UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Soporte de Comunicación Eficiente en Plataforma de Entrenamiento Distribuido de Redes Neuronales

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Miguel Ángel De Moya Jiménez

Tutores: José Flich Cardo
Pedro Juan López Rodríguez

Curso 2020-2021

Resum

El món de la intel·ligència artificial progressa a una gran velocitat, proporcionant-nos aplicacions cada vegada millors i més eficients que ens ajuden a millorar la nostra qualitat de vida i augmentar la nostra productivitat. Un exemple d'això són les xarxes neuronals artificials, que en els últims anys han revolucionat els paradigmes d'automatització i anàlisi de dades.

Malgrat això, aquestes xarxes han de passar per un procés anomenat entrenament abans de fer qualsevol tasca. Aquest procés pot arribar a requerir de grans capacitats de còmput per a completar-se. A causa d'això, per a poder realitzar aquests càlculs en una quantitat de temps assumible, es necessita de sistemes que permeten realitzar aquests càlculs eficientment.

Una alternativa per a realitzar aquestes execucions en una quantitat de temps raonable és l'ús de sistemes distribuïts. Aquests permeten dividir les operacions de còmput necessàries entre diversos components i per tant, realitzar càlculs de manera paral·lela. Aquesta classe de sistemes necessiten d'un ús eficient de les comunicacions perquè el pes d'aquestes no llastre el funcionament general del sistema.

Per aquesta raó s'ha fet aquest treball final de grau, amb l'objectiu de realitzar un estudi de com optimitzar les comunicacions en entrenaments de xarxes neuronals en sistemes distribuïts.

En aquest treball s'utilitza l'eina d'entrenament i inferència de xarxes neuronals *HELENNA podent comprovar mitjançant aquesta l'impacte real de les comunicacions en un entrenament distribuït de xarxes neuronals.

Aquest estudi s'enfoca principalment en la comunicació entre GPU, ja que permet realitzar aquests entrenaments de forma més eficient en comparació amb una CPU.

En concret s'analitza l'impacte de diferents factors que ens permeten optimitzar de manera eficient l'entrenament distribuït de xarxes neuronals. Analitzem diferents tecnologies i components del programari de comunicacions d'altes prestacions, aprofundint en aquelles combinacions que ens ofereixen uns temps de comunicació més adequats per a un entrenament distribuït.

Paraules clau: GPU, CUDA, MPI, NCCL,entrenament síncron, entrenament asíncron, sistemes distribuïts, sistemes heterogeneos, HELENNA

Resumen

El mundo de la inteligencia artificial progresa a una gran velocidad, proporcionándonos cada vez mejores y más eficientes aplicaciones que nos ayudan a mejorar nuestra calidad de vida y aumentar nuestra productividad. Un ejemplo de esto son las redes neuronales artificiales, que en los últimos años han revolucionado los paradigmas de automatización y análisis de datos.

Pese a esto, estas redes deben pasar por un proceso llamado entrenamiento antes de realizar cualquier tarea. Este proceso puede llegar a requerir de grandes capacidades de cálculo para completarse. Debido a esto, para poder realizar estos cálculos en una cantidad de tiempo asumible, se necesita de sistemas que permitan realizar estos cálculos eficientemente.

Una alternativa para realizar estas ejecuciones en una cantidad de tiempo razonable es el uso de sistemas distribuidos. Estos permiten dividir las operaciones de cálculo necesarias entre varios componentes y por tanto, realizar cálculos de forma paralela. Esta

clase de sistemas necesitan de un uso eficiente de las comunicaciones para que el peso de estas no lastre el funcionamiento general del sistema.

Por esta razón se ha realizado este trabajo final de grado, con el objetivo de realizar un estudio de cómo optimizar las comunicaciones en entrenamientos de redes neuronales en sistemas distribuidos.

En este trabajo se utiliza la herramienta de entrenamiento e inferencia de redes neuronales HELENNA, pudiendo comprobar mediante ésta el impacto real de las comunicaciones en un entrenamiento distribuido de redes neuronales. Este estudio se enfoca principalmente en la comunicación entre GPU, ya que permite realizar estos entrenamientos de forma más eficiente en comparación con una CPU.

En concreto se analiza el impacto de distintos factores que nos permitan optimizar de forma eficiente el entrenamiento distribuido de redes neuronales. Analizamos diferentes tecnologías y componentes software de comunicaciones de altas prestaciones, profundizando en aquellas combinaciones que nos ofrecen unos tiempos de comunicación más adecuados para un entrenamiento distribuido.

Palabras clave: GPU, CUDA, MPI, NCCL,entrenamiento síncrono, entrenamiento asíncrono, sistemas distribuidos, sistemas heterogéneos, HELENNA

Abstract

The world of artificial intelligence is progressing at a great speed, providing us with better and more efficient applications that help us improve our quality of life and increase our productivity. An example of this is artificial neural networks, which in recent years have revolutionized the automation and data analysis paradigms.

Despite this, these networks must go through a process called training before performing any task. This process may require large computing capacities to complete. Due to this, in order to perform these calculations in an acceptable amount of time, systems are needed that allow these calculations to be carried out efficiently.

An alternative to perform these executions in a reasonable amount of time is the use of distributed systems. These allow the necessary computational operations to be divided among several components and therefore, to perform calculations in parallel. This class of systems require an efficient use of communications so that the weight of these does not weigh down the general operation of the system.

For this reason, this final degree project has been carried out, with the aim of carrying out a study on how to optimize communications in neural network training in distributed systems.

In this work, the HELENNA neural network training and inference tool is used, being able to verify by means of this the real impact of communications in a distributed training of neural networks.

This study mainly focuses on the between GPUs, since it allows to perform communicationthese trainings more efficiently compared to a CPU.

Specifically, the impact of different factors that allow us to efficiently optimize the distributed training of neural networks is analyzed. We analyze different technologies and high-performance communications software components, delving into those combinations that offer us more adequate communication times for distributed training.

Key words: GPU, CUDA, MPI, NCCL, synchronous training, asynchronous training, distributed systems, heterogeneous systems, HELENNA

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En este primer capítulo realizaremos una introducción a los diversos aspectos sobre los que tratará el trabajo de fin de grado, abordándose la motivación que nos ha llevado a realizar este trabajo, los objetivos que tratamos de conseguir, la estructura de esta memoria, un contexto sobre cómo se ha llevado a cabo el desarrollo de la investigación y una mención de los Objetivos de Desarrollo Sostenible (ODS) que abordamos.

1.1 Contexto y motivación

El concepto de una Inteligencia Artificial (IA) que permita a máquinas comportarse de una forma que pueda ser llamadas inteligente lleva definido desde los años 50 [54] [52], fechas en las que personajes como John McCarthy con su definición del concepto de IA en 1956, Alan Turing (Figura 1.1) con la creación del famoso Test de Turing para determinar la inteligencia de una máquina en 1950 [53] o Frank Rosenblat con el primer diseño de una red neuronal artificial en 1957, establecieron las bases de lo que la inteligencia artificial es hoy en día.

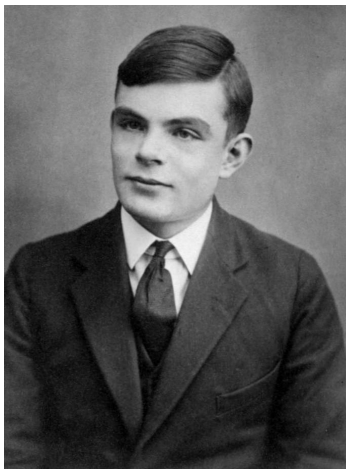


Figura 1.1: Fotografía de Alan Turing, matemático creador del Test de Turing y considerado el precursor de la Inteligencia Artificial.

Pese a esta temprana concepción de la IA y a sucesivos avances en este campo en años posteriores, su evolución sufrió un estancamiento, ya que pese a ser una tecnología prometedora no podía hacerse uso de esta para resolver problemas complejos, por lo que fue perdiendo popularidad. Esto comenzó a cambiar a partir de los años 90, en los que muchas de las metas históricas de la IA fueron alcanzadas, consiguiendo así que su po-

pularidad volviera a subir. Algunos ejemplos de los avances conseguidos durante estos años, incluye la famosa victoria por parte de la IA *Deep Blue* contra el campeón mundial de ajedrez Garri Gasparov en 1997 o en la implantación de un software de reconocimiento de voz en Windows ese mismo año [35].

Esta serie de avances permitieron reactivar un interés en la IA que se había perdido durante la década de los 70, ocasionando un incremento de las investigaciones en este ámbito en los años 2000 y desembocando por fin en un desarrollo exponencial de la IA durante esta última década.

Esto ha propiciado que la IA se coloque como uno de los puntos clave en el desarrollo económico y tecnológico de muchos sectores en los próximos años, gracias a sus múltiples aplicaciones en múltiples campos. Un ejemplo del estancamiento que llego a sufrir esta tecnología y del *boom* de estos últimos años, está en que el Test de Turing pese a haber sido concebido en los años 50, no fue superado por una IA hasta el año 2014 [6].

Algunos ejemplos de las posibles aplicaciones de la Inteligencia Artificial están en el reconocimiento de imagen, automatización de procesos o ayuda en la toma de decisiones, además de aplicaciones más especificas como son la conducción autónoma, la robótica o los *chatbots*.

Esta explosión durante la última década está impulsada por avances en el desarrollo tecnológico como el incremento en la capacidad de cálculo, la aparición de ciertas herramientas software como TensorFlow [31] o H20 AI [19], entornos de computación distribuida o la computación en la nube que facilita el acceso a entornos con una gran capacidad de cálculo. Otro factor que potenció su desarrollo fue el acceso a grandes cantidades de datos que pueden ser utilizados como *dataset* para realizar el "entrenamiento" de una IA o la aparición de componentes como las GPU (Graphics Procesing Unit) que permiten acelerar el cálculo de ciertas operaciones necesarias para realizar un "entrenamiento".

Gracias a este desarrollo, los problemas a resolver por las IA son cada vez más y más complejos. Esto ocasiona que para poder resolverlos se necesite de algoritmos más complejos. Uno de los algoritmos con mayor popularidad actualmente y que nos permite resolver muchos de estos problemas y acercarnos a una reproducción del aprendizaje humano es el *Deep Learning*.

Este algoritmo está compuesto por las llamadas redes neuronales, basadas (de forma muy ligera) en el comportamiento del cerebro humano. Estos algoritmos no requieren de reglas programadas anteriormente, sino que permiten a un sistema "aprender" como realizar una tarea determinada gracias a una fase de entrenamiento en la que podrá reconocer patrones presentes en los datos de entrada. Esta fase de entrenamiento es indispensable en los algoritmos de *Deep Learning*. El problema que presenta es que las redes neuronales que conforman el algoritmo pueden llegar a ser muy complejas, lo que implica que sea necesaria una gran capacidad de cálculo para llevar a cabo estos entrenamientos en una cantidad de tiempo asumible.

Por ello, para realizar estos entrenamientos de una forma que sea eficiente desde el punto de vista de su tiempo de ejecución, deben usarse sistemas que presenten una serie de características, como pueden ser el uso de CPU (Central Processing Unit) lo suficientemente rápidas, el uso de GPU que permitan acelerar ciertas operaciones de cálculo, el uso de múltiples CPU con memoria compartida, el uso de sistemas heterogéneos que permitan el uso conjunto de sistemas en los que se encuentren nodos con CPU y/o GPU o el uso de sistemas distribuidos que permitan realizar el entrenamiento paralelamente en varios nodos.

La razón por la que una GPU es mejor a una CPU a la hora realizar el entrenamiento de una red neuronal está en la naturaleza de las operaciones necesarias para realizar dicho entrenamiento. Una red neuronal está compuesta por distintas capas con entradas y pesos, para realizar el entrenamiento se toman las entradas y se procesan en las distintas capas, cada capa tendrá unas entradas con unos "pesos" que se van ajustando durante el entrenamiento. Para realizar esto se realiza un proceso llamado regresión lineal, el cual consiste en multiplicaciones de matrices cuyo tamaño depende del número de neuronas por capa, del número de neuronas en la capa previa y del número de entradas.

Estas matrices tienen un tamaño muy elevado, ocasionando que el tiempo utilizado por una CPU para realizar esta tarea pueda llegar a ser inasumible, ya que debe realizar una cantidad muy alta de multiplicaciones, mientras que una GPU está dedicada precisamente a realizar operaciones de coma flotante disponiendo de múltiples núcleos ALU(*Arithmetic Logic Units*) (Figura 1.2) que pueden trabajar en paralelo a la hora de realizar estas multiplicaciones en un tiempo mucho menor.

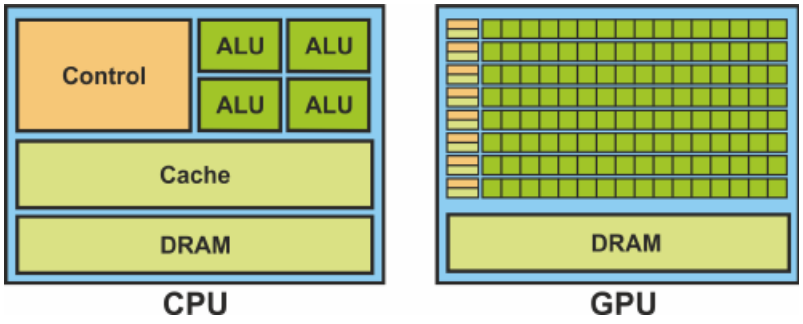


Figura 1.2: Comparación de las arquitecturas de una CPU vs GPU

Respecto a los sistemas distribuidos, estos permiten resolver problemas de computación masivos al usar un elevado número de unidades de cálculo organizados en *clusters* que permiten que este conjunto de computadores se comporten como si fueran uno solo. Esta clase de sistemas permiten distribuir una misma información a través de distintos computadores, donde cada uno de estos computadores realice una serie de cálculos de forma independiente, enviando los resultados a una unidad central o a distintos computadores del sistema dependiendo de como esté organizado. Por tanto, se realizan tareas de forma paralela pudiendo así disminuir el tiempo necesario a la hora de efectuar los cálculos.

Esta clase de sistemas facilitan el entrenamiento de redes neuronales, ya que permiten dividir la elevada cantidad de cálculos necesarios entre los computadores que conformen el sistema, pudiendo realizar así los cálculos en un tiempo menor. Para poder llevar a cabo correctamente estos entrenamientos distribuidos, los computadores que conformen el sistema deben sincronizarse para obtener los parámetros actualizados una vez realizados los cálculos correspondientes.

En principio puede parecer que mediante un entrenamiento distribuido formado por varios nodos siempre se obtendrán mejores resultados a los obtenidos en un sistema con una sola unidad de cálculo. Sin embargo, esto no siempre es así. A la hora de sincronizar las distintas unidades del sistema debe llevarse a cabo una comunicación entre estas. El peso de estas comunicaciones puede lastrar las prestaciones obtenidas al realizar el entrenamiento causando que no se aproveche en su totalidad el paralelismo de los sistemas distribuidos o que incluso la versión paralela necesite una mayor cantidad de tiempo para completarse que la secuencial.

A causa de estos problemas se ha planteado el desarrollo de este trabajo. En éste se analizarán las comunicaciones en este tipo de sistemas [48] con el objetivo de encontrar alternativas que permitan aprovechar las capacidades de los sistemas distribuidos.

Para realizar este trabajo se abordarán distintas técnicas de entrenamiento distribuido que nos permiten disminuir el tiempo necesario para realizar la sincronización, permitiendo aprovechar el uso de sistemas distribuidos para el entrenamiento de redes neuronales. Para ello se abordarán tanto distintas arquitecturas para conformar un sistema distribuido como son *parameter server*, entrenamiento síncrono o entrenamiento asíncrono además de evaluar el impacto de ciertos factores en su comportamiento.

También se analizarán diferentes alternativas para realizar la comunicación entre los distintos nodos y en particular como eliminar las copias de memoria necesarias para comunicar varias GPU, buscándose como sustituir esto por una comunicación directa entre GPU sin necesidad de realizar ninguna copia de memoria, teniendo como objetivo último la mejora de las comunicaciones y la consecuente mejora en la eficiencia de un entrenamiento distribuido.

1.2 Objetivos

El objetivo principal de este trabajo es optimizar las comunicaciones de un entrenamiento distribuido de redes neuronales. Para ello, se realizará un análisis de estas comunicaciones con el objetivo de buscar como reducir la sobrecarga que causan estas y que dificulta un correcto aprovechamiento de los recursos utilizados. A la hora de reducir esta sobrecarga se buscará reducir lo máximo posible el tiempo de ejecución necesario para realizar un entrenamiento, teniendo en cuenta también factores como la precisión del modelo obtenido (*accuracy*). A lo largo de este trabajo se desarrollarán 3 objetivos que nos permitirán cumplir el objetivo principal:

- Analizar el comportamiento de las distintas técnicas disponibles de entrenamiento distribuido.
- Identificar y evaluar diferentes formas de implementar las primitivas de comunicación necesarias para el entrenamiento distribuido. En particular, aquellas que permiten comunicar distintas GPU sin tener que realizar copias de memoria en CPU. Además de como afecta el tipo de reserva de memoria utilizada en la comunicación.
- Analizar cada cuanto se debe sincronizar un entrenamiento asíncrono y como afectan los distintos parámetros a las prestaciones obtenidas.

1.3 Estructura de la memoria

Este documento está dividido en los siguientes capítulos:

- **Capítulo 1**: En este capítulo se expone la motivación para la realización de este trabajo, los objetivos a realizar, la estructura que lo conforma, el contexto en el que se ha desarrollado el trabajo y qué Objetivos de Desarrollo Sostenible se abordan en el trabajo.
- **Capítulo 2**: En este capítulo se introducirán aspectos referentes a las tecnologías de cálculo utilizadas para realizar los cálculos requeridos en un entrenamiento distribuido, tratándose que es un sistema heterogéneo e Infiniband. Además se realizará

una explicación sobre distintas implementaciones de las comunicaciones y en que consisten las distintas primitivas que pueden utilizarse para llevar a cabo una comunicación. Por último, se hará una breve descripción de los componentes que conforman el *cluster* utilizado para realizar las distintas ejecuciones necesarias en este trabajo.

- **Capítulo 3**: En este capítulo se introducirán aspectos importantes sobre los que tratará este trabajo como son la Inteligencia Artificial, las redes neuronales, el *Deep Learning*, el entrenamiento distribuido o la plataforma de entrenamiento distribuido HELENNA (HEterogeneous Neural Network Application).
- **Capítulo 4**: En este capítulo se analizarán las prestaciones de las comunicaciones en un sistema distribuido. Para ello se analizarán las prestaciones obtenidas con distintas primitivas de comunicación necesarias para sincronizar un sistema distribuido. Se estudiarán distintas bibliotecas para implementar estas primitivas. Además, se analizarán diferentes tipos de reserva de memoria en la GPU.
- **Capítulo 5**: En este capítulo se abordará cómo distintas técnicas de entrenamiento distribuido y factores relativos a estas que influyen al realizar entrenamientos en la plataforma de entrenamiento distribuido HELENNA.
- **Capítulo 6**: En este último capítulo expondremos las conclusiones obtenidas a lo largo de la realización de este trabajo además de definir que trabajo futuro pueden realizarse sobre este tema de estudio.

1.4 Contexto desarrollo del trabajo

Este trabajo se ha desarrollado en el contexto de unas prácticas de empresa realizadas en el Grupo de Arquitecturas Paralelas (GAP) de la Universidad Politécnica de Valencia (UPV). En dichas prácticas formábamos parte 6 alumnos y 4 profesores, se realizaban reuniones semanales cada jueves entre 12.30 y 14.30, en las que se abordaban tanto las tareas realizadas por cada alumno durante esa semana como los objetivos a realizar para la siguiente semana. Además, cabe decir que cada uno de los alumnos realizaba tareas relacionadas con la misma herramienta de entrenamiento de redes neuronales (HELENNA) por lo que dichas reuniones generaron un ambiente de trabajo colaborativo, proporcionando la mejora de conocimientos relativos a su propio trabajo y específicos de la herramienta.

1.5 Objetivos de Desarrollo Sostenible

Los Objetivos de Desarrollo Sostenible (ODS) son unos principios básicos definidos por la Organización de las Naciones Unidas (ONU) en 2015 con el objetivo de poner fin a la pobreza (Figura 1.3), proteger el planeta y garantizar la paz y prosperidad de las personas.

Dentro de la Agenda 2030 para el desarrollo sostenible en este trabajo se han abordado los siguientes ODS:

- **ODS 7. Garantizar el acceso a una energía asequible, segura, sostenible y moderna para todos**: El principal objetivo del desarrollo de este trabajo es la optimización de las comunicaciones con el objetivo de obtener una reducción del tiempo necesario para realizar un entrenamiento, lo que permite aumentar la sostenibilidad de los entrenamientos al reducir la cantidad de energía utilizada en cada entrenamiento.

Figura 1.3: Listado de los 17 objetivos de desarrollo sostenible de la ONU.

- **ODS 8. Promover el crecimiento económico sostenido, inclusivo y sostenible, el empleo pleno y productivo y el trabajo decente para todos**: Este trabajo está enfocado en la mejora de un apartado de los algoritmos de inteligencia artificial, una tecnología que permite automatizar procesos posibilitando un aprovechamiento de los recursos más sostenible y productivo de estos. Además, es un campo que impulsa tanto el crecimiento económico como la generación de nuevos puestos de trabajo.

CAPÍTULO 2
Tecnologías de cálculo y comunicaciones

2.1 Sistemas heterogéneos

Hoy en día los sistemas de procesamiento están compuestos por arquitecturas [43] [49] heterogéneas conformadas por distintos componentes que se dividen las tareas a realizar. Un ejemplo de esto es un sistema conformado por algunas de las distintas arquitecturas de cálculo disponibles, como pueden ser GPU, CPU o FPGA (*Field Programmable Gate Array*). El objetivo de esta clase de arquitecturas es aprovechar las capacidades de cada uno de estos sistemas para realizar el cálculo necesario de la forma más eficiente posible.

En el desarrollo de este trabajo va a abordarse una de las arquitecturas heterogéneas más típicas, la combinación de CPU y GPU [44] [12], esta nos permite obtener un mayor rendimiento respecto a arquitecturas tradicionales en las que únicamente se hace uso de la CPU.

Una CPU es el ”cerebro” de un ordenador, permite ejecutar la mayoría de instrucciones necesarias a la hora de ejecutar todo tipo de programas en un computador. La CPU permite un procesamiento muy rápido de datos en secuencia al tener pocos núcleos con una elevada frecuencia. Puede ejecutar todo tipo de tareas, ya que está optimizada para la latencia y permite cambiar entre distintas tareas muy rápidamente. Esta baja latencia puede dar una impresión de paralelismo. Sin embargo, esta unidad de procesamiento está diseñada para ejecutar únicamente una tarea a la vez.

Una GPU es un coprocesador enfocado en el procesamiento de gráficos u operaciones de coma flotante. Permite aligerar la carga de la CPU acelerando el cálculo de ciertas operaciones. Esto se debe al elevado número de núcleos con los que cuenta una GPU, que le permite realizar estos cálculos con un elevado grado de paralelismo, al contrario de la CPU que realiza las operaciones secuencialmente.

Las GPU utilizan una gran cantidad de núcleos ligeros, aprovechan el paralelismo de datos y tienen una tasa de transferencia alta. La arquitectura de manejo de flujos SIMD (Single Instruction Multiple Data) es la empleada por la mayoría de las GPU modernas. Esta arquitectura esta definida en la clasificación conocida como ”Taxonomía de Flynn” [16].

