 us|        0 us|        0 us|
|   16|fc1_0      |fully connected     |      128|   262272|maxp3_0   |inputs 2048, outputs 128                                     |     0.00 GB||      524 us|     1715 us|      424 us|      195 us|
|   17|relu7_0    |relu                |      128|        0|fc1_0     |                                                             |     0.00 GB||       21 us|       38 us|        0 us|        0 us|
|   18|fc2_0      |fully connected     |       10|     1290|relu7_0   |inputs 128, outputs 10                                       |     0.00 GB||     3632 us|      185 us|       76 us|      217 us|
|   19|softmax_0  |softmax             |       10|        0|fc2_0     |                                                             |     0.00 GB||       47 us|     1173 us|        0 us|        0 us|
|---------------------------------------------------------------------------------------------------------------------------------------------------||------------|------------|------------|------------|
|     |           |TOTAL               |   243988|   550570|          |Memory (no layers):   0.00 GB                                |     0.50 GB||    20619 us|    11347 us|    32451 us|     2730 us|
|---------------------------------------------------------------------------------------------------------------------------------------------------||------------|------------|------------|------------|
                                                                                                                                                     | Tot:     67147 us, ETF: 00:04:22,  222.679 GFLOPS |
                                                                                                                                                     |-------------------------------------------------------|
```

```
-------------------------|
                         |
-------------------------|
Gradients   | Update     |
------------|------------|
            |            |
   13572 us|     1305 us|
       0 us|        0 us|
    9991 us|      195 us|
       0 us|        0 us|
       0 us|        0 us|
     443 us|      366 us|
       0 us|        0 us|
      59 us|      221 us|
       0 us|        0 us|
------------|------------|
   24065 us|     2087 us|
------------|------------|
:03:07,   91.027 GFLOPS |
-------------------------|
```

Figura A.10: Topología VGG3

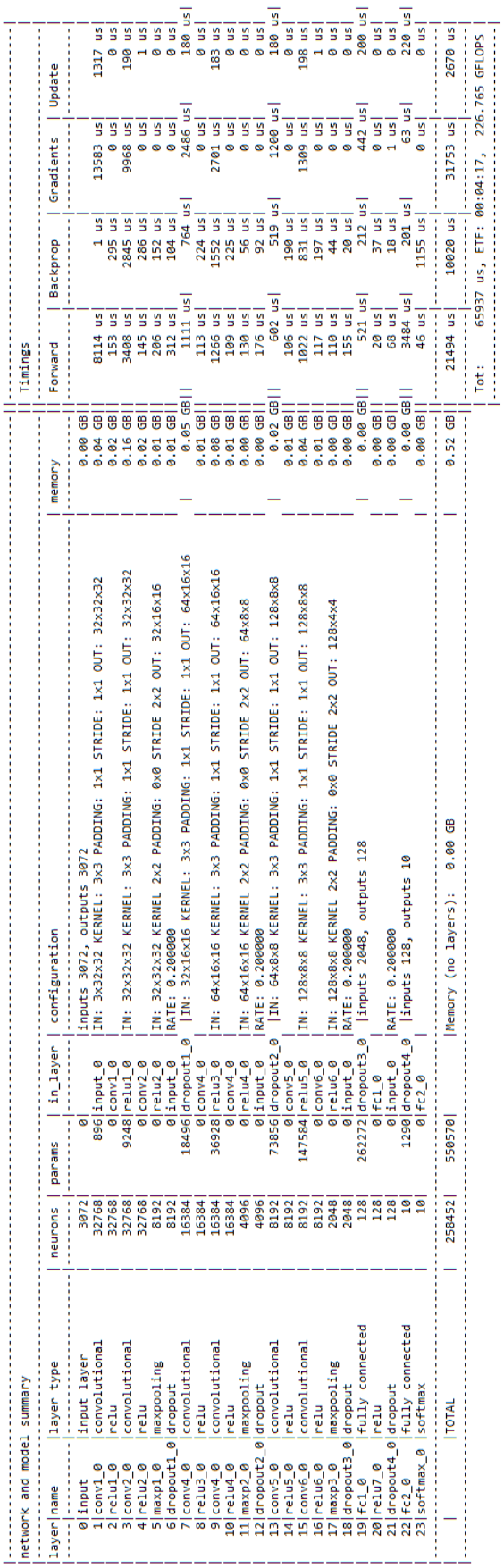


Figura A.11: Topología VGG3 Dropout

```
Network and model summary
| Layer | Name | Layer type | Neurons | Params | In_layer | Configuration | Memory | Timings: Forward | Backprop | Gradients | Update |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | 8029 us | 1 us | 13585 us | 1301 us |
| 2 | relu1_0 | relu | 32768 | 0 | conv1_0 | | 0.02 GB | 152 us | 322 us | 0 us | 0 us |
| 3 | conv2_0 | convolutional | 32768 | 9248 | relu1_0 | IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32 | 0.16 GB | 3444 us | 2895 us | 9962 us | 187 us |