La ”Taxonomía de Flynn” es una clasificación de arquitecturas de computadores propuesta por Michael J. Flynn en 1972. Esta se basa en que cada cálculo consiste en dos flujos de datos (datos y flujos de instrucciones), estos pueden ser procesados secuencialmente (1 flujo a la vez) o en paralelo (múltiples flujos a la vez). Al poder existir un flujo de da-

tos o instrucciones en paralelo o en secuencial podemos tener 4 posibles combinaciones (Tabla 2.1).

	Una instrucción	Múltiples instrucciones
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD

Tabla 2.1: Combinaciones existentes en la Taxonomía de Flynn

La razón por la que SIMD es la principal opción utilizada en las GPU, esta relacionada con la naturaleza de las operaciones que estas realizan. Las tareas a realizar por una GPU suelen consistir únicamente en ejecutar una misma función matemática una y otra vez sobre un conjunto de datos, por lo que un flujo en el que se ejecuta una misma instrucción para múltiples conjuntos de datos es ideal para este tipo de tareas.

Además de estas combinaciones que forman la Taxonomía de Flynn, existe otro término importante respecto a las GPU como son las ejecuciones SIMT (Single Instruction Multiple Threads). Esto puede ser visto como una extensión de SIMD, añade multihilo al funcionamiento de SIMD, permitiendo esto mejorar la eficiencia de las instrucciones al optimizar la ejecución de los subprocesos.

Las diferencias que presentan ambas tecnologías (CPU y GPU) [18] [17] residen en como procesan las tareas. Una CPU puede trabajar con una gran cantidad de cálculos de distinta naturaleza. Mientras que una GPU permite aprovechar sus capacidades de cálculo en paralelo en una tarea específica.

Esto se debe a las ya citadas características de estas dos unidades de cálculo, una CPU optimizada para operaciones de cálculo secuencial permite maximizar el rendimiento en la ejecución de una única tarea por lo que el rango de tareas que se pueden ejecutar es mayor. Una GPU usa miles de núcleos más pequeños y eficientes para una arquitectura paralela que tenga como objetivo ejecutar múltiples funciones al mismo tiempo.

Vistas las características de GPU y CPU, podemos concluir que dependiendo del tipo de tareas a ejecutar será más eficiente el uso de una u otra. Al conformarse un sistema heterogéneo, compuesto por GPU y CPU, las tareas de cálculo general serán realizadas por la CPU, mientras que tareas más específicas que requieran de un elevado grado de paralelismo serán realizadas por la GPU. Así podemos aprovechar las características de ambas unidades de procesamiento, pudiendo conformar un sistema más eficiente.

Esta arquitectura nos será muy útil a la hora de realizar entrenamientos de redes neuronales, ya que podemos enmarcar la fase de entrenamiento de estas como una tarea que aprovecha las capacidades de la GPU, reduciendo significativamente el tiempo de ejecución necesario para realizar esta tarea comparado con un sistema homogéneo que únicamente hiciera uso de la CPU.

2.2 Infiniband

Infiniband [21] [38] es un estándar de red de interconexión que presenta un elevado ancho de banda y una baja latencia. Permite interconectar una gran variedad de sistemas informáticos, entre los que pueden encontrarse servidores o sistemas de almacenamiento entre otros.

El *standard* Infiniband fue creado en 1999 a partir de la fusión de los proyectos *Future I/O* desarrollado por Compaq, IBM y Hewlett-Packard y *Next Generation I/O* desarrollado por Intel, Microsoft y Sun Microsystems.

El objetivo esta unión era resolver las limitaciones (cuellos de botella, fiabilidad, escalabilidad, etc) de los buses PCI (*Peripheral Component Interconnect*) y estandarizar tecnologías emergentes en el campo de los *clusters* (Servernet, Myricom, Giganet, etc.).

Infiniband utiliza RDMA (Remote Direct Memory Access) para transferir datos directamente desde y hacia un dispositivo de almacenamiento, al espacio de usuario de otro dispositivo. Así evita la sobrecarga de una llamada al sistema, ya que RDMA permite transmitir los mensajes directamente entre ubicaciones de memoria, sin la necesidad de realizar copia de datos o de que el sistema operativo intervenga.

2.2.1. Arquitectura Infiniband

Respecto a la arquitectura de Infiniband, esta define una red de área de sistema, que permite conectar ordenadores, sistemas de E/S (entrada y salida) y dispositivos de E/S. Infiniband presenta la infraestructura necesaria para realizar la comunicación y gestión de transacciones de E/S y de comunicación entre distintos ordenadores.

Infiniband, al definir una red conmutada, permite intercambiar datos de forma simultánea a muchos dispositivos, contando así con un elevado ancho de banda y baja latencia. El ser un sistema conmutado le permite obtener ciertas características como protección, fiabilidad, escalabilidad, seguridad, etc. Estas características eran impensables en sistemas de E/S hasta la aparición de Infiniband, e incluso en gran parte de las redes que suelen utilizarse para conectar distintos computadores.

Además, un nodo final de una red de área de sistema en Infiniband puede realizar una comunicación a través de múltiples puertos del conmutador al que esté conectado. Por lo que existen distintos caminos alternativos que permiten incrementar el ancho de banda utilizable y la existencia de tolerancia a fallos.

2.2.2. Infiniband vs Ethernet

Infiniband respecto a otras tecnologías como puede ser Ethernet [15] permite obtener tanto una escalabilidad como fiabilidad mayor, así como un mayor ancho de banda y menor latencia. En la siguiente tabla 2.2 podemos observar algunas de las características que diferencian Infiniband de Ethernet.

Una de las diferencias estaría en el control de errores. Mientras que en Ethernet únicamente se usa un CRC (Cyclic Redundancy Check) de 32 bits para comprobar posibles fallos en el envío de los datos, Infiniband usa 32 bits de ICRC (Invariant Cyclic Redundancy Check) y 16 bits de VCRC (Variant Cyclic Redundancy Check). ICRC cubre solo los campos que no cambian al saltar de enrutador en enrutador. El VCRC proporciona integridad de datos a nivel de enlace entre dos saltos y el ICRC integridad de datos de un extremo a otro. En un protocolo como Ethernet en el que solo utiliza CRC un error se puede introducir dentro de un dispositivo que recalcula el CRC. La comprobación en el siguiente salto revelaría un CRC válido aunque los datos se hayan corrompido. Mientras que Infiniband incluye ICRC para que cuando se introduzca un bit de error, este se detecte siempre.

Otras diferencias están en el adaptador de *Host* utilizado o en la velocidad de datos de cada una, siendo mayor el tamaño de enlace utilizado en Infiniband, este normalmente es de 4x de ancho aunque puede ser incluso mayor. El tipo de herramientas utilizadas para realizar la captura de paquetes también es diferente. En Ethernet se usan herramientas *standard* que pueden ser utilizadas por todo tipo de dispositivos y en Infiniband herramientas específicas del vendedor. Las diferencias en el control de la congestión de uno y otra también hace que Infiniband tenga una mayor fiabilidad en sus datos, ya que mien-

tras que Ethernet descarta paquetes una vez se detecta la congestión, Infiniband envía avisos de esta congestión para que así la inyección de paquetes disminuya. Todas estas diferencias, sumadas a otras diferencias existentes entre ambas tecnologías causan que un sistema que utilice Infiniband pueda llegar a alcanzar unas mejores prestaciones que las proporcionadas por un sistema con el protocolo Ethernet.

Característica	Ethernet	Infiniband
Adaptador de *Host*	NIC (Network Interface Card)	HCA (Host Channel Adapater)
Tablas de reenvío	Control distribuido; cada *switch* descubre vecinos independientemente	Control centralizado a través del controlador de subred
Velocidad de datos	1,2,5,10,25,40,50,100 Gbps	8,16,32,54,100 Gps (normalmente con un enlace de 4x de ancho)
Control de flujo	Fotogramas de pausa o control de flujo basado en la prioridad	Control de flujo enlace a enlace basado en créditos
Control de errores	CRC de 32 bits recalculado cada *switch*	ICRC extremo a extremo de 32 bits más VCRC de 16 bits recalculado cada *switch*
Captura de paquetes	Usa las herramientas *standard* para captura de paquetes como pueden ser Wireshark [39] o tcpdump [30]	Usa un herramienta especifica del vendedor como ibdump [28] de Mellanox
Control de congestión	Descartar paquetes, esto ocasiona que se reduzca el tamaño de la ventana de envío y de retransmisiones de capas superiores	La capa de transporte envía avisos de congestión para evitar que se propague esta congestión entre los *switches*

Tabla 2.2: Comparativa entre Infiniband y Ethernet

2.3 Comunicaciones

En este apartado se introducen algunas de las primitivas de comunicación más importantes como pueden ser *Allreduce* o *Bcast*. Para esto se partirá de la implementación en MPI (Interfaz de Paso de Mensajes), ya que es el estándar en este tipo de comunicaciones. También se abordarán otras implementaciones de estas primitivas de comunicación como son MPI CUDA-aware y NCCL (Nvidia Collective Communication Library) y que permiten eliminar las copias de memoria existentes en MPI. Además, se introducirá la librería CUDA (Compute Unified Device Architecture) que al contrario que MPI permite realizar comunicaciones entre GPU y será parte de la implementación MPI CUDA-aware.

2.3.1. Primitivas MPI

MPI [42] [40] es un estándar de paso de mensajes para la programación paralela. En los programas basados en el estándar MPI, múltiples computadores se comunican a través de mensajes, permitiendo esto la sincronización de tareas realizadas de forma distribuida. Los datos pueden ser enviados a través de operaciones punto a punto en las que un proceso envía o recibe mensajes de otro o con operaciones colectivas en la que un proceso envía o recibe mensajes de varios procesos (Figura 2.1).

En primer lugar, a la hora de explicar que es cada una de estas operaciones y como se comunican los distintos procesos, hay que tener en cuenta la diferencia entre un nodo y un proceso. Un nodo puede ser concebido como cada una de las máquinas que forman nuestro sistema, estas máquinas constan de distintas CPU y GPU con una serie de nú-

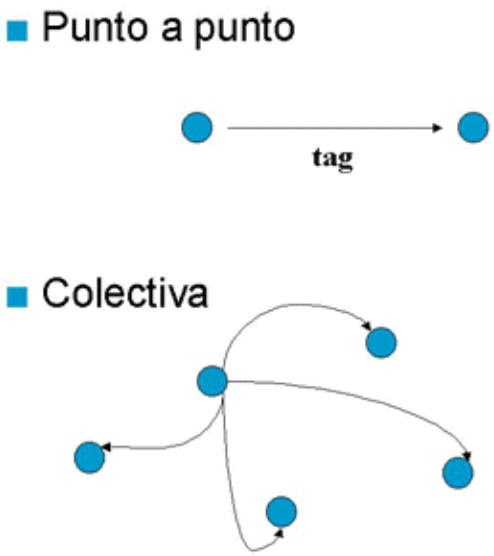


Figura 2.1: Tipos de operaciones permitidas en el estándar MPI

cleos. En una CPU cada uno de estos núcleos podrá ejecutar un proceso independiente, en nuestro caso tal y como veremos en el apartado 2.5 de esta memoria, disponemos de 28 núcleos en cada una de las CPU que conforman los nodos, por lo que un mismo nodo podría llegar a ejecutar 28 procesos. Respecto a una GPU, aunque el número de núcleos es mayor al de una CPU no podría lanzarse un proceso por cada de uno de estos núcleos al no contar estos con la potencia de los núcleos de una CPU, por lo que en este caso todos los núcleos se corresponderán con un mismo proceso.

Las principales operaciones punto a punto son *Send* y *Recv*. Para un correcto envío y recepción de una operación punto a punto tanto *Send* como *Recv* deben ser ejecutadas. Un proceso ejecutará *Send*, para así indicar que realiza el envío de un mensaje a otro proceso. Mientras que otro proceso ejecutara *Recv*, para indicar que espera a la recepción de un mensaje por parte de un proceso determinado.

Existe una gran cantidad de operaciones colectivas, como pueden ser *All-Gather*, *Gather*, *Scatter*, *All-Scatter*, *Broadcast*, *Reduce* o *All-Reduce*. Entre ellas destacaremos tres, ya que son aquellas que tendrán una mayor importancia en nuestra implementación de un entrenamiento distribuido de redes neuronales.

Broadcast se utiliza para distribuir datos desde un proceso al resto de procesos, el número de mensajes enviado será X-1, donde X se corresponde con el número de procesos ejecutándose en el sistema y 1 con el proceso emisor de los mensajes. En esta operación colectiva existirá un proceso emisor y X-1 procesos receptores. En la siguiente figura 2.2 podemos ver un ejemplo de esto. En esta, el proceso 0 envía un mensaje al resto de procesos. En este caso se ha enviado el mensaje a 7 procesos al estar ejecutándose 8 procesos.

En la operación *Reduce* se toma un *array* de datos de cada uno de los procesos que será devuelto "reducido" al proceso receptor. Por tanto existirán X procesos emisores y 1 proceso receptor. Existen distintas operaciones de reducción, alguna de ellas son MPI_MAX (devuelve el máximo elemento), MPI_MIN (devuelve el mínimo elemento), MPI_SUM (suma todos los elementos) MPI_PROD (multiplica todos los elementos), etc.

Un ejemplo de esto está en la figura 2.3. En esta, podemos observar como todos los procesos envían un mensaje al proceso 0, este mensaje se corresponde con un *array* que contiene dos elementos. En este caso la operación de reducción se corresponde con MPI_SUM por lo que el proceso 0 recibirá una suma de los *arrays* de los otros procesos.

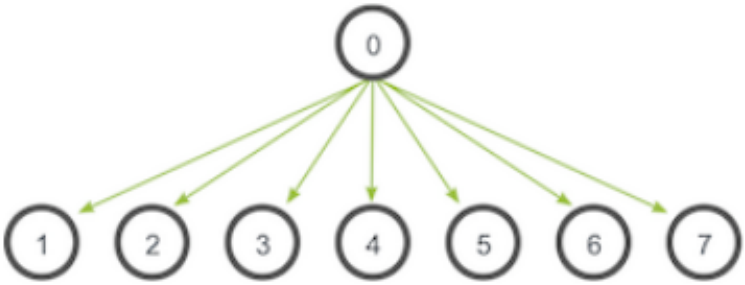


Figura 2.2: Esquema de comunicación de *Broadcast* para 8 procesos

Básicamente se realiza una suma de los valores de la primera ($5+2+7+4=18$) y de la segunda ($1+3+8+2=14$) posición del *array* que son recibidos por el proceso 0.

Por último tendríamos la operación *Allreduce*, ésta al igual que *Reduce* envía los valores de un *array* de datos por parte de todos los procesos y "reduce" este valor. Sin embargo, al contrario que en *Reduce* en el que únicamente existía un proceso receptor, en *Allreduce* el número de receptores es igual al número de emisores por lo que el resultado reducido será enviado a todos los procesos, existiendo X receptores y emisores.

Un ejemplo de esto está en la figura 2.4, en esta podemos apreciar que al igual que en el ejemplo mostrado para *Reduce* se realiza una misma reducción para cada uno de los *arrays*, con la diferencia de que todos los procesos reciben el resultado reducido, no únicamente el proceso 0.

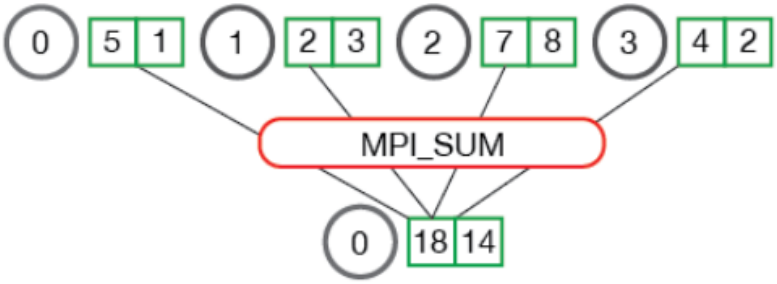


Figura 2.3: Esquema de comunicación de *Reduce* para 4 procesos para los que se realiza como operación reduce una suma

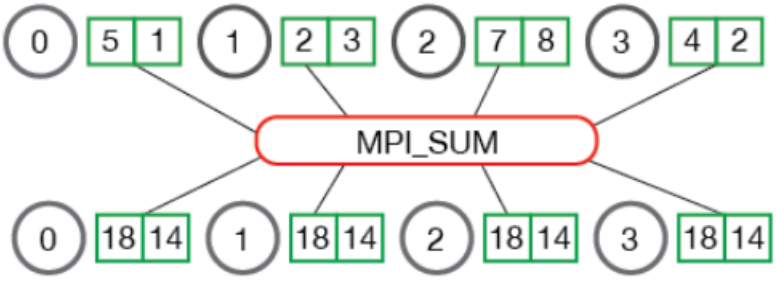


Figura 2.4: Esquema de comunicación de *Allreduce* para 4 procesos para los que se realiza como operación reduce una suma

2.3.2. CUDA

CUDA [13] es una plataforma de computación y modelo de programación que permite definir algoritmos que aprovechen las GPU de la compañía Nvidia para resolver problemas de todo tipo. Esta plataforma permite programar en lenguajes como C, C++ o Fortran incorporando a la versión de un programa que use CPU ciertas llamadas para que se haga uso de las GPU.

Existen una serie de elementos cruciales en todo programa CUDA. El primero sería lo que se conoce como *kernel*, el cual consiste en un fragmento de código al que se llama para que se ejecute como un conjunto de hilos (*threads*) de forma paralela dentro de la GPU. Este código será el ejecutado directamente por la GPU. Estos hilos se agrupan en bloques (*blocks*) que a su vez forman una malla (*grid*) (Figura 2.5). A la hora de invocar un kernel se indica el número de hilos por bloque y el número de bloques que forman la malla.

Otro elemento importante dentro de un programa de CUDA es *cudaMemcpy()*, que similarmente a lo que hace su versión en C (*memcpy()*) permite realizar copias de memoria. Lo que hace importante a esta función es que nos permite realizar copias de memoria entre GPU-CPU y viceversa, lo que será imprescindible a la hora de trabajar con sistemas heterogéneos formados por CPU y GPU.

Por último cabe citar cómo se realiza la reserva de memoria en la zona de memoria global de la GPU, permitiendo esto también el acceso a esta zona desde la CPU. Esta puntualización es importante ya que tanto la CPU como la GPU mantienen espacios propios de memoria, por lo que únicamente reservando memoria en la zona de memoria global se podrá acceder a los mismos datos tanto desde la CPU como desde la GPU.

Reserva de Memoria en CUDA

En el apartado anterior se ha tratado la necesidad de una reserva de memoria global en un programa CUDA. A la hora de realizar esta tarea, la función *cudaMalloc()* es la principal forma de efectuar esta tarea. Sin embargo, existen otros tipos de reserva memoria que también nos permiten realizar esta función.

Estas son las principales características que presentan los distintos tipos de reserva de memoria [32] [8]:

- cudaMalloc: La memoria en el *host* (CPU) se asigna mediante paginación lo que hace que a la hora de realizar transferencias a *device* (GPU) primeramente haya que asignar memoria fija y posteriormente realizar la transferencia a *device*.

- cudaMallocHost: Se asigna la memoria con "página bloqueada" en el *host*, lo que garantiza que el sistema operativo nunca ubicará la memoria fuera de esta zona de memoria. Esto permite que toda la memoria permanezca reservada en la zona de memoria fija, lo que nos ahorra el paso de realizar reserva de memoria fija desde la zona de paginación. Esto nos posibilita optimizar las copias en memoria al realizar transferencias de datos entre GPU (Figura 2.6). No debe asignarse una cantidad excesiva de memoria mediante *cudaMallocHost*, ya que esto puede producir una bajada del rendimiento general del sistema al estar reduciéndose la cantidad de memoria fija disponible para el sistema operativo y otros programas. Es complicado determinar con anticipación cuanto es una cantidad excesiva por lo que deben probarse los sistemas y aplicaciones a ejecutar para obtener los parámetros óptimos.

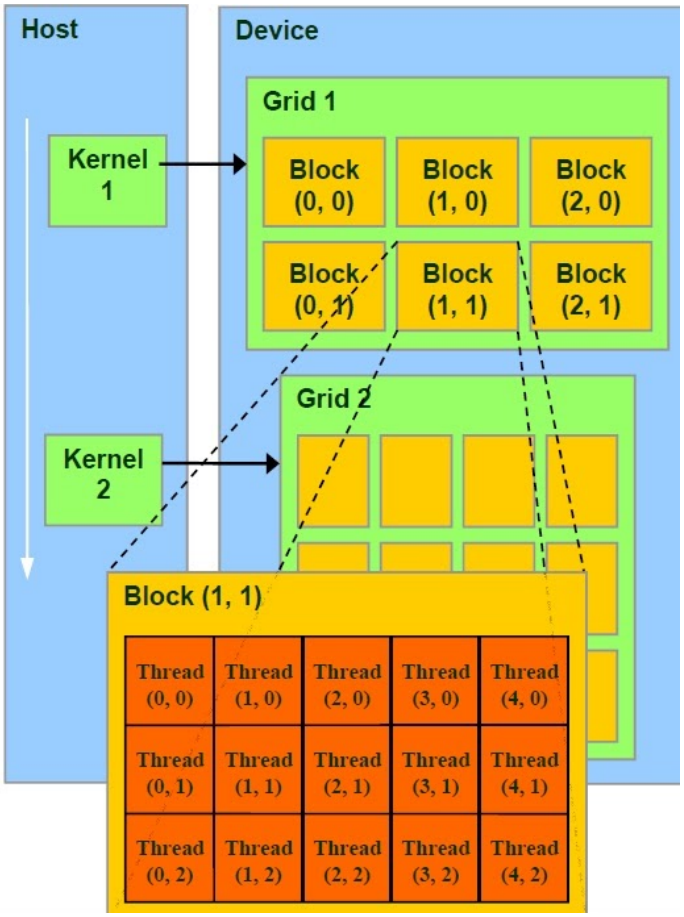


Figura 2.5: Representación de la ejecución de una comunicación entre un *host* (CPU) y *device* (GPU) en CUDA, se pueden apreciar dos kernel que llama cada uno a un *grid*, el primero a un *grid* compuesto por 6 bloques que a su vez están compuestos por 15 hilos y el segundo compuesto por 12 bloques.

- cudaMallocManaged: Al igual que *cudaMalloc* utiliza paginación para realizar la reserva de memoria. La particularidad de este tipo de reserva de memoria es que se crea un espacio de memoria común para GPU y CPU. Permite eliminar punteros duplicados (Figura 2.7).

2.3.3. GPUDirect

A la hora de realizar un programa en CUDA y realizar una comunicación entre distintas GPU es necesario llevar a cabo una copia de memoria de los *buffers* de GPU a CPU y de CPU a GPU para transferir adecuadamente los datos. Esta copia se realiza mediante *cudaMemcpy()*. Dicha operación causa un sobrecoste, ya que habría que realizar copias del *buffer* a transferir cada vez que varias GPU tuvieran que comunicarse.

Este problema puede solucionarse gracias a una tecnología conocida como GPUDirect que permite realizar estas transferencias directamente eliminando la necesidad de realizar estas copias. Algunas de las funcionalidades que proporciona GPUDirect [50] son GPUDirect RDMA [51] (Figura 2.8) que permite realizar la transferencia de los *buffers* directamente a tarjetas de red como Infiniband o GPU Direct P2P (Peer-to-Peer) (Figura 2.9) que permite enviar estos *buffers* directamente entre distintas GPU.

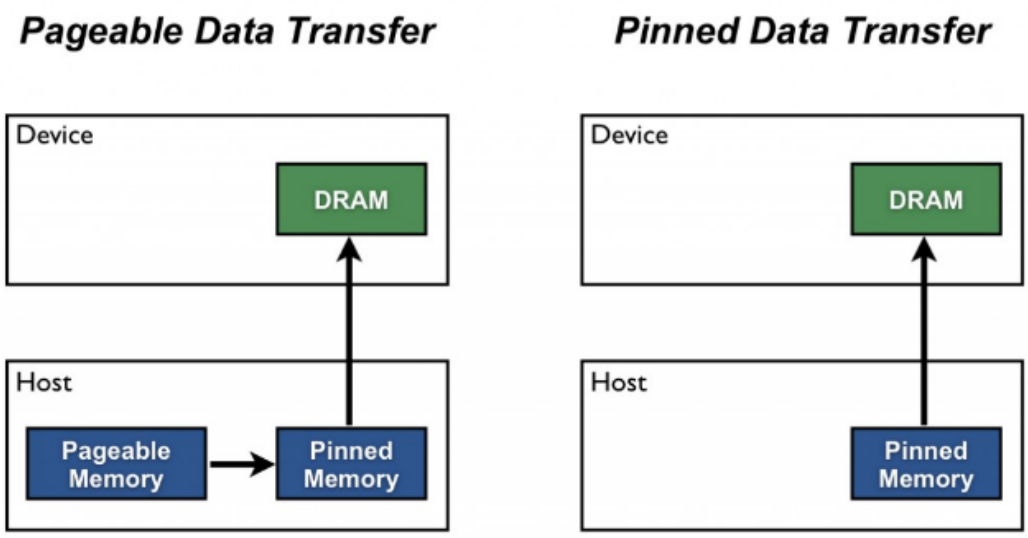


Figura 2.6: Comparativa de reserva de memoria *cudaMalloc()* (izquierda) con *cudaMallocHost()* (derecha)

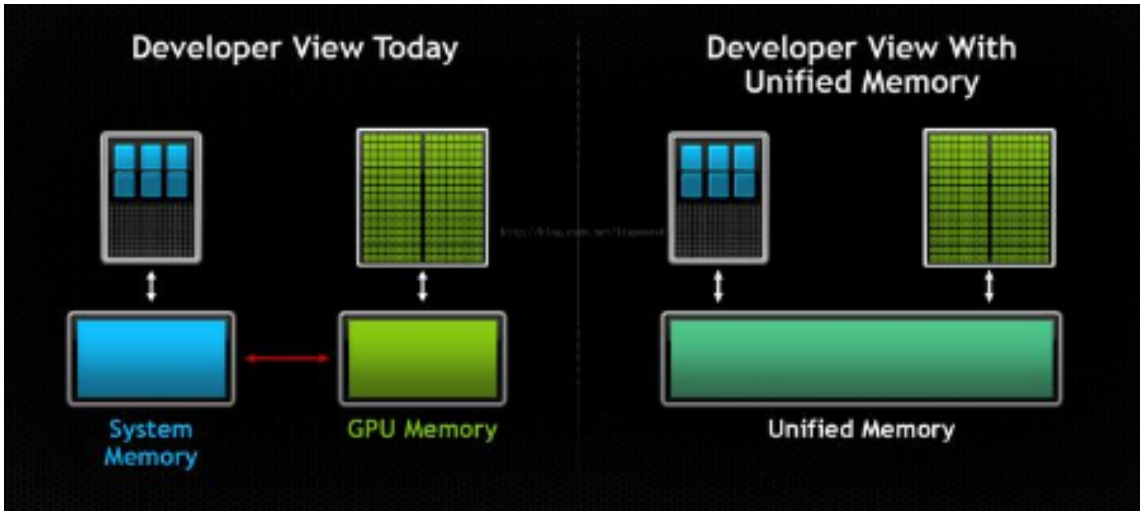


Figura 2.7: Comparativa de reserva de memoria *cudaMalloc* (*Developer View Today*) con *cudaMallocManaged* (*Developer View Today with Unified Memory*)

2.4 Tipos de Comunicación GPUDirect

Para la implementación de GPUDirect se van a considerar y evaluar dos alternativas [41] :

MPI CUDA-Aware

Esta alternativa nos permite combinar MPI con CUDA [4]. Esto nos permite eliminar llamadas a *cudaMemcpy()*, ya que en una versión con la implementación habitual de MPI únicamente pueden usarse punteros al *host* (CPU) a la hora de referenciar los datos a enviar o recibir, por lo que era necesario realizar la llamada de copia de memoria explícitamente para comunicar varias GPU al no poder usarse punteros a device (GPU) al llamar a una primitiva de comunicación. Sin embargo, con MPI CUDA-aware combinamos MPI y CUDA añadiéndole a MPI la capacidad de usar punteros a *device (GPU)*.

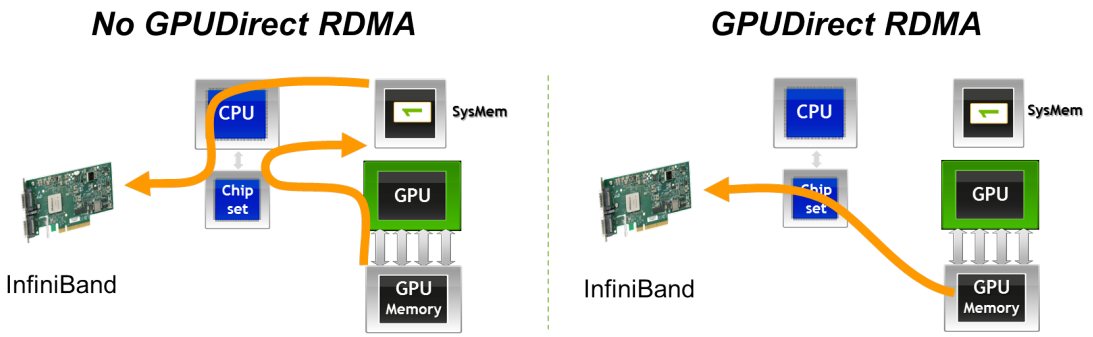


Figura 2.8: Comparación entre la comunicación de un sistema con y sin transferencias GPUDirect RDMA

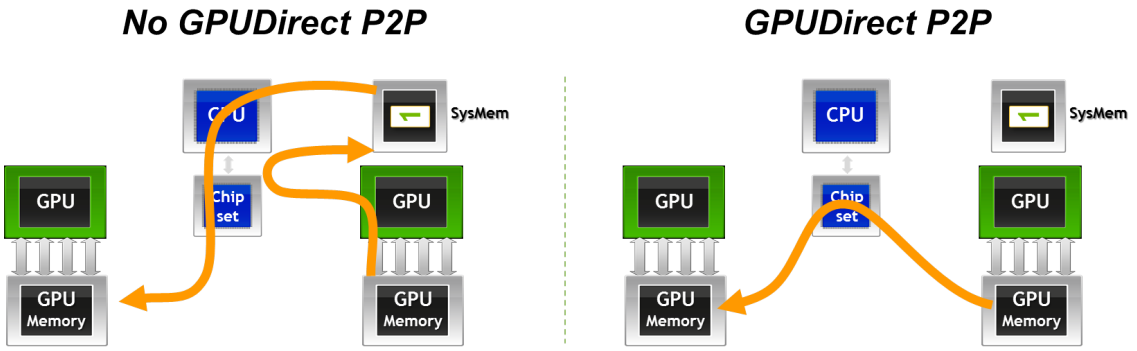


Figura 2.9: Comparación entre la comunicación de un sistema con y sin transferencias GPUDirect P2P

Además, usaremos las técnicas ya mencionadas GPUDirect P2P y GPUDirect RDMA para realizar las transferencias directamente de GPU a GPU o de GPU a Infiniband, ya que sin esta tecnología las copias de memoria se seguirían realizando implícitamente.

La implementación de MPI CUDA-Aware será realizada gracias a las librerías Open-MPI [27] y mvapich2 [24] que implementan primitivas que nos permiten eliminar las copias de memoria.

NCCL

NCCL [26]: es una librería de comunicación colectiva basada en GPU y orientada para cargas de trabajo de *deep learning*. Esta librería toma como base las operaciones de MPI y proporciona una interfaz propia con una implementación de las primitivas de comunicación colectivas como *Broadcast, All-Gather, Gather, Reduce, All-Reduce, Scatter o All-Scatter*. Además, también implementa operaciones punto a punto pese a estar orientada principalmente para operaciones colectivas.

NCCL, al igual que MPI Cuda-aware, implementa GPU-Direct, pudiendo realizar tanto copias de memoria directas entre GPU locales y también remotas utilizando Infiniband. Además, mientras que MPI está orientado a una comunicación eficiente entre una gran cantidad de nodos sea cual sea su naturaleza, NCCL está optimizado expresamente para sistemas multi-GPU por lo que permite acelerar las comunicaciones en este tipo de sistemas.

2.5 Descripción entorno utilizado

En este trabajo hemos utilizado el cluster ALTEC, del grupo de investigación. Este *cluster* se compone de 9 nodos de los cuales podemos aprovechar 8, ya que el primero de ellos se utiliza como nodo de acceso y ejecuta diversos servicios del sistema.

ALTEC es compartido por varios usuarios. Por tanto los trabajos deben lanzarse mediante el sistema de colas SLURM [14] que nos permite reservar una serie de nodos para realizar una ejecución. Pese a esto, en ciertos momentos en los que trabajos de distintos usuarios se encuentran en ejecución al mismo tiempo, se producen interferencias que pueden causar la obtención de resultados de tiempos de ejecución alterados.

Respecto a las características de ALTEC cabe decir que cada uno de estos nodos está compuesto por una CPU *Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz, 28 núcleos* (fig 2.10), 191 GB (GigaBytes) de memoria y una GPU Tesla V100 (fig 2.11) con 5120 núcleos CuDA y velocidad de 1455 MHz (Megahercios). Además se cuenta con una red Infiniband (4X EDR[Enhanced Data Rate]).

Figura 2.10: Imagen de un Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz, 28 núcleos

Figura 2.11: Imagen de una GPU Tesla V100

CAPÍTULO 3
Contexto y Entorno de trabajo

En este capítulo se describe tanto que son las redes neuronales, el *deep learning* [10] así como el entrenamiento distribuido y una descripción de la herramienta HELENNA.

3.1 Redes Neuronales

Las redes neuronales artificiales son uno de los múltiples modelos computacionales disponibles dentro del campo de la Inteligencia Artificial. Esta clase de sistemas permiten resolver ciertos problemas que a través de la programación tradicional serían muy difíciles de resolver, ya que en vez de programarse de forma explícita aprenden por sí mismos. Algunos ejemplos de tareas que permiten resolver estas redes neuronales son el reconocimiento de voz o la visión por computador.

Las redes neuronales están formadas por un conjunto de unidades llamadas neuronas artificiales. Éstas se encuentran conectadas entre sí para poder transmitirse datos entre ellas. Esta red además recibe una serie de datos de entrada, que tras ser procesados por toda la red neuronal permiten producir una salida.

Las neuronas que conforman la red neuronal están organizadas en una serie de capas. Cada una de las neuronas de una capa están conectadas con todas las neuronas de las siguientes capas. Esta serie de capas junto a las neuronas que las conforman y sus conexiones forman lo que conocemos como una red neuronal artificial.

Para comprender mejor las redes neuronales y como estas pasaron de ser una tecnología, que debido a sus limitadas funcionalidades prácticas llego a ser descartada por muchos de los investigadores de la época, a una de las corrientes de investigación más comentadas del momento, haré un breve repaso sobre su historia desde su descubrimiento en los años 50, hasta la actualidad.

3.1.1. Historia de las redes neuronales

Se puede llegar a creer que las redes neuronales fueron descubiertas en el siglo XXI, ya que su popularidad ha estallado a lo largo de los últimos años. Sin embargo, esto fue así. Fue durante los años 40 cuando se dieron los primeros pasos en la creación de esta tecnología [33].

Este desconocimiento sobre sus orígenes se debe a que los enfoques utilizados durante el siglo XX eran impopulares, ya que presentaban distintos defectos. Además, a lo largo de los años la terminología a la hora de referirse a esta tecnología ha ido cambiando. Términos como el *Deep Learning* no fueron introducidos hasta el 2006.

La historia de las redes neuronales puede ser comprendida dentro de 3 olas de avances. La primera conocida como cibernética o *Cybernetics* entre los años 40 y 60, la segunda como conexionismo o *Connectionism* entre los años 80 y 90, y una última conocida como *Deep Learning* o aprendizaje profundo a partir de 2006 hasta la actualidad.

Las investigaciones llevadas a cabo durante las 2 primeras olas fueron impopulares debido a sus defectos. Sin embargo, ayudaron a sentar las bases de lo que hoy en día son los algoritmos de redes neuronales. Procedamos a abordar los avances conseguidos en cada una de estas olas.

Cybernetics

Durante esta ola de avances entre 1940 y 1960 se plantearon los primeros modelos de redes neuronales, predecesores de las redes neuronales actuales. Las primeras investigaciones se realizan tomando como base el aprendizaje biológico (el funcionamiento del cerebro humano). Su objetivo es replicar el funcionamiento de un cerebro mediante un modelo computacional más simple, que permita construir sistemas que comiencen a aprender como lo haría un cerebro humano y permitan obtener conclusiones a partir de una serie de entradas. Este es un enfoque que sería descartado en años posteriores, dentro de lo que conocemos como modelos de redes neuronales. Sin embargo, actualmente esto se sigue investigando en el campo de la neurociencia computacional [25].

Esta ola de investigaciones comenzó en 1943 con el desarrollo de la neurona McCulloch-Pitts. Esta fue un intento de replicar el funcionamiento de una neurona biológica. Se basa en un modelo lineal que coge una serie de entradas, un peso asociado a estas entradas y una salida que puede ser 0 o 1. Los pesos se fijaban manualmente por un ser humano.

En la siguiente Figura 3.1, podemos apreciar como este tipo de neurona está dividida en dos partes. Una parte g que toma una serie de entradas con sus respectivos pesos y realiza una agregación de estos. Basándose en esta agregación la parte f tomará una decisión.

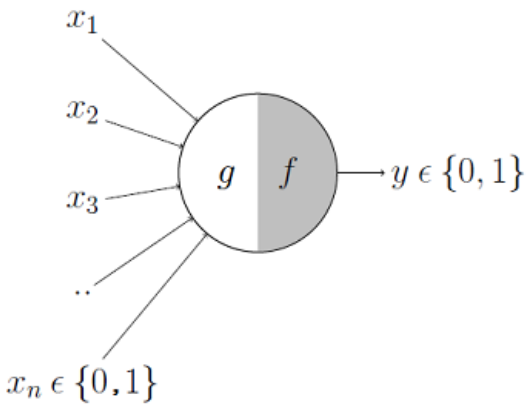


Figura 3.1: Representación del modelo de neurona McCulloch-Pitts

En los años 50, Frank Rosenblatt desarrollo Perceptron, una neurona artificial que al contrario que la neurona McCulloch-Pitts puede aprender que pesos asignar automáticamente. Inicialmente Perceptron fue concebido como una máquina (Figura 3.2) en vez de un programa. Esta máquina diseñada para realizar reconocimiento de imágenes, contenía células fotoeléctricas conectadas a múltiples neuronas que podían clasificar las entradas capturadas por las células fotoeléctricas.

Perceptron inicialmente parecía prometedor, pero rápidamente sé probo que no podía ser entrenado para reconocer muchas clases de patrones. Causando esto un estancamiento en este campo de estudio durante muchos años. Pese a esto, la importancia de Perceptron sería muy grande en el futuro, ya que plantó las bases que definen lo que es una neurona artificial hoy en día.

Durante el mismo periodo de tiempo al desarrollo de Perceptron, Bernard Widrow desarrolló ADALINE (ADAptive LInear NEuron). Esta neurona permitía adaptar los pesos basándose en la suma de los pesos de las entradas durante la fase de entrenamiento. Mientras que Perceptron únicamente utilizaba etiquetas de clase para permitirnos conocer si una predicción era correcta o no, ADALINE nos permite "cuantizar" como de buena o mala ha sido la predicción.

En la Figura 3.3 podemos apreciar una comparativa entre ambos modelos. Podemos observar que el modelo introducido por Perceptron es bastante similar al utilizado en las neuronas actuales, introduciendo conceptos como la función de activación o el uso del error en las predicciones para reajustar los pesos. También podemos apreciar como ADALINE es similar a este, con la diferencia de que introduce una "cuantización" de la calidad de la predicción previa a la salida. La función de aprendizaje de ADALINE es similar al *stochastic gradient descent* utilizado hoy en día en regresión lineal.

Estos modelos lineales presentan ciertas limitaciones, que causaron un descenso de su popularidad y un estancamiento en las investigaciones por un tiempo. Al estancarse las investigaciones de este tipo de modelos inspirados en investigaciones en el campo de la neurociencia, se empezaron a explorar modelos alejados de la neurociencia para resolver los distintos problemas presentados.

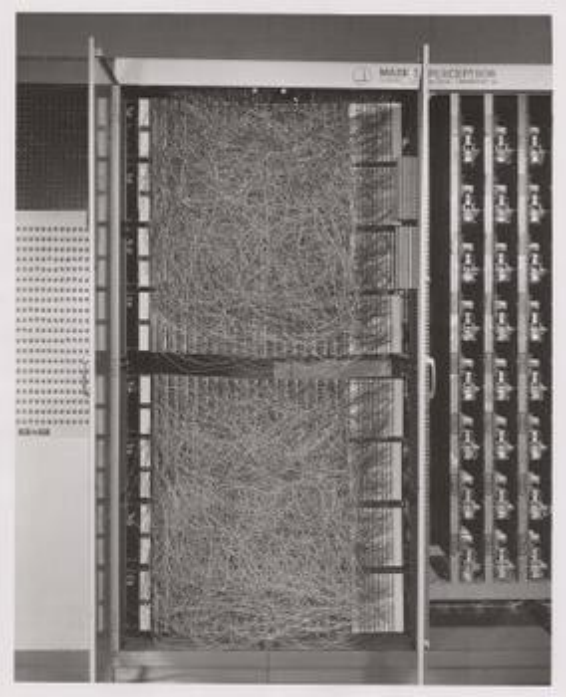


Figura 3.2: Imagen de la máquina conocida como Perceptron desarrollada en los años 50 por Frank Rosenblatt

Connectionism

Durante esta ola, proliferaron las investigaciones basadas en ciencias cognitivas (estudio científico de la mente y sus procesos). El objetivo detrás del desarrollo de las redes

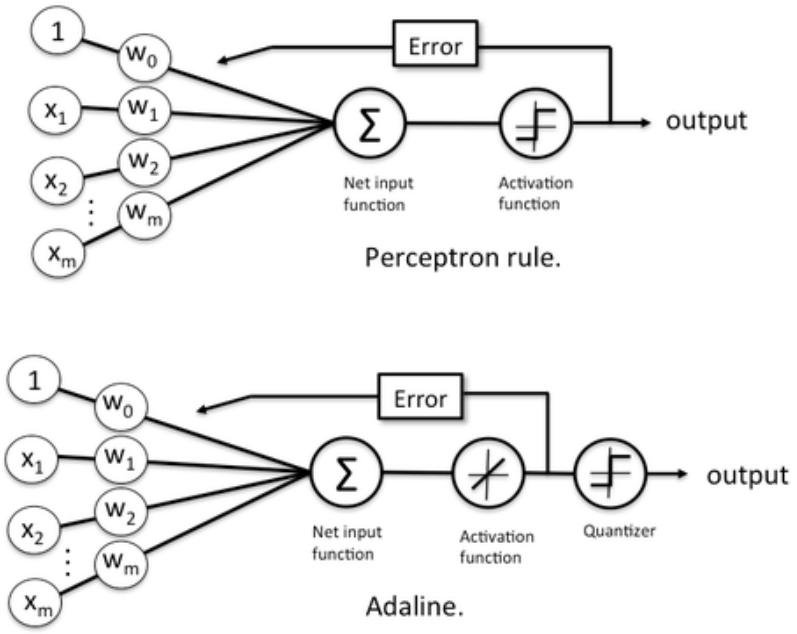


Figura 3.3: Comparativa de una neurona Perceptron y ADALINE

neuronales artificiales, era conformar una red compuesta por distintas unidades que puedan ser programadas para presentar un comportamiento inteligente. En este periodo se comenzó a hablar por primera vez sobre el concepto de capas ocultas entre las capas de entrada y salida, además de introducido el concepto de red neuronal artificial .

Durante este mismo periodo se desarrollaron distintos modelos para entrenar redes neuronales profundas como *back-propagation* o LSTM (Long short-term memory) [23] que continúan siendo claves para realizar entrenamientos de redes neuronales profundas hoy en día.

Este enfoque causó mucha expectación, ya que las investigaciones percibían que su comportamiento era bastante similar al presentado dentro de nuestro sistema nervioso. Sin embargo, a mediados de los 90 las *startups* basadas en IA empezaron a hacer afirmaciones poco realistas que esta tecnología era incapaz de cumplir, ya que el nivel de sofisticación de estos modelos no era lo suficientemente elevado. Esto se debía a que en esta época se carecía de los recursos de cálculo necesarios. Esta falta de sofisticación causó un descenso de la popularidad de este tipo de modelos.

Deep Learning

Tras estas dos olas que terminaron con una bajada de interés en las investigaciones que hacían uso de este tipo de modelos, emergió una nueva ola a partir de 2006. Esto sucedió gracias a que Geoffrey Hinton hizo uso el primer modelo *Deep Learning* conformado por una gran cantidad de capas ocultas para realizar un entrenamiento de redes neuronales.

Estos avances fueron usados por otros investigadores para entrenar distintos tipos de redes neuronales profundas. Investigadores de todo el mundo comenzaron a entrenar redes más y más profundas causando la popularización del término *Deep Learning*.

Las investigaciones de Geoffrey Hinton fueron claves en la popularización del *Deep Learning*. Sin embargo existieron otros factores que también fueron muy importantes en la mejora de la eficacia de este tipo de modelos. Algunos de estos factores son el incremento

de la capacidad de cálculo o la disponibilidad de *datasets* más grandes. Estas circunstancias posibilitaron que incluso algoritmos ideados en la ola del conexionismo comenzaran a presentar mejores resultados.

Estas redes permitieron tanto incrementar la precisión respecto a otras alternativas a la hora de resolver ciertas tareas como el reconocimiento de imagen, además de poder resolver problemas más complejos, que una red simple era incapaz de resolver.

3.1.2. Neuronas y funciones de activación

Una neurona artificial (Figura 3.4) es una función matemática que toma una serie de entradas. Cada entrada tiene un peso que define la importancia de dicha entrada para esa neurona. Además, existe el *bias*, una constante que fija lo predispuesta que está una neurona a que su salida tome el valor 0 o 1 independientemente de sus pesos y entradas. Una vez obtenida la suma de estos factores, este resultado pasará por una función de activación que marcará bajo que criterios se activará o no la neurona (si devuelve 0 no se activa y si devuelve 1 si se activa).

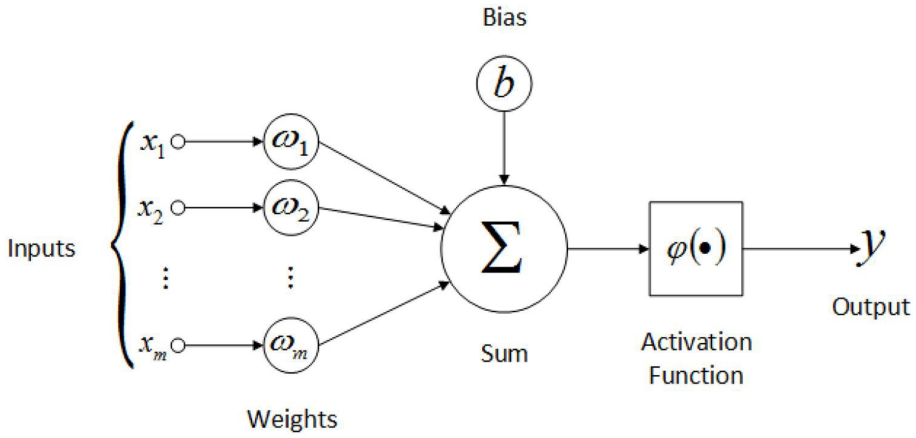


Figura 3.4: Esquema de las distintas partes que conforman una neurona.