| 4 | relu2_0 | relu | 32768 | 0 | conv2_0 | | 0.02 GB | 143 us | 278 us | 0 us | 0 us |
| 5 | maxpl_0 | maxpooling | 8192 | 0 | relu2_0 | IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16 | 0.01 GB | 208 us | 143 us | 0 us | 0 us |
| 6 | dropout1_0 | dropout | 8192 | 0 | input_0 | RATE: 0.200000 | 0.01 GB | 298 us | 96 us | 0 us | 0 us |
| 7 | conv4_0 | convolutional | 16384 | 18496 | dropout1_0 | IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16 | 0.05 GB | 1078 us | 748 us | 2486 us | 181 us |
| 8 | relu3_0 | relu | 16384 | 0 | conv4_0 | | 0.01 GB | 126 us | 236 us | 0 us | 0 us |
| 9 | conv4_0 | convolutional | 16384 | 36928 | relu3_0 | IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16 | 0.08 GB | 1315 us | 1575 us | 2702 us | 177 us |
| 10 | relu4_0 | relu | 16384 | 0 | conv4_0 | | 0.01 GB | 111 us | 221 us | 0 us | 1 us |
| 11 | maxp2_0 | maxpooling | 4096 | 0 | relu4_0 | IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8 | 0.00 GB | 128 us | 54 us | 0 us | 0 us |
| 12 | dropout2_0 | dropout | 4096 | 0 | input_0 | RATE: 0.300000 | 0.00 GB | 175 us | 92 us | 0 us | 0 us |
| 13 | conv5_0 | convolutional | 8192 | 73856 | dropout2_0 | IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8 | 0.02 GB | 597 us | 516 us | 1199 us | 172 us |
| 14 | relu5_0 | relu | 8192 | 0 | conv5_0 | | 0.01 GB | 100 us | 195 us | 0 us | 0 us |
| 15 | conv6_0 | convolutional | 8192 | 147584 | relu5_0 | IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8 | 0.04 GB | 1017 us | 846 us | 1305 us | 185 us |
| 16 | relu6_0 | relu | 8192 | 0 | conv6_0 | | 0.01 GB | 115 us | 137 us | 0 us | 0 us |
| 17 | maxp3_0 | maxpooling | 2048 | 0 | relu6_0 | IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4 | 0.00 GB | 109 us | 44 us | 0 us | 0 us |
| 18 | dropout3_0 | dropout | 2048 | 0 | input_0 | RATE: 0.400000 | 0.00 GB | 173 us | 21 us | 0 us | 0 us |
| 19 | fc1_0 | fully connected | 128 | 262272 | dropout3_0 | inputs 2048, outputs 128 | 0.00 GB | 523 us | 212 us | 433 us | 187 us |
| 20 | relu7_0 | relu | 128 | 0 | fc1_0 | | 0.00 GB | 21 us | 37 us | 0 us | 0 us |
| 21 | dropout4_0 | dropout | 128 | 0 | input_0 | RATE: 0.500000 | 0.00 GB | 63 us | 17 us | 0 us | 0 us |
| 22 | fc2_0 | fully connected | 10 | 1290 | dropout4_0 | inputs 128, outputs 10 | 0.00 GB | 3510 us | 202 us | 64 us | 216 us |
| 23 | softmax_0 | softmax | 10 | 0 | fc2_0 | | 0.00 GB | 47 us | 1165 us | 0 us | 0 us |
| | | TOTAL | 258452 | 550570 | | Memory (no layers): 0.00 GB | 0.52 GB | 21482 us | 10113 us | 31736 us | 2607 us |

Tot: 65938 us, ETF: 00:04:17, 226.762 GFLOPS
```

Figura A.12: Topología VGG3 Dropout 20-30-40-50

network and model summary / Timings

layer	name	layer type	neurons	params	in_layer	configuration	memory	Forward	Backprop	Gradients	Update
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	7909 us	0 us	13604 us	1288 us