Existe una gran cantidad de funciones de activación cuyo uso permite modificar el comportamiento de una red neuronal, entre ellas algunas de las más utilizadas son: Sigmoide, Softmax y ReLu [2] [3].

3.1.3. Estructura de las Redes Neuronales

Como se ha comentado anteriormente, las redes neuronales están organizadas en capas formadas por neuronas (Figura 3.5). Cada una de las neuronas de una capa se conecta con todas las neuronas de la siguiente capa. La primera capa se conoce como capa de entrada, ya que es aquella que recibe los datos de entrada. La última capa se conoce como capa de salida, ya que desde esta capa es visible la salida de la red neuronal. Por último, existen las capas situadas entre las capas de entrada y salida, estas son conocidas como capas ocultas y su número varía dependiendo de la red neuronal utilizada pudiendo ser bajo para redes neuronales convencionales y muy alto para redes neuronales profundas (*Deep Learning*). Estas capas, al igual que la capa de salida, toman como entrada la salida de la capa anterior.

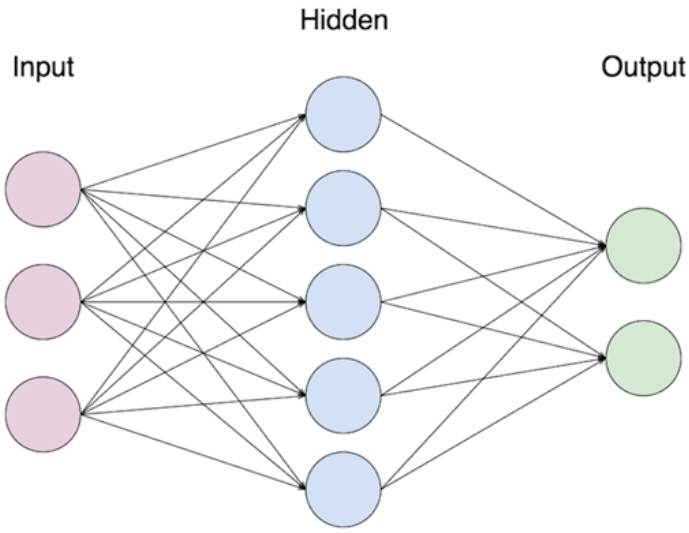


Figura 3.5: Red Neuronal conformada por una capa de entrada, una capa oculta y una capa de salida, cada una formada por 3, 5 y 2 neuronas, respectivamente.

3.1.4. *Entrenamiento de Redes Neuronales*

Una vez explicadas las distintas partes que forman una red neuronal cabe resaltar qué procesos deberá realizar un modelo basado en redes neuronales para su correcto funcionamiento. El primero es la fase de entrenamiento, en el cual permitimos al modelo aprender ciertos patrones a partir de unos datos de entrada. El segundo es la fase de inferencia en el que tras entrenar el modelo, lo utilizamos para realizar predicciones y así comprobar la fiabilidad de este.

Respecto a la fase de entrenamiento de redes neuronales, esta se lleva a cabo ajustando los pesos de las entradas de cada una de las neuronas, tratando de obtener una salida que se aproxime a la salida deseada. Existen distintos algoritmos para llevar a cabo este proceso. En este trabajo nos centraremos en el algoritmo de *backpropagation*, ya que es el algoritmo más utilizado para realizar dicho entrenamiento, además de ser el algoritmo utilizado en la herramienta sobre la que trabajaremos.

En este algoritmo se modifica el peso de cada una de las neuronas para obtener el resultado deseado. Esta modificación se realiza teniendo en cuenta el error obtenido en la red neuronal respecto a la salida y la importancia (peso) de cada una de las neuronas en la obtención de este resultado.

Como podemos ver en la figura 3.6 en este proceso existe una primera fase llamada *forward* en la que se realiza la predicción desde la capa de entrada a la capa de salida. Es decir, que a partir de los datos de entrada cada una de las neuronas transmite su salida a las neuronas de la capa posterior, utilizando las salidas de las neuronas de la capa anterior como su entrada. Este proceso se repite secuencialmente hasta que se llega a la capa de salida obteniéndose una predicción basándonos en los datos de entrada. Tras esto se utiliza una función de pérdida para calcular cuál ha sido el error de la predicción comparando el resultado obtenido con el resultado real.

Tras calcular el error se realiza el *backpropagation*, el cual propaga desde la capa de salida a la de entrada el error obtenido. Al igual que en *forward* cada neurona de una capa transmite el error a todas las neuronas de la capa anterior. Dicho error se distribuye entre todas las neuronas dependiendo del peso específico de cada una de ellas en la obtención de la salida que ha ocasionado este error.

Una vez completo el *backpropagation* se reajusta el peso de las neuronas. Esto se hace mediante una técnica llamada descenso de gradiente. Esta técnica nos permite calcular los ajustes necesarios en los pesos gracias al cálculo de la derivada de la función de pérdida, teniendo como objetivo con esto disminuir lo máximo posible el error total. Existen distintas versiones del descenso del gradiente [37]:

- Descenso del gradiente en lotes (*Batch Gradient Descent*): se introducen todos los datos disponibles a la vez. Es decir que *Batch Gradient Descent* calcula el error para cada observación en el lote (el conjunto de datos de entrenamiento) y actualiza el modelo una vez se hayan evaluado todas las observaciones, o lo que es lo mismo, se actualizan los parámetros al final de cada época (*epoch*) (una época se refiere a una iteración a través de todos los datos de entrenamiento).

  Al usar todos los datos de entrenamiento de una vez se alcanza más rápidamente el mínimo y es computacionalmente más eficiente, ya que únicamente realiza actualizaciones cada época. Sin embargo, esto también causa problemas de estancamiento. Al usarse siempre todas las muestras llegará un momento dado en el que las variaciones son mínimas.

- Descenso del gradiente estocástico (*Stochastic Gradient Descent*): en cada iteración se introduce una única muestra concreta aleatoria. Esta aleatoriedad dificulta que se produzca un estancamiento. El problema que presenta esta versión es su lentitud, ya que necesita de muchas más iteraciones para realizar el entrenamiento. Además de no aprovechar los recursos disponibles.

- Descenso del gradiente en mini-lotes (*Mini-Batch Gradient Descent*): en vez de utilizar una única muestra como en *Batch Gradient Descent* se introducen N muestras cada iteración. Esto permite tanto obtener aleatoriedad como en *Stochastic Gradient Descent* como conseguir un entrenamiento más rápido al poder paralelizar las operaciones en distintos nodos.

Entre estos tres tipos de descenso de gradiente se utilizará el *Mini-Batch Gradient Descent* debido a su balance entre aleatoriedad y tiempo de entrenamiento. A la hora de utilizar este tipo de entrenamiento habrá que elegir un valor N para este tipo de descenso de gradiente teniendo en cuenta que a mayor valor de N menor será su tiempo de entrenamiento y mayor su aleatoriedad, por lo que habrá que elegir un valor que permita obtener un equilibrio entre estos dos aspectos. Este valor se conoce como *batch* y determina cuantas muestras se introducen en cada iteración.

Otro parámetro a tener en cuenta es el de *learning rate*. Este indica el porcentaje de cambio que sufrirán las actualizaciones de los pesos. Como podemos ver en la figura 3.7, un *learning rate* más grande permite reducir el error de forma más rápida, pero menos precisa, lo que puede provocar que se converja rápidamente en dirección del mínimo error global, pero que nunca llegue a alcanzarlo, mientras que con un *learning rate* pequeño se acercará de forma más precisa al mínimo pero de forma más lenta.

Además del *learning rate*, otros parámetros importantes a la hora de realizar el entrenamiento de una red neuronal son el citado *batch size* que indica el número de muestras del *dataset* utilizadas por cada nodo antes de actualizar los pesos del modelo o el *epoch* que indica cuantas veces se pasa a través del conjunto del *dataset* a la hora de realizar el entrenamiento (Figura 3.8).

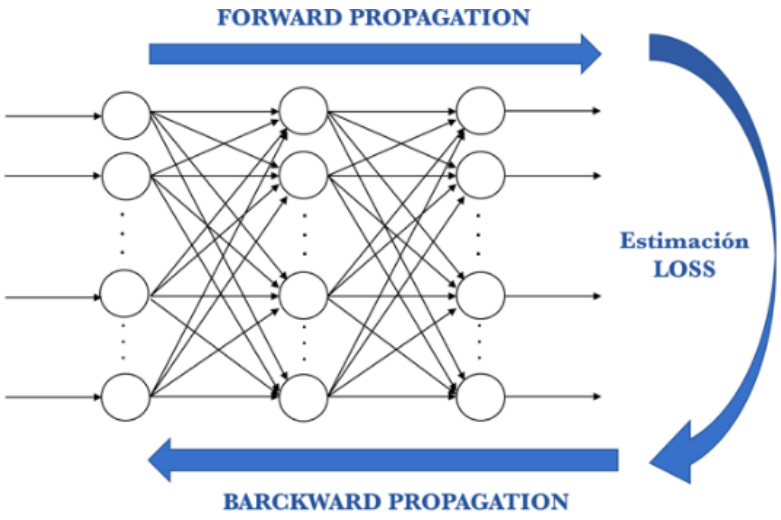


Figura 3.6: Representación Backward y Forward Propagation en una red neuronal.

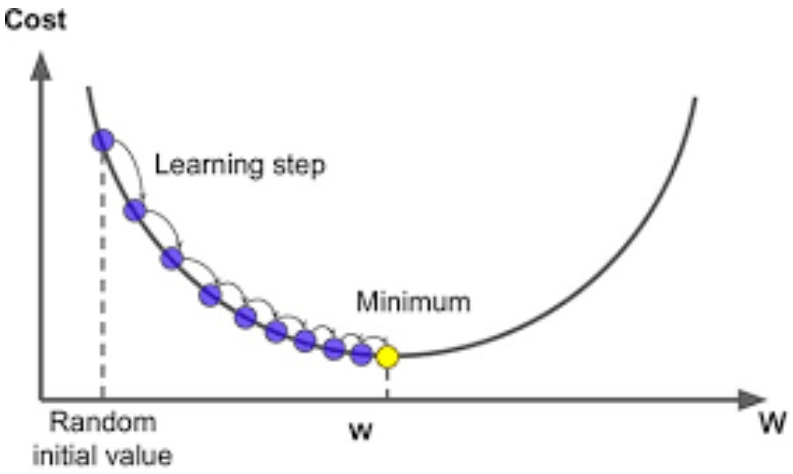


Figura 3.7: Representación del descenso del gradiente con un *learning rate* dado desde un valor inicial aleatorio hasta llegar a un mínimo.

3.2 *Deep Learning* y problemática

El *Deep Learning* (Aprendizaje profundo) o DNN (*Deep Neural Networks*) es un conjunto de algoritmos de aprendizaje dentro del campo de algoritmos de redes neurales. Estos algoritmos se diferencian de un algoritmo estándar de redes neuronales, por la gran cantidad de capas ocultas que pueden presentar (Figura 3.9).

3.2.1. Problemática

Tal y como hemos podido apreciar en este pequeño repaso de la historia de los algoritmos de *Deep Learning*, un factor muy importante a la hora de hacer uso de este tipo de algoritmos es la capacidad de cálculo disponible para realizar los entrenamientos de redes neuronales. Estos algoritmos son muy exigentes computacionalmente, por lo que si no se dispone de los recursos suficientes, el tiempo de ejecución necesario para realizar un entrenamiento de una red neuronal profunda medianamente compleja puede llegar a ser demasiado elevado.

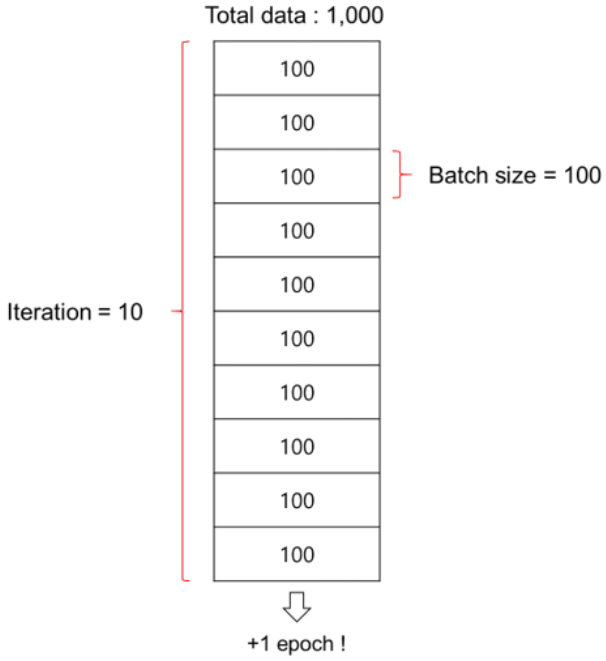


Figura 3.8: En esta figura se muestra un *dataset* de tamaño 1000, y con *batch size* de tamaño 100 que necesita de 10 iteraciones en las que actualiza los pesos del modelo para completar un *epoch*

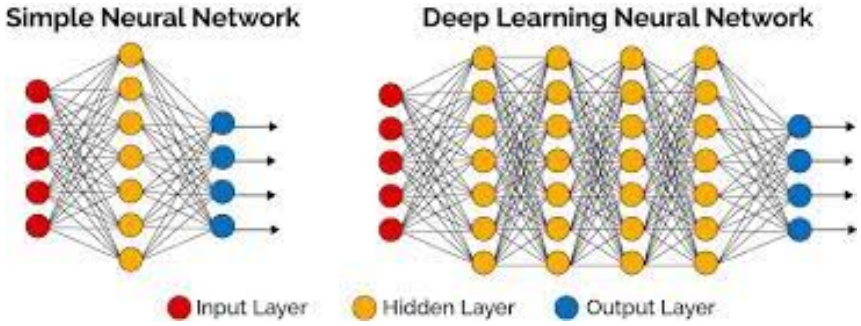


Figura 3.9: Comparativa de red neuronal simple con red neuronal profunda (*Deep Learning*)

Existen distintos factores que han permitido incrementar la capacidad de cálculo a lo largo de los últimos años. Algunos de estos son, la mejora en la velocidad de las unidades de procesamiento, el desarrollo de las GPU que permiten realizar estas operaciones de cálculo de forma mucho más rápida, o a la aparición de arquitecturas de procesamiento distribuido, que permiten aumentar la capacidad de cálculo sin la necesidad de acumular todos los recursos en un mismo equipo. Además, estas arquitecturas distribuidas permiten realizar un entrenamiento paralelamente, pudiendo así incrementar la velocidad de cálculo.

Un correcto uso de todos estos factores permite reducir el tiempo de ejecución necesario para realizar el entrenamiento de redes neuronales profundas, permitiendo que el entrenamiento de redes neuronales complejas sea asumible desde el punto de vista computacional.

3.3 Entrenamiento distribuido de redes neuronales

Para realizar el entrenamiento distribuido de una DNN, se hace uso de sistemas distribuidos [45]. Un sistema distribuido consiste en un conjunto de sistemas de cálculo

trabajando juntos como si fueran un mismo equipo [34]. El uso de estos sistemas permite distribuir la carga de trabajo entre varios nodos, pudiendo así acelerar el proceso de entrenamiento. Existen dos formas principales de entrenar una red neuronal [46] [1] mediante un sistema distribuido, el paralelismo de modelo o el paralelismo de datos.

- Paralelismo de modelo (Figura 3.10): el modelo que conforma la red neuronal es divido en distintas partes, siendo cada nodo responsable de los cálculos de una de estas partes. Cuando la entrada de una neurona provenga de la salida de una neurona que se encuentra asignada a un nodo diferente se realizará la comunicación entre estos nodos (esto lo podemos ver representado en la Figura 3.10 donde estas comunicaciones se ven representadas con líneas amarillas que unen las neuronas como ocurre cuando una neurona del nodo 5 toma como entrada una salida de una neurona asignada al nodo 4). El rendimiento de este modelo suele ser peor que el de paralelismo de datos, ya que requiere de muchas más comunicaciones.

- Paralelismo de datos (Figura 3.11): se replica un mismo modelo en diferentes nodos, a los que se les asigna una parte del *dataset*. El *batch* asignado se divide entre los distintos nodos, teniendo cada uno un *batch* local que es igual al *batch* global dividido por el número de nodos. A la hora de realizar la sincronización esta únicamente se realizará en la fase de *backward*, en la que se intercambian los distintos gradientes necesarios para el entrenamiento. En la fase *forward* esto no será necesario, ya que todos los nodos tienen una copia del modelo.

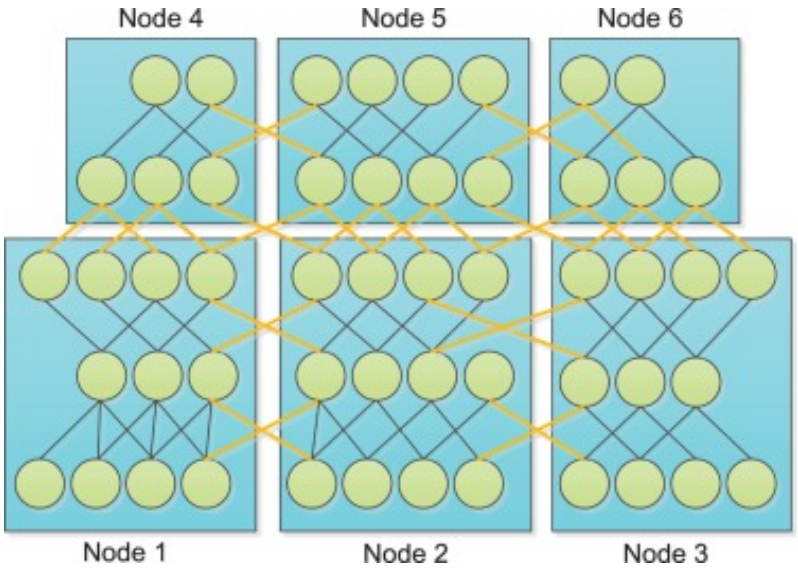


Figura 3.10: Representación de entrenamiento distribuido utilizando paralelismo de modelo para 6 nodos

Entre estos dos modos de realizar entrenamientos distribuidos, el utilizado en la herramienta HELENNA es el paralelismo de datos, ya que nos permite reducir el número de comunicaciones.

A la hora de realizar un entrenamiento distribuido, uno de los aspectos más importantes a tener en cuenta es como los diferentes nodos comparten y sincronizan sus resultados. Este aspecto es muy importante, ya que afectará tanto a la precisión como al tiempo de ejecución obtenido. Una sincronización demasiado frecuente puede causar que debido al elevado número de comunicaciones no se aproveche la paralelización presente en un entrenamiento distribuido y una demasiado escasa que la precisión del modelo se vea

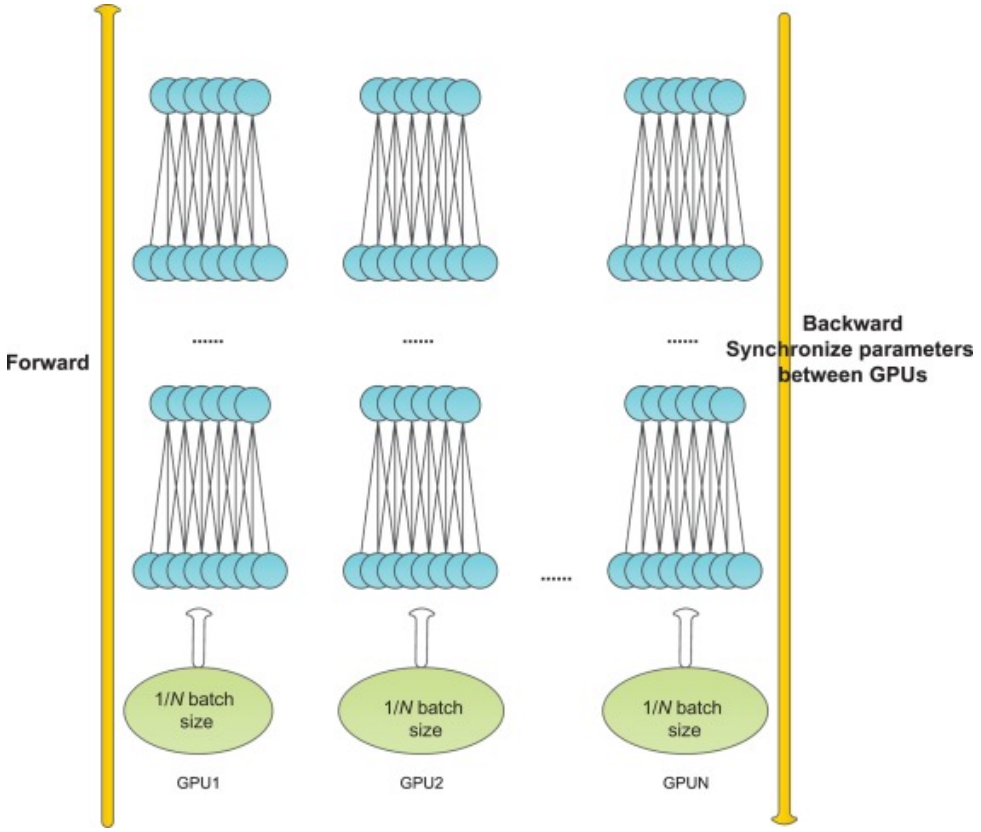


Figura 3.11: Representación de entrenamiento distribuido utilizando paralelismo de datos para N nodos

resentida. Incluso se puede dar el caso de que sea más costosa temporalmente la versión paralela de un entrenamiento que la secuencial.

En un entrenamiento distribuido con paralelismo de datos, se dispone de distintas formas de realizar la sincronización entre los distintos nodos. Algunas de las principales son [11]:

- *Parameter Server* (Figura 2.1): está conformada por *n* nodos entre los cuales existe un nodo maestro y *n-1* nodos esclavos. Los datos de entrada se distribuyen entre los nodos esclavos, los cuales realizan el cálculo de gradiente y envían el resultado al maestro. Estos nodos esclavos no se comunican entre sí, únicamente lo hacen con el maestro cuya función es mantener los parámetros de los esclavos actualizados. Para ello se ocupa de recibir los cálculos de los gradientes por parte de los esclavos y transmitir los pesos actualizados a cada uno de ellos. En el caso de nuestra implementación de este modelo, esta actualización se lleva a cabo cada *batch*. Además, cabe decir que si el *batch* global fijado es x, cada uno de nuestros nodos tendrá un *batch* local de $\frac{x}{n-1}$.
- Entrenamiento síncrono (Figura 3.13). Esta formado por n nodos, el *batch size* se divide entre cada uno de los nodos por lo que si el tamaño del *batch* fuera x el *batch* local de cada uno de estos nodos sería $\frac{x}{n}$. Tras realizar el descenso de gradiente de su *batch* local, los distintos nodos actualizarán los valores de sus pesos mediante la llamada *Allreduce* que realizará una suma de los parámetros de cada uno de los nodos y la distribuirá entre todos estos nodos.
- Entrenamiento asíncrono [47] : presenta unas características similares al síncrono con la diferencia de que al contrario del síncrono en el que la comunicación se rea-

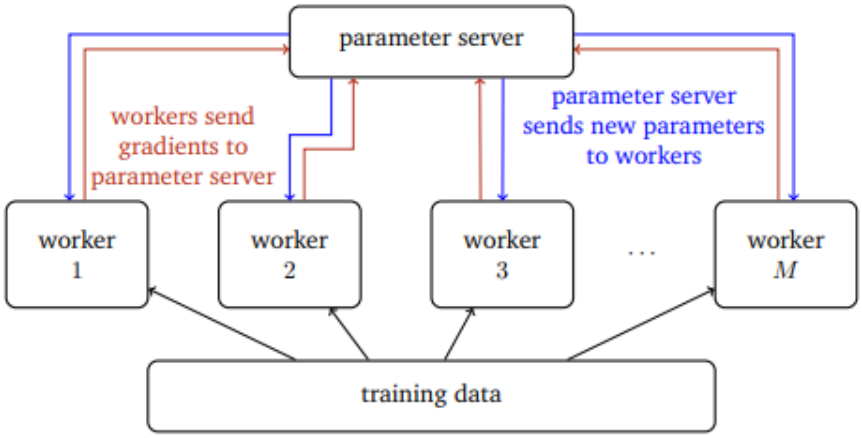


Figura 3.12: Arquitectura Parameter Server formada por M nodos esclavos

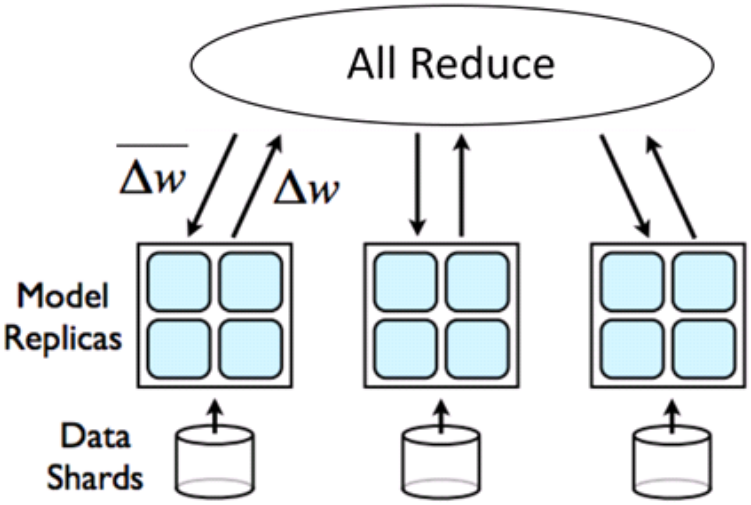


Figura 3.13: Arquitectura de comunicación síncrona formado por 3 nodos.

liza tras el cálculo del *batch* local, en este se comunica cada *x batches* siendo este parámetro una variable que habrá que fijar a la hora de realizar el entrenamiento.

3.4 Tau Commander

Tau Commmander [29] es una herramienta que nos permite analizar el comportamiento de un software para así determinar como de óptimo es este software para el sistema en el que se ejecuta. TAU commander nos permite analizar programas MPI, CUDA, OpenMP, C, C++ entre otros. A través de esta herramienta podremos conocer el impacto tanto de las comunicaciones como de otras operaciones en la ejecución de la aplicación. Gracias a esto podrá conocerse más detalladamente el impacto de estas comunicaciones en las prestaciones del sistema, pudiendo así realizar un análisis más exhaustivo.

Cualquier ejecución en Tau Commander se basa en tres conceptos: el objetivo (*target*), la aplicación (*application*) y las medidas (*measurement*). Esto lo podemos apreciar en la figura 3.14. En primer lugar tendríamos el objetivo, que describe el entorno donde los

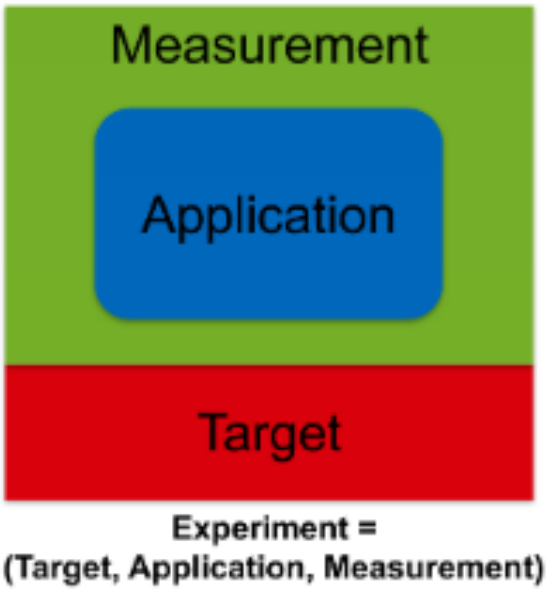


Figura 3.14: Estructura básica de Tau Commander

datos son recolectados. Esto incluye la plataforma donde se ejecuta el código, el sistema operativo, la arquitectura CPU, compiladores y software instalado. En segundo lugar tendríamos la aplicación, que consiste en los elementos subyacentes a esta como puede ser el uso de programas MPI, CUDA, Open MP o demás. Por último tendríamos las medidas que define que serán los datos y en qué formato se recogerán.

3.5 Descripción de la herramienta *HELENNA*

HELENNA es una plataforma software para el desarrollo de procesos de entrenamiento e inferencia de redes neuronales. HELENNA es el acrónimo de HEterogeneous LEarning Neural Networks Application. Como tal, tiene como objetivo el uso de arquitecturas heterogéneas de cálculo.

Aunque existen múltiples soluciones para el entrenamiento e inferencia de redes neuronales, HELENNA se ha desarrollado para explorar nuevos métodos de entrenamiento y nuevas redes neuronales en el contexto de proyectos de investigación. El uso de una plataforma desarrollada exclusivamente por el grupo de investigación GAP permite un mayor control sobre los procesos de entrenamiento e inferencia, y por tanto, una mejor adecuación para el estudio y análisis de procesos alternativos que mejoren su rendimiento.

HELENNA está desarrollado en lenguaje de programación C y C++ (principalmente en C) y tiene una estructura de código similar a la programación en objetos aunque no los utiliza. En concreto, HELENNA permite la definición de capas de redes neuronales aislando el código de la capa en un único fichero e implementando diferentes funciones básicas para el soporte del entrenamiento e inferencia de la capa. Desde este punto de vista, crear el soporte para una nueva capa conlleva la creación de un nuevo fichero y la instanciación de las funciones en función del tipo de capa a soportar. Cada capa soportada en HELENNA debe implementar cinco funciones básicas:

- Función *parse*. Esta función interpreta los parámetros de definición de la capa e inicializa sus estructuras

- Función *allocate*. Con esta función la capa crea los buffers (tensores) adecuados para el soporte de la capa.
- Función *initialize*. En esta función se inicializan los *buffers* (tensores) de la capa.
- Función *forward*. En esta función se realiza el proceso de *forward* (inferencia) de la capa.
- Función *backpropagation*. Con esta función se calculan los gradientes de la capa para su posterior aplicación sobre los pesos.
- Función *propagate_error*. Con esta función se calcula el error en la entrada de la capa a partir del error en la salida de la función.

Las últimas tres funciones enumeradas anteriormente sirven para realizar el proceso de entrenamiento de la capa una vez insertado en una red neuronal.

Otro aspecto de HELENNA es la gestión de memoria y la abstracción de la ubicación de los *buffers* de memoria que se utilizan en un proceso de entrenamiento o inferencia. De hecho, los tensores (implementados con *buffers*) se crean de forma dinámica en HELENNA y se ubican en el dispositivo objetivo seleccionado (CPU, GPU, FPGA, ...). El usuario no es consciente en el momento de la instanciación y en el uso de los *buffers* de donde están ubicados. HELENNA realiza internamente las transferencias necesarias para una correcta utilización de los *buffers*/tensores. Cada tensor creado en HELENNA se identifica con un identificador único y cada acción sobre el tensor utiliza exclusivamente el identificador.

HELENNA permite el entrenamiento en CPU y en GPU con diferentes optimizaciones, principalmente en el soporte de bibliotecas de cálculo matricial. En concreto, HELENNA permite los siguientes dispositivos:

- CPU. Con este dispositivo se utiliza la CPU, todos sus núcleos, en el entrenamiento y/o inferencia. No se utiliza ninguna biblioteca de cálculo matricial. Este dispositivo es el básico.
- MKL [22]. Con este dispositivo se utiliza la CPU junto a la librería de cálculo matricial MKL de Intel. Esta librería funciona exclusivamente con procesadores Intel. Con este dispositivo se obtiene el máximo rendimiento utilizando la CPU.
- Cublas [7]. Con este dispositivo se utiliza la GPU de NVIDIA. En concreto, se utiliza la librería CuBLAS de NVIDIA para el cálculo matricial.
- CuDNN [9]. Recientemente, NVIDIA ofrece CuDNN como una librería para el entrenamiento e inferencia de redes neuronales. Esta librería funciona sobre las GPU de NVIDIA. Con este dispositivo se utiliza dicha librería, mejorando significativamente las prestaciones.
- OpenCL-FPGA. Con este dispositivo se utiliza la librería OpenCL en sistemas basados en FPGA para el proceso de inferencia. En concreto, este dispositivo permite utilizar módulos generados para FPGA específicos como convoluciones y funciones de activación.

Actualmente, HELENNA soporta una variedad de capas de redes neuronales y funciones de activación. A continuación se listan las capas soportadas, así como algunos detalles de cada una de ellas:

En HELENNA se soportan diferentes formatos de bases de datos así como bases de datos muy utilizadas en el contexto de las redes neuronales. Estas son:

Capa	Parámetros	Descripción
Fully_connected	Número de parámetros	Capa densa, también conocida como *MultiLayer Perceptron* (MLP)
Convolution	Tamaño (KH, KW) de filtro, Tamaño (SH, SW) de *stride*, Tamaño (PH, PW) de *padding*	Capa convolucional
Maxpooling	Tamaño (SH, SW) de *stride*, Tamaño (PH, PW) de *padding*	Capa de *maxpooling*
Batch Normalization	Gamma y Beta	Capa de normalización
Dropout	Umbral	Capa de normalización
ReLU,		Función de activación
Sigmoide		Función de activación
Softmax		Función de activación

Tabla 3.1: Capas suportadas por HELENNA

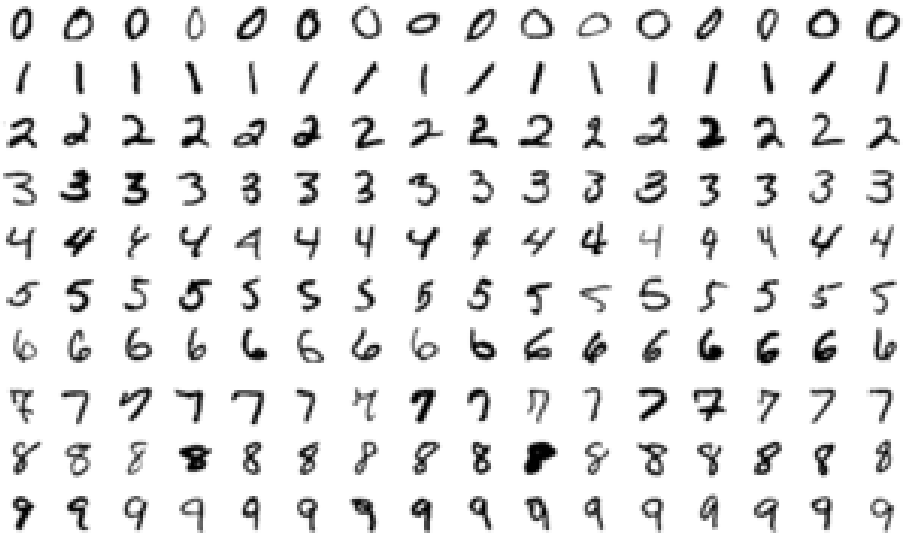


Figura 3.15: Ejemplo de las imágenes que componen el *dataset* MNIST, mostrándose las distintas clases que lo componen, números del 0 al 9

- MNIST (*Modified National Institute of Standards and Technology*): Base de datos conformada por números escritos a mano del 1 al 9, está compuesta por 60.000 imágenes de entrenamiento y 10.000 imágenes de *test* (fig 3.15).
- CIFAR-10 (*Canadian Institute for Advanced Research 10 classes*) [36]: Base de datos conformada por 60.000 imágenes a color de 32x32 píxeles de 10 clases distintas, entre estas imágenes 50.000 son de entrenamiento y 10.000 de *test* (fig 3.16).
- CIFAR-100 (*Canadian Institute for Advanced Research 100 classes*) [5]: Base de datos conformada por 600 imágenes a color de 32x32 píxeles de 100 clases distintas, entre estas imágenes 500 son de entrenamiento y 100 de *test*.
- Imagenet [20]:Base de datos compuesta por más de 14 millones de imágenes y más de 20.000 categorías (fig 3.17).

A partir de las capas de redes neuronales y funciones de activación se puede crear una gran cantidad de redes neuronales. En HELENNA existe una gran variedad, usándose

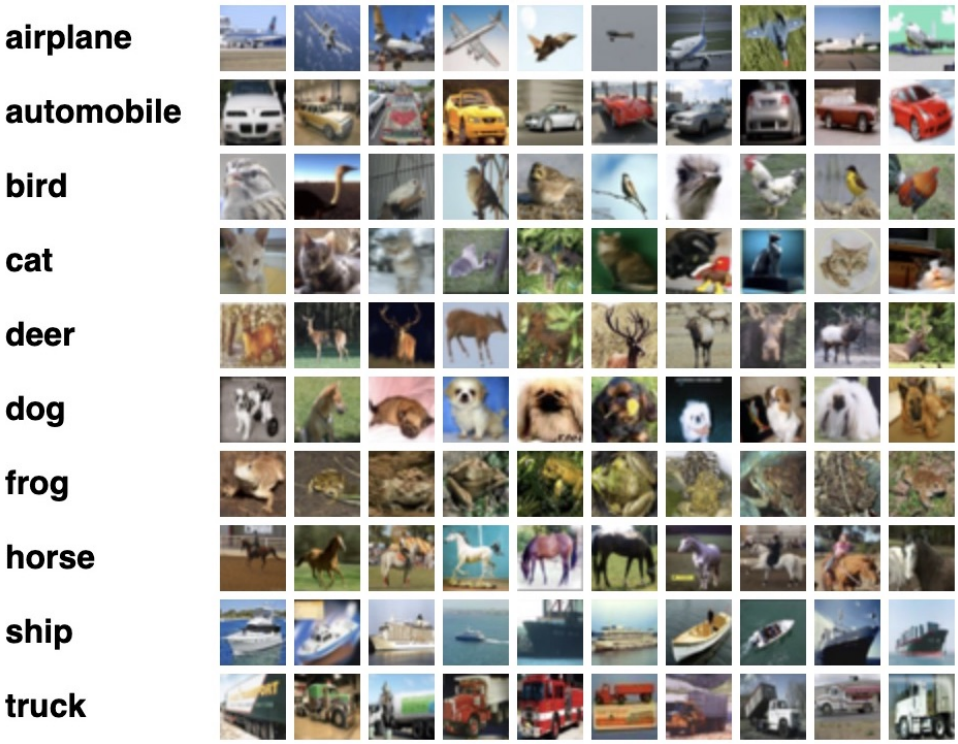


Figura 3.16: Ejemplo de imágenes de las 10 clases del *dataset* CIFAR-10

Figura 3.17: *Collage* de imágenes de distintas categorías del *dataset* Imagenet.

diferentes redes dependiendo del *dataset* utilizado. Algunas de las redes utilizadas son redes conocidas como vgg16 o resnet (Figura A.4 y A.5) mientras que otras son creadas específicamente para su uso en HELENNA. Un ejemplo de esto podría ser la denominada vgg1 (Figura A.1), una red basada en vgg16 (Figura A.2 y A.3) que usa una capa *fully connected* como entrada, seguida de dos capas convencionales con función de activación ReLU cuya salida pasaría por una capa *maxpooling*. Finalmente tendríamos una capa *fully connected* como capa de salida, en el apéndice A podemos observar qué capas y funciones de activación conforman cada una de estas redes.

En este trabajo de fin de grado se han implementado dos nuevas opciones de ejecución llamadas -nccl y -cuda-aware para indicar el uso de estas primitivas de comunicación con GPUDirect dentro de HELENNA, ejecutándose NCCL al introducir -nccl y MPI CUDA-Aware al introducir -cuda-aware. La versión con MPI se ejecuta si no se introduce ninguna de estas opciones.

Por último, HELENNA permite el entrenamiento distribuido tanto en CPU como en GPU. Diferentes estrategias de entrenamiento distribuido están soportadas en HELENNA. Entre ellas están las ya mencionadas en el apartado de entrenamiento distribuido, *parameter server*, *syncronous training* y *asynchronous traing* que cuenta con la opción "mpi-avg" para indicar cada cuantos *batch* se realiza la sincronización. Todas estas opciones están implementadas para su uso en CPU. Sin embargo, para GPU está implementado únicamente el entrenamiento asíncrono, pudiendo reproducir mediante este el comportamiento de un entrenamiento síncrono si a un parámetro "mpi-avg" recibe como valor un 1, indicando así que se sincroniza cada *batch*.

CAPÍTULO 4
Optimización de primitivas de comunicación

Existen distintos factores a analizar a la hora de realizar las comunicaciones entre varios dispositivos. En este capítulo trataremos algunos de estos factores como son la librería que implementa las primitivas de comunicación, el tipo de memoria utilizado o el número de procesos.

4.1 Librería

El estándar a la hora de realizar cualquier tipo de comunicación entre varios dispositivos en un *cluster* es MPI, que nos permite comunicar los procesos mediante el paso de mensaje con operaciones colectivas o punto a punto.

En principio, este estándar es usado para cualquier tipo de comunicación. El caso que nos ocupa es el de la comunicación entre GPU y no entre CPU, ya que las GPU nos permiten acelerar los cálculos necesarios en el entrenamiento de las redes neuronales.

En principio, este estándar permite comunicar distintos dispositivos CPU como GPU. Sin embargo, presenta ciertas particularidades al realizar la sincronización entre GPU que no están presentes en una comunicación entre CPU. A la hora de realizar una comunicación de GPU a GPU o de GPU a la red de interconexión Infiniband, es necesario realizar copias de memoria de los *buffers* desde la CPU a la GPU y viceversa, por lo que al tiempo necesario para completar la comunicación mediante MPI se le sumará el sobrecoste del tiempo empleado en realizar las copias de memoria.

A la hora de solucionar este problema se necesita de alguna alternativa a la comunicación estándar de MPI. Esta debe permitir realizar una comunicación entre GPU sin tener que recurrir a copias de memoria. La solución la podemos encontrar en GPUDirect, una tecnología que nos permite eliminar estas copias de memoria realizando las transferencias directamente entre distintas GPU o desde una GPU a un dispositivo de interconexión Infiniband. Se han encontrado dos alternativas a la hora de realizar una implementación de GPUDirect.

Estas son MPI CUDA-Aware y NCCL. La primera es una versión que nos permite combinar MPI con CUDA permitiendo eliminar la necesidad de usar *cudaMemcpy()* para realizar copias de la memoria entre CPU-GPU o GPU-CPU cada vez que se realiza una comunicación, ya que al combinar CUDA y MPI permite a MPI trabajar con punteros a zona de memoria de GPU tal y como permite CUDA. Esto también lo tolera NCCL, que permite realizar transferencias de GPU a GPU sin requerir de copias de memoria. Además de esto, ambas alternativas hacen uso de GPUDirect P2P y GPUDirect RDMA

para realizar las comunicaciones directamente entre GPU, ya que sin el uso de GPUDirect las copias de memoria solo desaparecerían explícitamente no implícitamente, únicamente se eliminaría la necesidad de llamar a cudaMemcpy().

Las diferencias entre un programa que utilice MPI o MPI CUDA-aware la podemos apreciar en los siguientes fragmentos de código (Código MPI 4.1 y Código MPI CUDA-aware 4.2). Respecto al fragmento de código de MPI, primeramente se realiza una copia de memoria desde un *buffer* en *device* (GPU) a un *buffer* en *host* (CPU) para así, conocer el valor del *buffer* de GPU a la hora de realizar las comunicaciones.

Para indicar entre que dispositivos se va a realizar la copia de memoria (*host-host*, *host-device*, *device-device* o *device-host*) se utiliza el sufijo de la llamada *cudaMemcpy()*. Este parámetro tiene la siguiente estructura: cudaMemcpyXToY donde X es el dispositivo emisor de la copia de memoria e Y el receptor, indicándose en cada caso si esta copia se realizará desde o hacia un *device* o *host*. Un ejemplo de los nombres que puede tomar la llamada y su significado seria que para realizar una copia de memoria de *device* a *host* se utilizaría cudaMemcpyDeviceToHost y si fuera de *host* a *device* cudaMemcpyHostToDevice.

Tras realizar la copia de memoria, el valor del *buffer* se comunica mediante las primitivas de comunicación de MPI de un *host* a otro. Una vez que se ha recibido este valor, se realiza una copia de memoria de *host* a *device* para que así se complete la comunicación entre las GPU.

Este proceso para comunicar varias GPU se acorta para MPI CUDA-aware, donde *Send* y *Recv* directamente transfieren el *buffer* de una GPU a otra sin la necesidad de invocar a *cudaMemcpy()*. Esto se debe a que esta variante es capaz de utilizar punteros a la zona de memoria de las GPU, cosa que no era posible para la implementación en MPI. Además, utiliza GPUDirect para que esta transferencia se realice directamente sin copias de memoria.

```
1 //MPI rank 0
2 cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
3 MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
4
5 //MPI rank 1
6 MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
7 cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

Listing 4.1: Ejemplo de comunicación punto a punto entre GPU con MPI

```
1 //MPI rank 0
2 MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
3
4 //MPI rank n-1
5 MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

Listing 4.2: Ejemplo de comunicación punto a punto entre GPU con MPI CUDA-aware

La otra alternativa es NCCL, que implementa primitivas de comunicación diseñadas específicamente para trabajar eficientemente con GPU. Para trabajar con este tipo de primitivas hay que cambiar ciertos aspectos en la llamada a las primitivas de comunicación. El primero es la interfaz de las primitivas de comunicación. Este cambio lo podemos observar en el siguiente fragmento de código 4.3. En este se presenta una comparativa del código de NCCL y MPI a la hora de realizar la primitiva de comunicación *Bcast*. En pri-

mer lugar se muestra la implementación en NCCL y posteriormente su implementación en MPI.

Podemos observar entre la invocación de una misma primitiva de comunicación para MPI o NCCL deberemos cambiar tanto el nombre de la función como algunos parámetros. Uno de estos parámetros es el tipo de datos, por ejemplo para MPI el tipo de datos *float* seria *MPI_FLOAT* y para NCCL *ncclFloat* (esto puede ser aplicado para todos los tipos de datos). Otro parámetro a cambiar sería el comunicador, este define un conjunto de procesos que pueden intercambiar mensajes, asignándole a cada uno de estos procesos un identificador único. Este comunicador será distinto en MPI y NCCL. Además en NCCL habrá que añadir como último parámetro un *stream* o lo que es lo mismo, una secuencia de operaciones que se ejecutan en orden. Las llamadas NCCL están asociadas a este *stream*, que permite ejecutar de forma asíncrona una operación colectiva en un *device* CUDA una vez haya sido encolada correctamente al *stream*, en caso contrario se devolvería error.

Otro aspecto que varía en NCCL respecto a MPI es la inicialización y creación de los comunicadores, mientras que en MPI únicamente hay que inicializar el propio MPI y obtener la información del número de procesos y del proceso local (Código 4.4), en NCCL dependiendo del número de procesos o del número de dispositivos asociados a cada proceso se inicializará de una manera u otra creando su correspondiente comunicador. Algunos ejemplos de diferentes tipos de inicialización serían una en el que hubiera un proceso con un único hilo en el que se ejecutaran múltiples *devices*, una en el que haya múltiples procesos y múltiples *devices* por proceso y la opción que utilizaremos nosotros, un *device* para cada proceso (Código 4.5).