2	relu1_0	relu	32768	0	conv1_0		0.02 GB	149 us	297 us	0 us	0 us
3	bn1_0	batch normalization	32768	64	relu1_0		0.10 GB	12115 us	1239894 us	4734 us	187 us
4	conv2_0	convolutional	32768	9248	bn1_0	IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32	0.16 GB	3373 us	2791 us	9977 us	187 us
5	relu2_0	relu	32768	0	conv2_0		0.02 GB	148 us	282 us	0 us	0 us
6	bn2_0	batch normalization	32768	64	relu2_0		0.10 GB	6141 us	8531 us	4681 us	178 us
7	maxp1_0	maxpooling	8192	0	bn2_0	IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16	0.01 GB	225 us	143 us	0 us	0 us
8	dropout1_0	dropout	8192	0	input_0	RATE: 0.200000	0.01 GB	307 us	100 us	0 us	0 us
9	conv4_0	convolutional	16384	18496	dropout1_0	IN: 32x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16	0.05 GB	1057 us	757 us	2555 us	177 us
10	relu3_0	relu	16384	0	conv4_0		0.01 GB	126 us	221 us	0 us	0 us
11	bn3_0	batch normalization	16384	128	relu3_0		0.05 GB	2263 us	7336 us	5226 us	173 us
12	conv4_0	convolutional	16384	36928	bn3_0	IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x16x16	0.08 GB	1217 us	1524 us	2704 us	179 us
13	relu4_0	relu	16384	0	conv4_0		0.01 GB	110 us	224 us	0 us	0 us
14	bn4_0	batch normalization	16384	128	relu4_0		0.05 GB	2236 us	7394 us	5235 us	177 us
15	maxp2_0	maxpooling	4096	0	bn4_0	IN: 64x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x8x8	0.00 GB	138 us	56 us	0 us	0 us
16	dropout2_0	dropout	4096	0	input_0	RATE: 0.300000	0.00 GB	181 us	97 us	0 us	0 us
17	conv5_0	convolutional	8192	73856	dropout2_0	IN: 64x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8	0.02 GB	621 us	526 us	1203 us	173 us
18	relu5_0	relu	8192	0	conv5_0		0.01 GB	105 us	200 us	1 us	0 us
19	bn5_0	batch normalization	8192	256	relu5_0		0.03 GB	1210 us	13139 us	9364 us	169 us
20	conv6_0	convolutional	8192	147584	bn5_0	IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x8x8	0.04 GB	1057 us	849 us	1248 us	192 us
21	relu6_0	relu	8192	0	conv6_0		0.01 GB	122 us	208 us	0 us	0 us
22	bn6_0	batch normalization	8192	256	relu6_0		0.03 GB	1213 us	10879 us	20901 us	313 us
23	maxp3_0	maxpooling	2048	0	bn6_0	IN: 128x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x4x4	0.00 GB	261 us	64 us	0 us	1 us
24	dropout3_0	dropout	2048	0	input_0	RATE: 0.400000	0.00 GB	295 us	33 us	0 us	1 us
25	fc1_0	fully connected	128	262272	dropout3_0	inputs 2048, outputs 128	0.00 GB	821 us	425 us	795 us	338 us
26	relu7_0	relu	128	0	fc1_0		0.00 GB	34 us	63 us	0 us	0 us
27	bn1_0	batch normalization	128	256	relu7_0		0.00 GB	643 us	258906 us	9397 us	171 us
28	dropout4_0	dropout	128	0	input_0	RATE: 0.500000	0.00 GB	254 us	21 us	0 us	0 us
29	fc2_0	fully connected	10	1290	dropout4_0	inputs 128, outputs 10	0.00 GB	3506 us	196 us	69 us	218 us
30	softmax_0	softmax	10	0	fc2_0		0.00 GB	42 us	47 us	0 us	0 us
		TOTAL	373268	551722		Memory (no layers): 0.00 GB	0.88 GB	47879 us	1555203 us	91694 us	4122 us

Tot: 1698898 us, ETF: 01:50:34, 8.801 GFLOPS

Figura A.13: Topología VGG3 Dropout 20-30-40-50 BN