En la opción utilizada, a la hora de efectuar la inicialización primeramente obtendremos información sobre el proceso a través de MPI, tras esto crearemos una ID única para enviar a cada proceso repartiendo este ID con *Bcast* para que todos los procesos la tengan y por último crearemos el comunicador a utilizar, pudiendo ya realizar la llamada a la primitiva de comunicación.

También existen diferencias a la hora de finalizar la ejecución un programa. En MPI únicamente deberemos llamar a *MPI_Finalize()* para terminar la ejecución del entorno MPI, mientras que en NCCL habrá que llamar a *MPI_Finalize()* si se ha inicializado un entorno de ejecución MPI y *ncclCommDestroy(comm)* para liberar los recursos del comunicador.

```
ncclResult_t
ncclBcast(void* buff, size_t count, ncclDataType_t datatype,
int root, ncclComm_t comm, cudaStream_t stream)

int
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm)
```

Listing 4.3: Comparación interfaz de una llamada Bcast en nccl con su versión en MPI

```
MPICHECK(MPI_Init(&argc, &argv));
MPICHECK(MPI_Comm_rank(MPI_COMM_WORLD, &myRank));
MPICHECK(MPI_Comm_size(MPI_COMM_WORLD, &nRanks));
```

Listing 4.4: Inicialización en MPI

```

//Obtener informacion de MPI sobre el proceso
```

```
int myRank, nRanks;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &nRanks);

//Crear una, id unica para enviar a los demas procesos
ncclUniqueId id;
if (myRank == 0) ncclGetUniqueId(&id);
MPI_Bcast(&id, sizeof(id), MPI_BYTE, 0, MPI_COMM_WORLD);

//Crear el comunicador
ncclComm_t comm;
ncclCommInitRank(&comm, nRanks, id, myRank);
} */
```

Listing 4.5: Inicialización en NCCL

4.2 Reserva de memoria

Otro factor que tendremos en cuenta a la hora de analizar el rendimiento de las distintas primitivas de comunicación es la reserva de memoria, ya que el como se realiza esta puede llegar a afectar en las prestaciones obtenidas.

La reserva de memoria más típica para una GPU es *cudaMalloc()*, que permite asignar memoria fija mediante paginación. Esto quiere decir que a la hora de realizar transferencias entre distintos dispositivos debe accederse primeramente a la zona de paginación, desde la cual accedemos a la zona de la memoria fija en el *host* para transferir los datos contenidos en esta zona de memoria de la GPU. Además de esta forma de reservar la memoria existen otras dos que incluiremos en nuestras pruebas.

La primera es *cudaMallocHost()*, en la que se elimina la necesidad de ubicar la memoria fija mediante paginación, ya que toda la memoria se ubica en la zona de memoria fija. Para usar este tipo de memoria cambiaremos la llamada utilizada de *cudaMalloc()* a *cudaMallocHost()*. En este caso no hay que hacer cambios en los parámetros introducidos, el único es a la hora de liberar la memoria asignada sustituyéndose *cudaFree()* por *cudaFreeHost()*.

El último tipo de memoria a analizar es *cudaMallocManaged()*. Este es parecido a *cudaMalloc()* a la hora de asignar la memoria mediante paginación, con la diferencia de que se crea un espacio de memoria común para GPU y CPU, cosa que no ocurre en *cudaMalloc()* donde cada uno tiene su propio espacio de memoria. Estas llamadas para reservar la memoria presentan ciertas diferencias que podemos apreciar en el fragmento de código 4.6.

Respecto a la llamada *cudaMallocManaged()*, presenta una pequeña diferencia respecto a *cudaMalloc()* y *cudaMallocHost()*. Los dos primeros parámetros, al igual que *cudaMalloc()* y *cudaMallocHost()*, se corresponden al puntero al *buffer* y el tamaño de este *buffer*. En *cudaMallocManaged()* se le añade un tercer parámetro que corresponde a una *flag* que puede tomar dos valores, *cudaMemAttachHost* para que el espacio de memoria asignado pueda ser accedido por una GPU cuyo máximo número de hilos por bloques sea 0 o el valor escogido en nuestra implementación o *cudaMemAttachGlobal* para que este espacio de memoria sea accesible por cualquier *stream* o dispositivo. Respecto a liberar memoria, tal y como podemos ver en el fragmento de código 4.7, se realiza de igual manera para *cudaMalloc()* y *cudaMallocManaged()*, realizando una llamada a *cudaFree()*.

```

```

```
__host____device__cudaError_t cudaMalloc ( void** devPtr, size_t
    size )

__host__cudaError_t cudaMallocHost ( void** ptr, size_t size )

__host__cudaError_t cudaMallocManaged ( void** devPtr, size_t size,
     unsigned int flags = cudaMemAttachGlobal
```

Listing 4.6: Distintas formas de asignar memoria en un buffer en CUDA

```

//Liberar memoria asiganda en cudaMalloc o cudaMallocManaged
__host____device__cudaError_t cudaFree ( void* devPtr )
//Liberar memoria asiganda en cudaMallocHost
__host__cudaError_t cudaFreeHost ( void* ptr )
```

Listing 4.7: Distintas formas de liberar memoria de un textitbuffer en CUDA

4.3 Pruebas ejecutadas

Para comprobar los efectos que tienen la asignación de memoria, la biblioteca que implementa las primitivas de comunicación y el número de procesos que están ejecutándose, se ha realizado una serie de pruebas para comprobar el efecto de estos factores en el ancho de banda obtenido para distintas primitivas de comunicación.

Para conseguirlas se ha realizado un *benchmark* mediante el cual podemos probar las distintas combinaciones para distintas primitivas de comunicación como pueden ser *Allreduce, Reduce, Bcast, Send* o *Recv*.

Para cada una de estas pruebas se realizarán ejecuciones en las que se intercambiarán *buffers* mediante las distintas primitivas de comunicación a probar. El tamaño de estos *buffers* va desde 1 *float* a 268435456 *floats* lo que equivale a 1 GB (GigaByte). Cada uno de estos intercambios de *buffer* sé realizará un total de 100 veces, obteniéndose el tiempo medio para ejecutar la comunicación en estas 100 iteraciones.

Se han realizado 3 gráficas para cada una de las primitivas de comunicación en las que observamos su comportamiento para 2, 4 y 8 procesos, salvo para *Send* y *Recv* para las cuales únicamente se han realizado pruebas con 2 procesos. Para cada una de estas gráficas el eje vertical representa el ancho de banda obtenido medido en GigaBits por segundo (Gb/s) que ha sido obtenido mediante la siguiente fórmula:

$$\frac{(\frac{tamanyo\_buffer*4*8}{tiempo\_ej})}{10^9} \tag{4.1}$$

En esta fórmula primeramente se calcula el tamaño del *buffer* en bytes multiplicándolo por 4, ya que un *float* equivale a 4 bytes y pasando estos 4 bytes a bit al multiplicarlos por 8. Tras esto se divide por el tiempo de ejecución en segundos para así obtener el ancho de banda en bits por segundo. Por último se divide esto por 10^9 para obtener así el resultado en GigaBits por segundo. El tiempo de ejecución usado para calcular el ancho de banda se corresponde con un tiempo promedio que viene dado por la suma del tiempo de cada uno de los procesos para realizar las 100 iteraciones dividido por el número de procesos multiplicado por el número de iteraciones. La fórmula es:

$$tiempo\_ej = \frac{tiempoTotal}{numProc * numIter} \tag{4.2}$$

El eje horizontal se corresponde con el tamaño del mensaje en *floats*. Un último aspecto a citar respecto a las gráficas es que las líneas verdes representan el ancho de banda de las combinaciones con NCCL, las azules MPI CUDA-aware y las de color naranja MPI. Seguidamente, procedemos a analizar el comportamiento obtenido para cada una de las primitivas de comunicación.

4.3.1. Análisis Allreduce

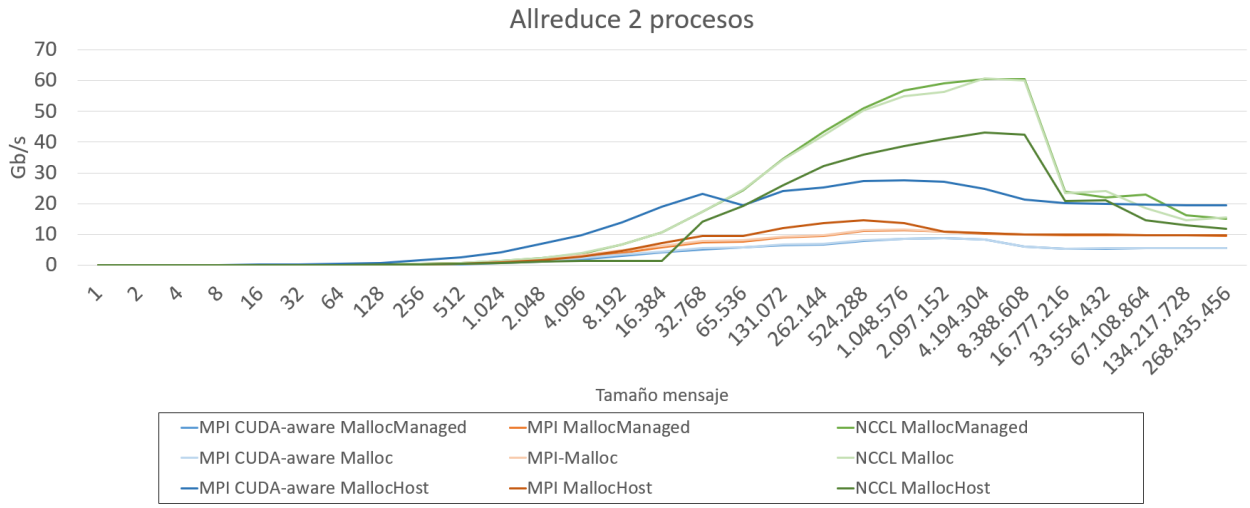


Figura 4.1: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Allreduce* ejecutado con 2 procesos

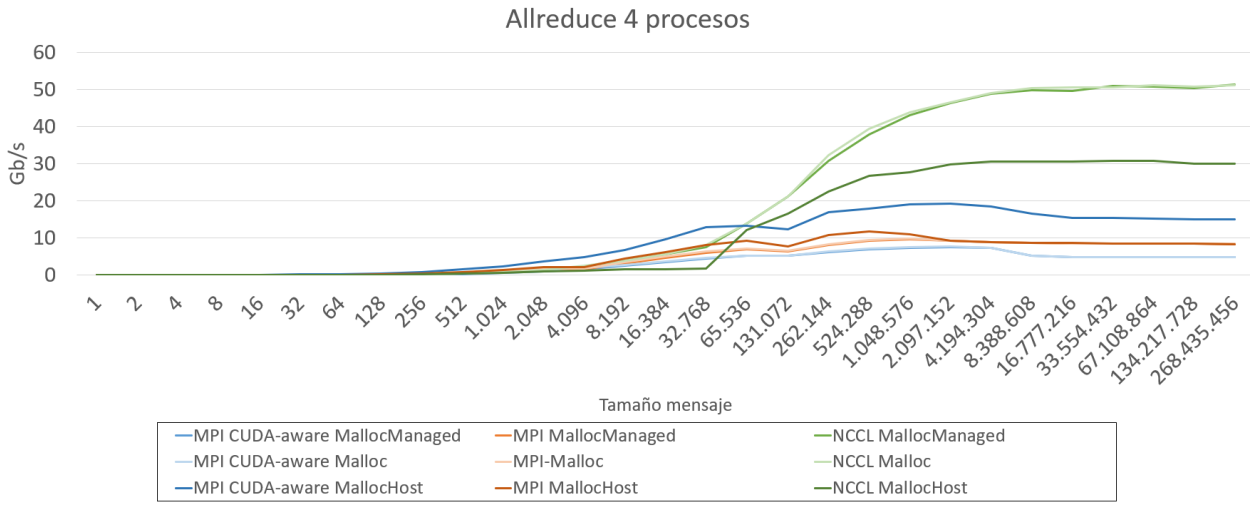


Figura 4.2: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Allreduce* ejecutado con 4 procesos

En la gráfica que representa las combinaciones para 2 procesos (Figura 4.1) podemos apreciar que las combinaciones que utilizan la librería NCCL son aquellas que obtienen un mayor ancho de banda entre los tamaños de *buffer* entre 32768 y 16777216 *floats*. Entre estas combinaciones, las versiones que reservan la memoria mediante *Malloc()* o *MallocManaged()* son las que obtienen un mayor ancho de banda, mientras que NCCL

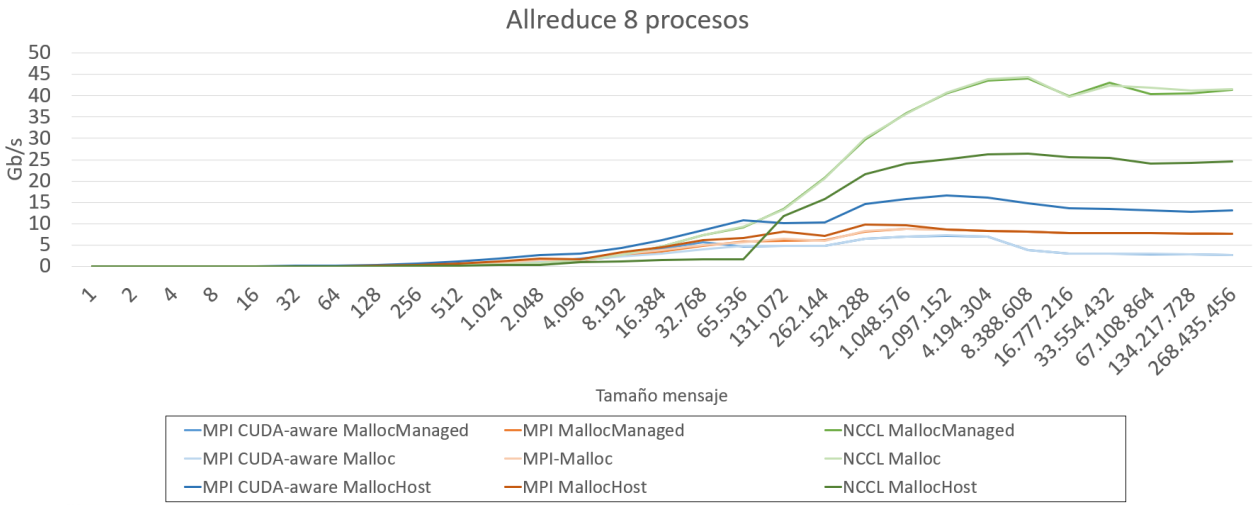


Figura 4.3: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Allreduce* ejecutado con 7 procesos

presenta un menor rendimiento con *MallocHost()*. Tras NCCL tendríamos MPI CUDA-aware con *MallocHost()* como la combinación que obtiene un mayor ancho de banda a partir del tamaño de 32768 *floats*. Para tamaños menores a 32768, MPI CUDA-aware con *MallocHost()* es la combinación que presenta un mayor ancho de banda. Otro aspecto a resaltar es que las prestaciones de NCCL experimentan un decrecimiento muy grande para tamaños superiores a 16777216 *floats*. Este decrecimiento es tan grande que causa que para estos tamaños las combinaciones que utilizan NCCL tengan un ancho de banda similar o incluso menor que MPI CUDA-aware con *MallocHost()* para los tamaños más grandes. Estos comportamientos se deben a las características de las librerías de NCCL y MPI CUDA-aware.

MPI CUDA-aware dispone de diseños más avanzados de operaciones punto a punto que permiten soluciones alternativas eficientes para ciertos cuellos de botella, como la lectura de GPUDirect a través de los *sockets*. Estos diseños optimizados no pueden ser utilizados por librerías especializadas como NCCL que están diseñadas para una serie de tareas específicas. Esto causa que para tamaños de mensaje pequeños o medianos tendremos unas mejores prestaciones con MPI CUDA-aware [41].

Pese a esto, NCCL ofrece unas mejores prestaciones para tamaños grandes y muy grandes de mensajes. Esto se debe a que NCCL proporciona una mayor escalabilidad para este tipo de tamaños al estar diseñado específicamente para realizar operaciones colectivas entre GPU, por lo que las operaciones colectivas de NCCL permiten realizar la comunicación más eficientemente que sus homónimas en MPI CUDA-aware para este tipo de tamaños.

Eso si, aunque esta escalabilidad sea mayor en NCCL, para 2 procesos (que no para 4 y 8 como veremos ahora cuando los comentemos) hemos podido observar que existe un tamaño (que en este caso es 16777216) a partir del cual las prestaciones decrecen significativamente. Esto se debe a que NCCL tal y como dice su nombre, está orientado a operaciones colectivas, no a operaciones punto a punto por lo que para tamaños muy elevados no puede mantener su escalabilidad. Podemos observar que para 4 (Figura 4.2) y 8 (Figura 4.3) procesos este decrecimiento de las prestaciones no existe al realizarse para este número operaciones colectivas y no punto a punto tal y como ocurre con 2 procesos.

Por lo tanto, podemos concluir que para tamaños de mensajes (en nuestro caso son *floats*) pequeños y medianos será más adecuado utilizar MPI CUDA-aware, mientras que para tamaños de mensaje grandes o muy grandes será más conveniente el uso de NCCL

salvo que se comuniquen 2 procesos, caso en el que si el tamaño es muy elevado puede llegar a ser más conveniente el uso de MPI CUDA-aware.

Otro comportamiento notable es el que tiene *cudaMallocHost()*. Podemos apreciar que con MPI CUDA-aware permite mejorar de forma significativa su funcionamiento. En MPI también permite mejorar en cierta medida el ancho de banda obtenido, aunque a partir de cierto punto su rendimiento decrece y se iguala al de las otras dos reservas de memoria. Para NCCL, esta forma de reservar memoria no permite mejorar las prestaciones obtenidas, sino que obtiene un ancho de banda menor al de las otras dos reservas de memoria.

Estos comportamientos tienen varias causas. Para comprenderlas, primero debemos tener en cuenta cuál es la naturaleza de *MallocHost()*. Como ya se ha descrito a lo largo de este trabajo, este tipo de reserva de memoria permite reservar directamente memoria fija lo que nos permite que cada vez que queramos acceder a una dirección de memoria no tengamos que pasar por el proceso de paginación, sino que podemos acceder y realizar las transferencias del contenido de una dirección de memoria a otra GPU directamente.

Esto nos permite aumentar el ancho de banda de nuestro *host* (CPU) pudiendo acceder directamente a la memoria de la GPU. Por este motivo, encontramos anchos de banda mayores en MPI y MPI CUDA-Aware, siendo más notable este comportamiento en MPI CUDA-Aware, ya que aquí no es necesario realizar copias de memoria para realizar la transferencia, por lo que se optimizan aún más las comunicaciones. Sin embargo, este tipo de reserva de memoria presenta ciertos inconvenientes. No debe asignarse demasiada cantidad, ya que la memoria física utilizada para la paginación se reduciría, lo que conllevaría a un empeoramiento en las prestaciones. Esto lo podemos ver en todas las variantes que hacen uso de *MallocHost()*, pudiendo apreciarse como comienza a disminuir su ancho de banda para los tamaños de *buffer* más grandes.

Respecto a NCCL, se puede apreciar que en este caso, al contrario que MPI y MPI CUDA-aware, *MallocHost()* no presenta un mayor ancho de banda para los tamaños mostrados respecto a *Malloc()* y *MallocManaged()*. Esto puede deberse a que al ser NCCL una librería tan especializada, esta se encuentre optimizada a la hora de utilizar memoria con paginación. Siendo este comportamiento más significativo en una primitiva de comunicación como *Allreduce* en la que todos los nodos se comunican entre sí.

Otro comportamiento que podemos apreciar en estas gráficas es que el ancho de banda obtenido para *Malloc()* y *MallocManaged()* es prácticamente idéntico, por lo que no podemos extraer conclusiones sobre si uno es mejor que otro, ya que las diferencias entre ambos es mínima. Esto se debe a que ambos utilizan la paginación a la hora de reservar memoria por lo que en este apartado no presentarían diferencias. Además, respecto al punto que los diferencia que es el de la unificación de memoria, en nuestro programa no se ve afectado, puesto que no necesita de las características que proporciona esta particularidad para obtener una mejora de rendimiento como la que podría proporcionar la posibilidad que da este tipo de memoria de eliminar ciertos punteros duplicados.

En cuanto a una comparativa entre los resultados para 2, 4 y 8 procesos podemos observar que el comportamiento presentado por cada una de las combinaciones es similar, no existiendo una gran diferencia respecto a este apartado. La mayor diferencia la podríamos encontrar en el ya mencionado bajón en el ancho de banda para tamaños mayores a 1677216, que únicamente se produce para 2 procesos. Mientras que para 4 y 8 procesos el ancho de banda solamente deja de aumentar al ritmo que lo hacía para tamaños menores, estancándose su progreso.

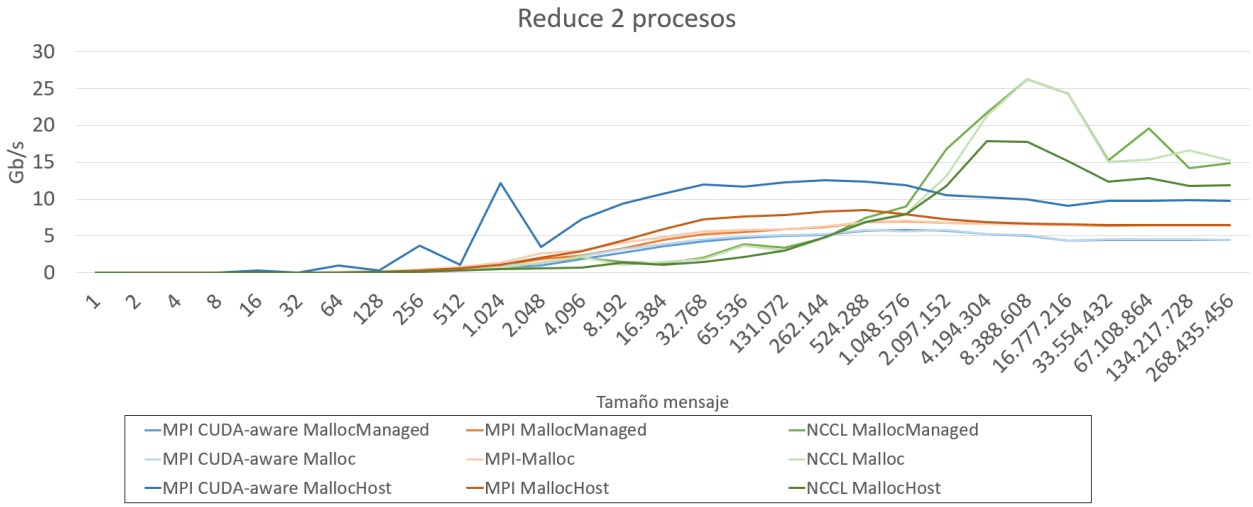


Figura 4.4: Gráfica comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Reduce* ejecutado con 2 procesos

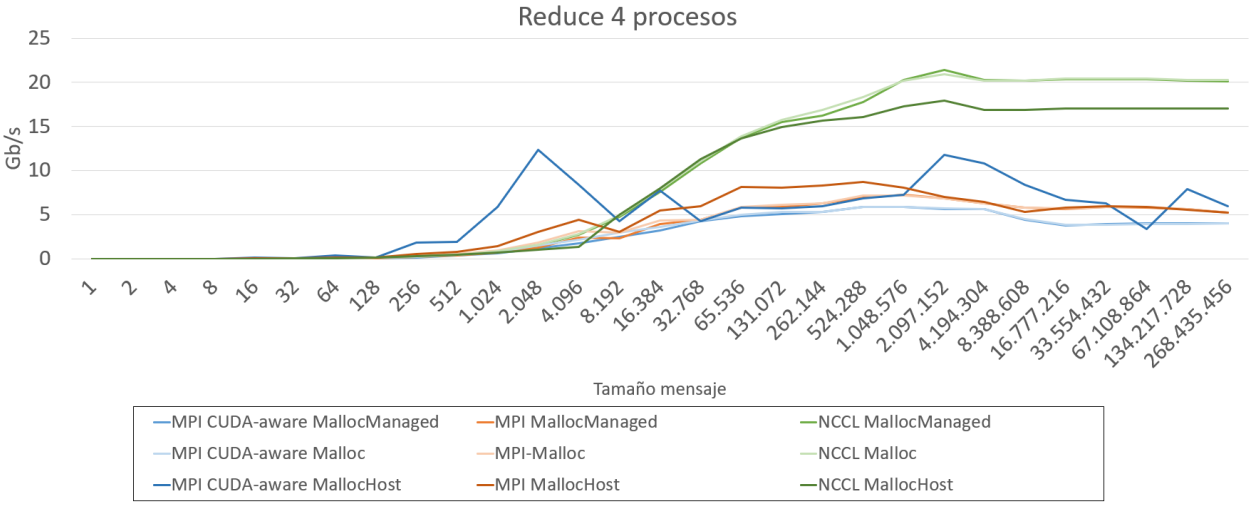


Figura 4.5: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Reduce* ejecutado con 4 procesos

4.3.2. Análisis *Reduce*

Estas gráficas tanto para 2 (Figura 4.4), 4 (Figura 4.5) como 8 procesos (Figura 4.6) presentan comportamientos similares a los presentes en las gráficas que representan los comportamientos de la primitiva *Allreduce*. Algunos de estos comportamientos comunes son: unas mejores prestaciones de MPI CUDA-Aware respecto a las combinaciones con NCCL hasta cierto tamaño de *buffer*, la similitud entre *MallocManaged()* y *Malloc()*, el comportamiento de *MallocHost*, el cual presenta un mayor ancho de banda para tamaños de mensaje pequeños y que empieza a decrecer para tamaños de mensaje más grandes o que NCCL satura bruscamente para 2 procesos y experimenta una leve bajada y una estabilización para 4 y 8 procesos.

Aunque existen muchas similitudes entre el comportamiento de *Allreduce* y *Reduce* también existen ciertas diferencias. La primera de ellas se encuentra en el comportamiento de MPI CUDA-aware *MallocHost()*. Es cierto que esta combinación sigue siendo la mejor para los tamaños de *buffer* pequeños y medianos o la segunda mejor tras las combinaciones de NCCL en el resto de tamaños de *buffer* (exceptuando ciertos tramos en la

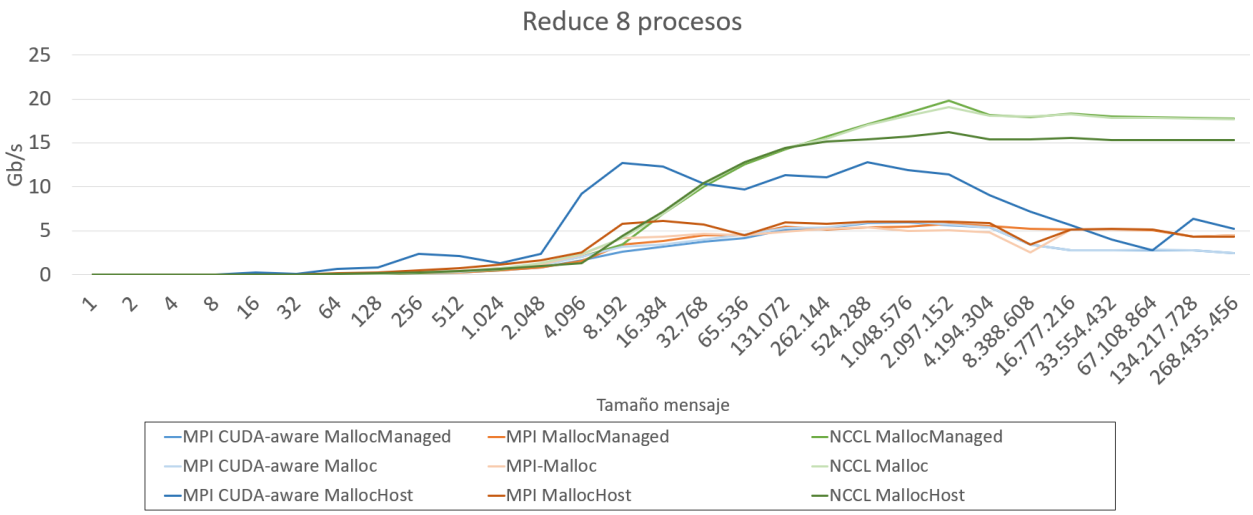


Figura 4.6: Gráfica comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Reduce* ejecutado con 7 procesos

ejecución con 4 procesos y cierto punto para 8 procesos). Sin embargo, mientras que para *Allreduce* su comportamiento representaba una curva medianamente uniforme, aquí podemos observar que se presenta un comportamiento más irregular, al darse distintos picos tanto positivos como negativos para distintos tamaños del *buffer*.

Esta irregularidad aunque está presente para 2,4 y 8 procesos es más notable para 4 y 8 procesos. Aunque esta irregularidad es notable para 4 y 8 procesos, podemos observar como para 4 procesos encontramos picos más pronunciados. Mientras que para 8 procesos existe un comportamiento irregular, con un ancho de banda que no se termina de estabilizar, pero que aumenta o decrece de forma más moderada, a lo largo de la ejecución con 4 procesos podemos observar varios picos en los que varía el ancho de banda de forma significativa. Para 4 y 8 procesos (en un periodo mayor para 4 procesos) incluso ocurre que en ciertos tramos de la ejecución de MPI CUDA-aware con *MallocHost()*, las prestaciones obtenidas son peores que las obtenidas con MPI.

Por lo tanto, podemos concluir que para la primitiva de comunicación *Reduce* el comportamiento obtenido por la primitiva de comunicación *Reduce* es ciertamente irregular, siendo más notorio este comportamiento para 4 y 8 procesos, ya que con 2 procesos a partir de cierto tamaño el ancho de banda obtenido es más o menos estable. Para este tipo de primitivas al contrario de lo que ocurría con la primitiva *Allreduce*, en la que pese a que para tamaños de mensaje grandes se aconseja el uso de NCCL debido a sus mejores prestaciones, tras estas combinaciones MPI CUDA-aware con *MallocHost()* es la mejor opción para los tamaños de *buffer* mostrados (eso si no sería recomendable usar *MallocHost()* con tamaños superiores a los probados, ya que las prestaciones podrían verse resentidas al reservar demasiada memoria fija). Sin embargo, para la primitiva *Reduce* no podríamos afirmar que esta es la segunda mejor opción para estos tamaños probados, ya que dependiendo del tamaño de *buffer* para 4 y 8 procesos se da que incluso MPI obtiene mejores prestaciones.

4.3.3. Análisis *Bcast*

Estas gráficas también presentan ciertos comportamientos comunes con las gráficas que representan el comportamiento de las primitivas *Reduce* y *Allreduce*. Algunos de estos comportamientos son unas mejores prestaciones de MPI CUDA-Aware comparado con NCCL hasta cierto tamaño de *buffer*, la similitud entre *MallocManaged()* y *Malloc()*,

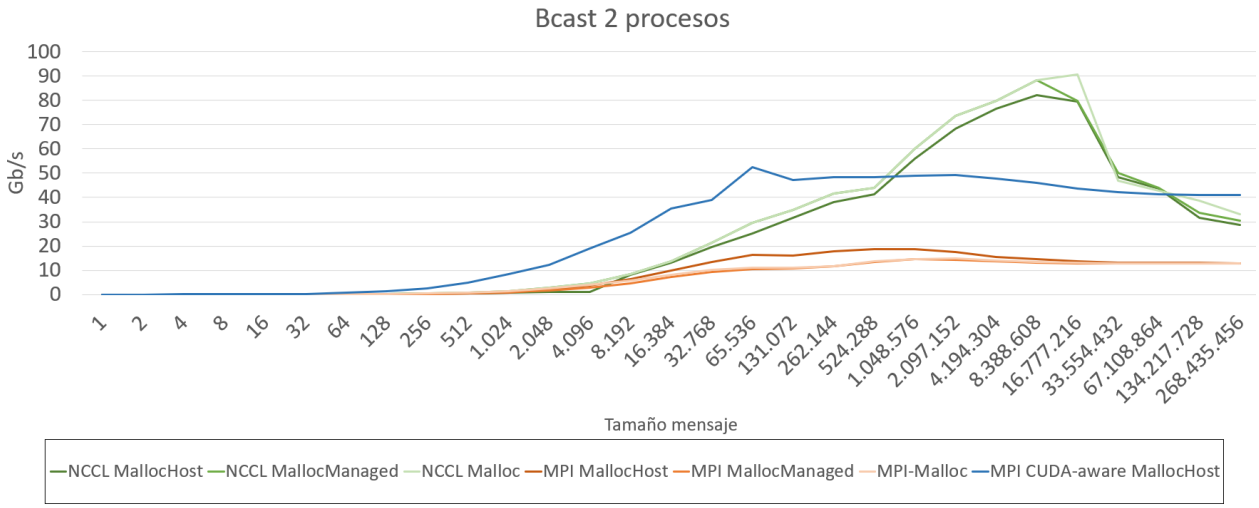


Figura 4.7: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Bcast* ejecutado con 2 procesos

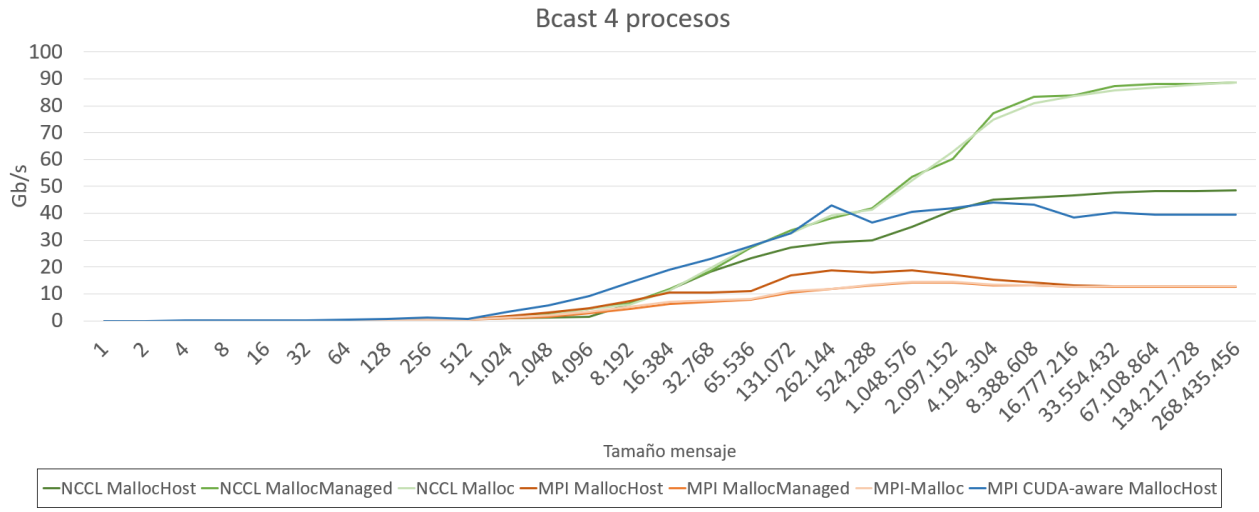


Figura 4.8: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Bcast* ejecutado con 4 procesos

el comportamiento de *MallocHost()* que presenta un mayor ancho de banda para tamaños de mensaje pequeños que empieza a decrecer para tamaños de mensaje más grandes o que NCCL satura bruscamente para 2 procesos y experimenta una leve bajada y una estabilización para 4 y 8 procesos.

Los únicos aspectos que varían respecto a los ofrecidos en *Reduce* pero sobre todo en *Allreduce* (ya que Reduce presentaba ciertas irregularidades) son referentes a la diferencia entre el ancho de banda de MPI CUDA-aware con *MallocHost()* y NCCL con MallocHost(), estas dos combinaciones obtienen anchos de banda más próximos entre sí para 4 y 8 procesos, en comparación al obtenido con otras primitivas de comunicación. Mientras que con 4 y 8 procesos se da este comportamiento, con 2 procesos todas las combinaciones de NCCL obtienen unas prestaciones similares, estas combinaciones con NCCL tal y como hemos comentado experimentan un descenso del ancho de banda a partir de cierto punto para 2 procesos al igual que el resto de primitivas analizadas. Otro aspecto que se puede observar en esta gráfica es que para 2 procesos, con MPI CUDA-aware *MallocHost()* los mayores tamaños de buffer obtienen un mayor ancho de banda que las combinaciones con NCCL.

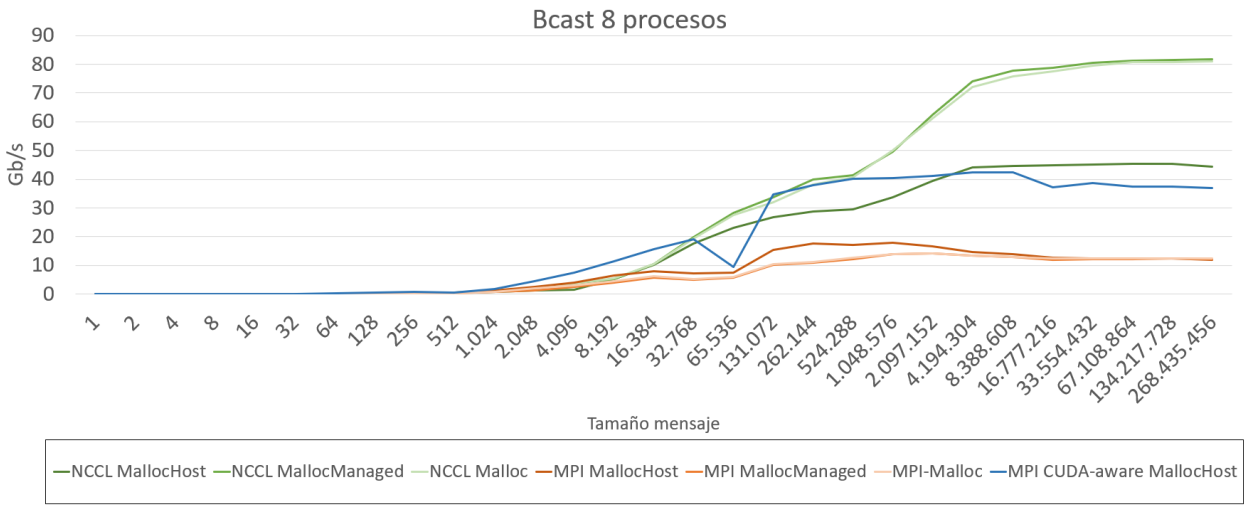


Figura 4.9: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Bcast* ejecutado con 7 procesos

Respecto a las gráficas de la primitiva *Bcast*, en primer lugar hay que puntualizar que no se encuentran entre las combinaciones MPI CUDA-aware con *Malloc()* y *MallocManaged()*. Para estas combinaciones se han encontrado errores de ejecución que no han permitido incluirlas. Los errores son distintos para uno y otro. Para *Malloc()*, cuando reservamos la memoria nos indica un error (Figura 4.10) debido a que a la hora de llamar a la implementación de la función *Bcast* para MPI CUDA-aware accedes a un objeto sobre el que no tenemos permisos. Esto debe referirse al puntero al *buffer* CUDA que le proporcionamos, no pudiendo acceder a esta zona de memoria como le ocurriría a una implementación MPI.

Una demostración de que esta es la causa del error esta en el uso de *MallocManaged()* ya que en este caso, al unificar los espacios de memoria de CPU y GPU, no devuelve error hasta ciertos tamaños, por lo que podemos suponer que el error se encuentra en la imposibilidad de acceder a un *buffer* de GPU. Respecto al error que presenta *MallocManaged()* (Figura 4.11), este completa su ejecución correctamente para tamaños de hasta 524288 *floats*. Sin embargo, a partir de este tamaño nos devuelve un error relacionado con la comprobación de los valores que deberían de tener los *buffer*. Con estos nos referimos a una función implementada en el *benchmark* que comprueba si el valor final de cada uno de los *buffers* es correcto. Si no es correcto, devuelve un error y imprime para que proceso y posición del *buffer* ocurre el error. Este error debe deberse a que, por algún motivo, no se completa la operación de *Bcast* correctamente por lo que los procesos receptores no reciben el *buffer* como deberían recibirlo.

4.3.4. Análisis *Send/Recv*

Respecto a esta primitiva de comunicación, primeramente cabe decir que al igual que *Bcast*, no presenta resultados para la combinación MPI CUDA Aware *Malloc()*, debido a un error similar al comentado para *Bcast*. Este error está relacionado con la falta de permisos para acceder a la dirección de memoria asignada en un *buffer* GPU.

También presenta ciertas similitudes comparada con las primitivas analizadas previamente. Algunos de los comportamientos similares que presenta esta primitiva son la similitud entre los anchos de banda obtenidos para *MallocManaged()* y *Malloc()* o el comportamiento que presenta MPI con unas mejores prestaciones para *MallocHost()* hasta cierto punto en el que se iguala con los otros dos tipos de reserva de memoria.

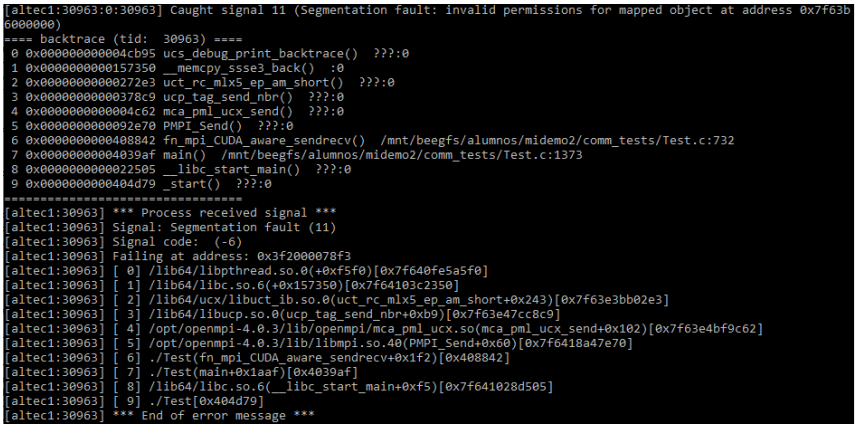


Figura 4.10: Error presente en la ejecución de la primitiva de comunicación *Bcast* al utilizar MPI CUDA-aware con reserva de memoria mediante *cudaMalloc()*

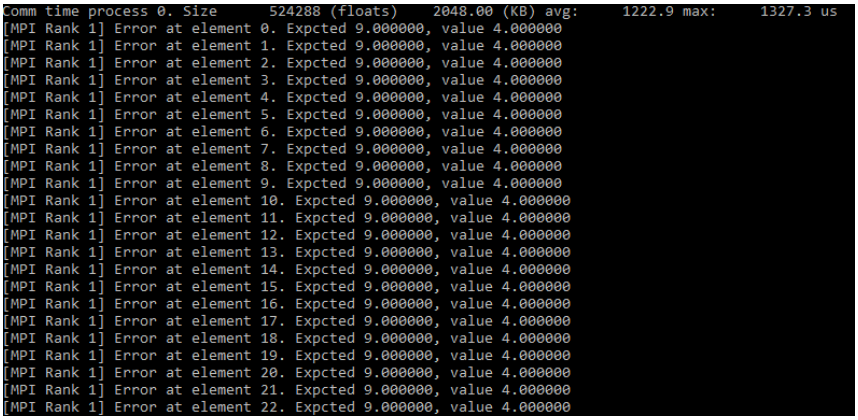


Figura 4.11: Error presente en la ejecución de la primitiva de comunicación *Bcast* al utilizar MPI CUDA-aware con reserva de memoria mediante *cudaMallocManaged()*

En la figura 4.12 podemos observar grandes diferencias respecto al resto de primitivas. En primer lugar, podemos apreciar que aquí NCCL presenta un comportamiento mucho más pobre al mostrado en gráficas anteriores, estas no superan en ningún momento el ancho de banda obtenido por las combinaciones de MPI CUDA-aware y únicamente superan a las variantes con MPI para tamaños de *buffer* mayores a 8338608.

Mientras que las peores prestaciones las muestra NCCL, con MPI CUDA-aware ocurre justo lo contrario, esta variante es la que presenta unas mejores prestaciones. El caso más notable es el de MPI CUDA-AWARE *MallocHost()*, donde se obtienen una prestaciones muy altas hasta cierto punto en el que el ancho de banda cae drásticamente igualándolo con el obtenido con el otro tipo de reserva de memoria disponible para MPI CUDA-aware.

Este caso nos permite observar mejor ciertos comportamientos y características de esta serie de gráficas que hemos estado comentando. Primeramente el mal comportamiento de NCCL es causado por que es una librería especializada en operaciones colectivas que no es óptima para operaciones punto a punto, por lo que en una comunicación como la de *Send* y *Recv*, que hace uso de operaciones punto a punto no obtiene buenos resultados. Al igual que observamos que NCCL presenta en esta gráfica el peor rendimiento entre las primitivas analizadas, en MPI CUDA-AWARE ocurre justo el contrario, ya que en esta variante se realizan más eficientemente las operaciones punto a punto. Esta implementación junto a *MallocHost()* nos permite lograr un ancho de banda muy elevado.

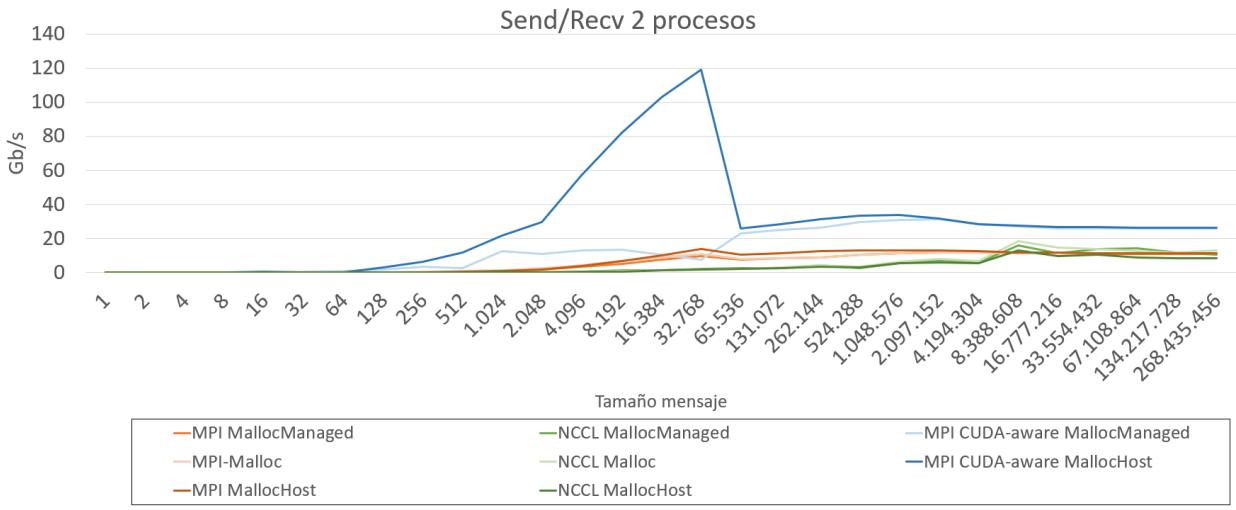


Figura 4.12: Comparativa del ancho de banda (eje vertical) para diferentes primitivas de comunicación y reserva de memoria para *Send/Recv* ejecutado con 2 procesos

4.3.5. Conclusiones

A partir del análisis que hemos realizado sobre cada una de las primitivas de comunicación analizadas podemos sacar una serie de conclusiones:

- No existen grandes diferencias de prestaciones a la hora de reservar la memoria mediante las funciones *Malloc()* o *MallocManaged()*. La única diferencia importante es que permite eliminar el error presente en *Send/Recv* para MPI CUDA-aware.
- En las operaciones de comunicación colectiva para tamaños de mensaje pequeños o medianos Cuda Aware *MallocHost()* será la combinación que presente unas mejores prestaciones y para tamaños grandes NCCL con *Malloc()* o *MallocManged()* indistintamente será la que presente unas mejores prestaciones. En las operaciones punto a punto, tenemos un comportamiento similar pero a su vez diferente ya que para tamaños de mensaje pequeños o medianos MPI Cuda-aware *MallocHost()* sera la que presentará mejores prestaciones tal y como ocurre en las operaciones colectivas. Sin embargo para tamaños de mensaje grandes, sus prestaciones se igualarán con las de MPI Cuda-aware *MallocManaged()* y para tamaños de mensaje muy grandes será mejor MPI Cuda-aware *MallocManaged()*. Estos comportamientos los podemos ver representados en la tabla 4.1 en la que se muestra para distintas pruebas qué tamaño modifica la combinación más óptima para dicha prueba, indicándose que combinación era la mejor para tamaños mayores o menores al tamaño indicado.
- El tipo de reserva de memoria *MallocHost()* permite acelerar las comunicaciones para tamaños pequeños o medianos de mensaje para MPI o sobre todo MPI CUDA-aware.
- NCCL no es adecuado para operaciones punto a punto como *Send/Recv*.
- NCCL con *MallocHost()* obtiene unas peores prestaciones respecto a las combinaciones de NCCL con los otros dos tipos de reserva de memoria.
- NCCL sufre una bajada brusca de las prestaciones a partir de cierto tamaño del *buffer* mientras que para 4 y 8 procesos únicamente se producirá una leve bajada y estabilización de las prestaciones.

Comunicación	Num proc	Tamaño	Mejor Opción >Tamaño	Mejor Opcion <Tamaño
Allreduce	2	65536	NCCL Malloc/Managed	Cuda Aware Malloc Host
Allreduce	4	131072	NCCL Malloc/Managed	Cuda Aware Malloc Host
Allreduce	8	131072	NCCL Malloc/Managed	Cuda Aware Malloc Host
Reduce	2	2097152	NCCL Malloc/Managed	Cuda Aware Malloc Host
Reduce	4	32768	NCCL Malloc/Managed	Cuda Aware Malloc Host
Reduce	8	65536	NCCL Malloc/Managed	Cuda Aware Malloc Host
Bcast	2	1048576	NCCL Malloc/Managed	Cuda Aware Malloc Host
Bcast	4	524288	NCCL Malloc/Managed	Cuda Aware Malloc Host
Bcast	8	262144	NCCL Malloc/Managed	Cuda Aware Malloc Host
Send/Recv	2	2097152	Cuda Aware Malloc Host	Cuda Aware Malloc Managed/Host

Tabla 4.1: Comparativa de las prestaciones obtenidas en cada una de las gráficas analizadas, en esta se indica que combinación de biblioteca-reserva de memoria proporciona mejores prestaciones, al variar esto para distintos tamaños de *buffer* se indica a partir de que tamaño cambia esto

CAPÍTULO 5
Optimización de las comunicaciones en HELENNA

En el capítulo anterior hemos realizado pruebas para comprobar cómo afectaban diferentes variantes al comportamiento obtenido a la hora de ejecutar primitivas de comunicación entre procesos. En este capítulo, realizaremos una serie de pruebas sobre la herramienta de entrenamiento distribuido de redes neuronales HELENNA para comprobar cómo podemos optimizar las comunicaciones presentes en los entrenamientos que esta herramienta realiza. Entre estas pruebas comprobaremos la diferencia entre un entrenamiento de CPU y GPU, cómo afecta la frecuencia de sincronización en un entrenamiento asíncrono en MPI, MPI CUDA-aware y NCCL y compararemos en HELENNA los diferentes tipos de memoria estudiados en el capítulo 3.

5.1 Comparación CPU-GPU

En primer lugar analizaremos el uso de distintos tipos de entrenamiento de redes neuronales sobre CPU para así poder comparar cómo afecta el uso de uno u otro en las prestaciones obtenidas. A su vez, realizaremos una comparación de los resultados obtenidos en estos entrenamientos mediante la CPU con un mismo entrenamiento realizado sobre la GPU. Así podremos comparar las prestaciones para cada uno de estos dispositivos así como la efectividad de una GPU a la hora de acelerar los cálculos necesarios para realizar un entrenamiento de redes neuronales. Al hacer uso de la GPU utilizaremos la librería cuBLAS que nos permitirá acelerar los cálculos realizados por esta al igual que se usará MKL para la CPU con el mismo fin.

A la hora de realizar las pruebas, los parámetros escogidos han sido el uso del *dataset* CIFAR-10, la red vgg1 y un *batch size* de 512 que será dividido entre los distintos procesos, teniendo un *batch* local de 256 para 2 procesos y de 128 para 4 procesos. Además, la ejecución de la GPU será asíncrona, comunicándose cada 64 *batch* al igual que la versión asíncrona de CPU.

Podemos apreciar en la figura 5.1 que la variante que utiliza *parameter server* con la CPU es la que presenta un tiempo de ejecución más elevado, seguido de las versiones síncrona y asíncrona de CPU. Por último tenemos la versión de GPU, que emplea el mismo tipo de entrenamiento que la versión asíncrona de CPU. Podemos ver que la diferencia es muy significativa pasando de 500 segundos en las versiones de CPU a unos 5 segundos en la versión con GPU. Este ejemplo nos muestra cuan útil puede ser el uso de una GPU en un entrenamiento de redes neuronales presentado este un rendimiento mucho mayor al de las CPU.

A partir de este punto, exploraremos formas de explotar el uso de las GPU en este tipo de entrenamientos a través de la optimización de las comunicaciones. Esto también nos muestra que los entrenamientos tanto síncronos como asíncronos presentan tiempos de ejecución menores a *parameter server*, ya que estos dos tipos de entrenamientos aprovechan todos los nodos disponibles al contrario de *parameter server*, que necesita utilizar uno exclusivamente para actualizar los parámetros, no aprovechándolo a la hora de realizar las operaciones de cálculo.

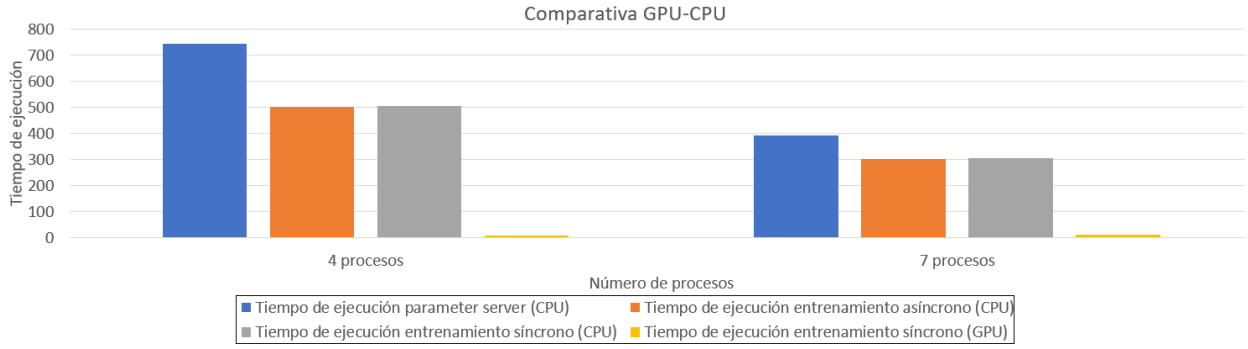


Figura 5.1: Comparativa de entrenamientos en CPU síncronos, asíncronos y mediante *parameter server* con un entrenamiento asíncrono en GPU con el *dataset* CIFAR-10,la red vgg1,1 *epoch* y un batch size de 512

5.2 Entrenamiento asíncrono y librerías GPUDirect

A la hora de analizar, cómo reducir el tiempo de comunicación se tendrán en cuenta tres factores: 1) cada cuanto se comunican los distintos procesos entre sí, 2) qué implementación utilizaremos entre MPI, MPI CUDA-aware y NCCL y 3) cuantos procesos se ejecutan. Para realizar el análisis usaremos el modo de entrenamiento asíncrono, variando para una misma configuración cada cuantos *batches* se comunican los procesos. Esto lo podemos indicar a través del parámetro "mpi-avg" que nos proporciona HELENNA y que nos permite indicar cada cuántos *batches* se realizan las comunicaciones.

5.2.1. Gráficas TAU de comunicaciones

Al sincronizar los procesos, cada *x batches*, en lugar de cada *batch* reduciremos la frecuencia con la que los distintos nodos se comunican entre sí, por lo que habrá menos llamadas a las primitivas de comunicación necesarias para realizar dicha comunicación. Este comportamiento lo podemos apreciar en unas gráficas obtenidas mediante la herramienta TAU. En estas comparamos el comportamiento mostrado para una misma configuración con el *dataset* CIFAR-10, la red vgg16, 8 procesos y un *batch* de 64. Con estos parámetros se realiza una ejecución en HELENNA que únicamente cambia el parámetro "mpi-avg", siendo 64 para una y 2048 para otra por lo que en una nos comunicaremos cada 64 *batches* (Figura 5.2) y en la otra cada 2048 *batches* (Figura 5.3).

Respecto a estas gráficas hay que resaltar que estas muestran la evolución del tiempo de ejecución de una aplicación, mostrando eventos que transcurren durante este tiempo. Podemos observar que en estas gráficas existen 8 "líneas de tiempo" que se corresponde con el número de procesos con el que se ha ejecutado la aplicación, por lo que cada una representa la ejecución de un proceso. Para cada proceso, se muestran ciertos sucesos que ocurren a lo largo de su ejecución. Estos se identifican con los rectángulos de distintos colores que encontramos en cada una de las "líneas de tiempo" para los distintos procesos.

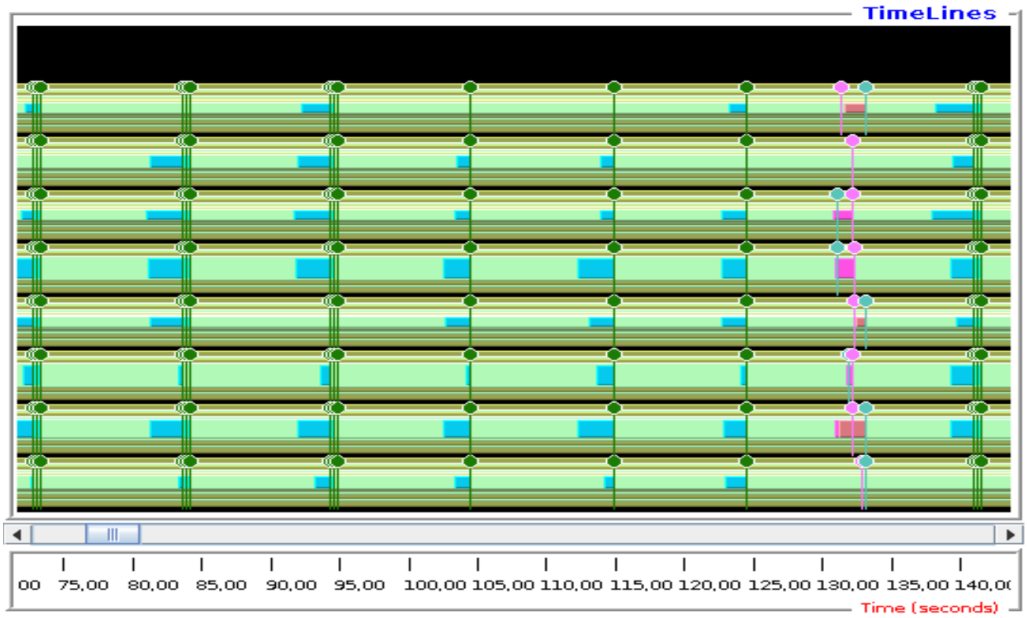


Figura 5.2: Representación de las comunicaciones para una ejecución con un "mpi-avg" de 64

Estos rectángulos pueden representar por ejemplo la ejecución de primitivas de comunicación como *Broadcast*, *Reduce* o *Allreduce* durante cierta parte del tiempo de ejecución total.

En el apéndice B podremos encontrar una leyenda para los "mpi-avg" de 64 (Figura B.1) y 2048 (Figura B.2) en el que se indica a qué corresponde cada uno de los colores de estos rectángulos respecto a la aplicación, los cuadros exteriores corresponden con la ejecución de la aplicación TAU y de la aplicación HELENNA cubriendo estas la totalidad del tiempo de ejecución.

El único aspecto que no se cita en estas leyendas y que aparece representado en la gráfica son estas figuras de diferentes colores formadas por un círculo y una línea del mismo color. Estas figuras se corresponden con un mensaje con el tamaño del *buffer* a enviar por distintas primitivas de comunicación siendo diferente el color de esta figura para cada primitiva de comunicación. Para la gráfica de 64 "mpi-avg" el color rosa corresponde con *Broadcast*, el verde oscuro con *Allreduce* y el azul verdoso con Reduce. Respecto a la gráfica de 2048 "mpi-avg" el negro corresponde con *Allreduce*, el rosa con Reduce y el verde con *Broadcast*.

Una vez explicada la simbología utilizada podemos comentar lo que se observa en ellas. Podemos ver que para un "mpi-avg" de 64 el número de llamadas a primitivas de comunicación es significativamente mayor al observado para un "mpi-avg" de 2048. Esto es así incluso teniendo en cuenta que tal y como podemos observar en la parte inferior de estas figuras, donde se muestra el fragmento de tiempo en segundos representado por estas, el tiempo de ejecución que abarca la figura 5.2 es menor al de la figura 5.3. Mientras que con un "mpi-avg" 64 se está cubriendo aproximadamente desde el segundo 25 al 137.5 de la ejecución, para un "mpi-avg" de 2048 se representa más o menos desde el segundo 75 al 135 de la ejecución, siendo una fracción de tiempo menor a la de "mpi-avg" 64. En conclusión, podemos observar cómo el aumentar el parámetro de "mpi-avg" conlleva que se realice una menor cantidad de llamadas a primitivas de comunicación.

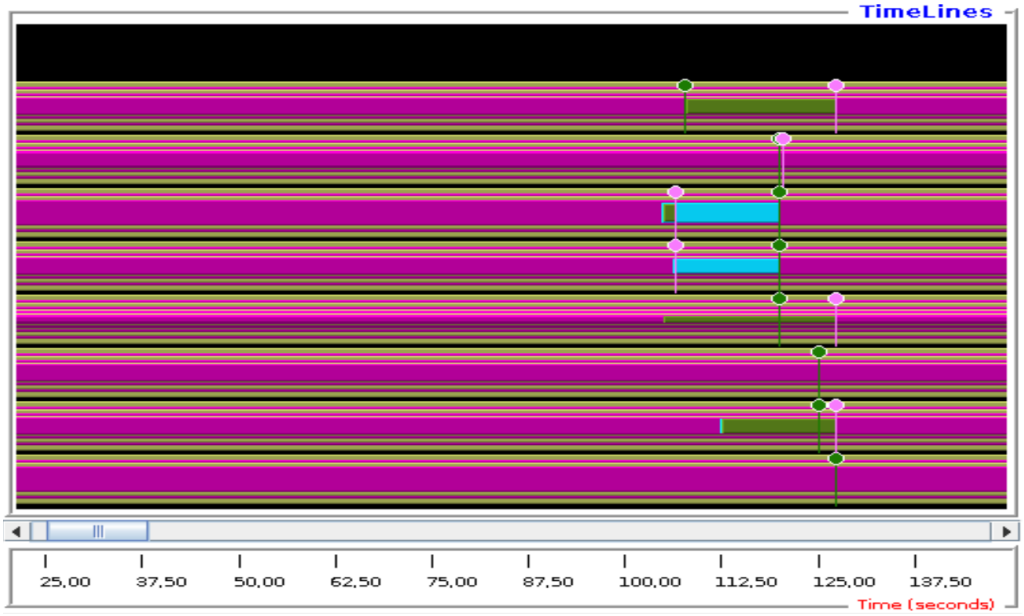


Figura 5.3: Representación de las comunicaciones para una ejecución con un "mpi-avg" de 2048

5.3 Diferentes tipos de reserva de memoria

En este apartado, analizaremos como los diferentes tipos de reserva de memoria introducidos anteriormente pueden afectar en una ejecución en la herramienta HELENNA. Para realizar estas pruebas se ha utilizado una misma configuración para red vgg1 con el *dataset* CIFAR-10 en la que se ejecutarán 4 procesos, el tamaño de *batch* global será de 2048, un "mpi-avg" de 64 y comunicación mediante MPI.

En la Figura 5.4 se miden los distintos tiempos de ejecución en segundos para los 3 tipos de reserva de memoria estudiados. Podemos observar que para *MallocHost()* se obtiene un tiempo de ejecución muchísimo mayor al obtenido para *MallocManaged()* y *Malloc()*, que presentan tiempos similares. Este comportamiento se debe a las características de *MallocHost()* que como ya hemos mencionado anteriormente si se realizan reservas de cantidades de memoria muy grandes por parte de *MallocHost()* esto implicará pérdidas en las prestaciones globales del sistema. En este caso se realiza una reserva de memoria de 4.74 GB, bastante superior a los 32 MB con los que realizábamos pruebas en el capítulo 3. Para este tamaño ya era notable la bajada de las prestaciones, por lo que para una reserva de memoria mucho mayor como en este caso la bajada de las prestaciones será mucho más notable.

Este comportamiento lo podemos observar más detalladamente mediante TAU Commander que nos permite observar cómo aumenta el tiempo de ejecución de la función *pthread_join* en *MallocHost()* (Figura 5.7) respecto a *MallocManaged()* (Figura 5.6) Y *Malloc()* (Figura 5.5). Además, en el apéndice B también podemos encontrar la leyenda correspondiente a cada una de estas gráficas (Figura B.3 para *Malloc()*, Figura B.4 para *MallocManaged()* y Figura B.5 para *MallocHost()*).

La función *pthread_join* suspende la ejecución del hilo de llamada hasta que el hilo de destino finaliza, a menos que el hilo de destino ya haya terminado. Esto quiere decir que a la hora de reservar la memoria mediante *MallocHost()* causamos que existan una gran cantidad de hilos esperando a otros hilos, lo que ralentiza la ejecución.

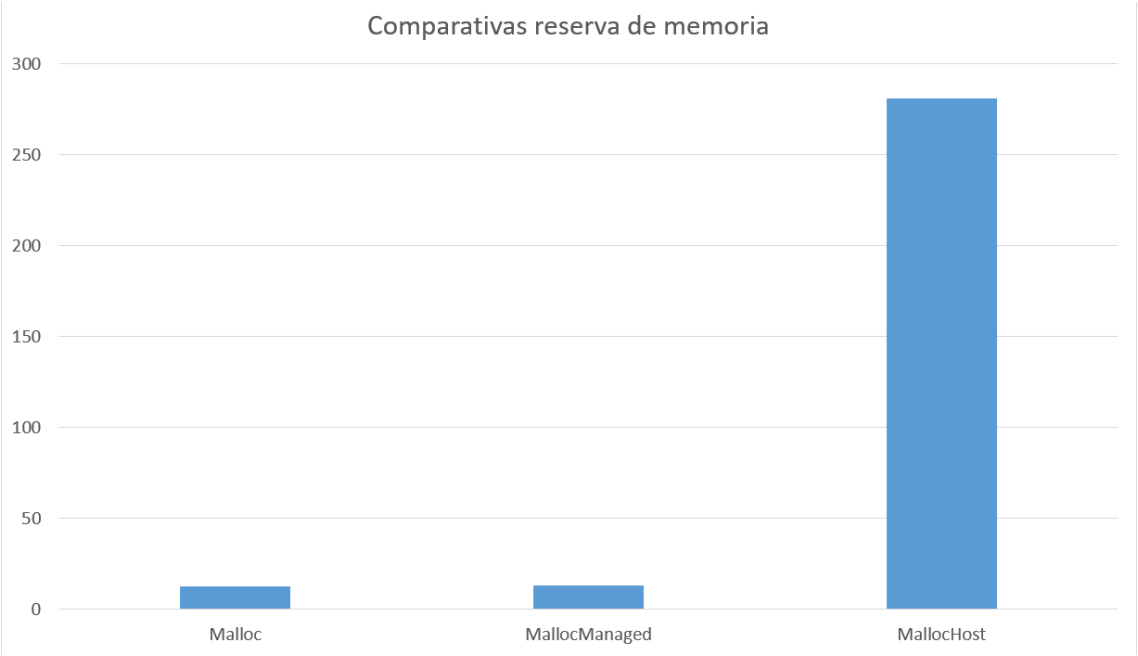


Figura 5.4: Comparativa del tiempo de ejecución de una reserva de memoria mediante *Malloc()*, *MallocManaged()* y *MallocHost()* en la red neuronal vgg16

Estos experimentos nos hacen concluir que no sería adecuado el uso de *MallocHost()* para este tipo de redes neuronales que requieren de una gran cantidad de reserva de memoria para su entrenamiento.

5.4 Gráficas comparativas "mpi-avg"

En este apartado vamos a analizar los resultados obtenidos para ejecuciones en las que se varía el parámetro introducido mediante la opción "mpi-avg" desde un valor 1 hasta un cierto valor máximo. El valor 1 corresponde al comportamiento de un entrenamiento síncrono, ya que se realizarían sincronizaciones cada *batch*. El tamaño máximo a estudiar para el parámetro "mpi-avg" viene dado en base al número de invocaciones a *Allreduce* que se han realizado durante el transcurso de la ejecución. Utilizamos el valor que corresponde a que el número de invocaciones sea 0. Un ejemplo de este comportamiento lo tenemos en la tabla 5.1 para un tamaño de *batch* de 256 y 8 procesos ejecutando en la red vgg16.

En dicha tabla podemos apreciar tanto como disminuye el número de llamadas a *Allreduce* al aumentar el "mpi-avg" como que para un "mpi-avg" de 8096 no se registra ninguna llamada a *Allreduce*. En este punto lo que se produce es una sincronización al final del *epoch*, cuando se ha recorrido por completo el *dataset* de entrada. Por lo tanto, y siguiendo con el motivo de introducir esta tabla, el máximo "mpi-avg" que estudiaremos para esta combinación será de 8096. Este procedimiento de determinar el máximo valor para "mpi-avg" a partir del punto en el que el número invocaciones a "Allreduce" es 0, se seguirá también en las ejecuciones con las redes vgg1 y resnet, que visualizaremos posteriormente en este capitulo.

Además de variar el tamaño del parámetro "mpi-avg", en estas pruebas se variará el número de procesos, lanzándose ejecuciones para 2, 4 y 8 procesos en los que se variará el "mpi-avg" y una ejecución con 1 proceso para comprobar cuáles serían las prestaciones

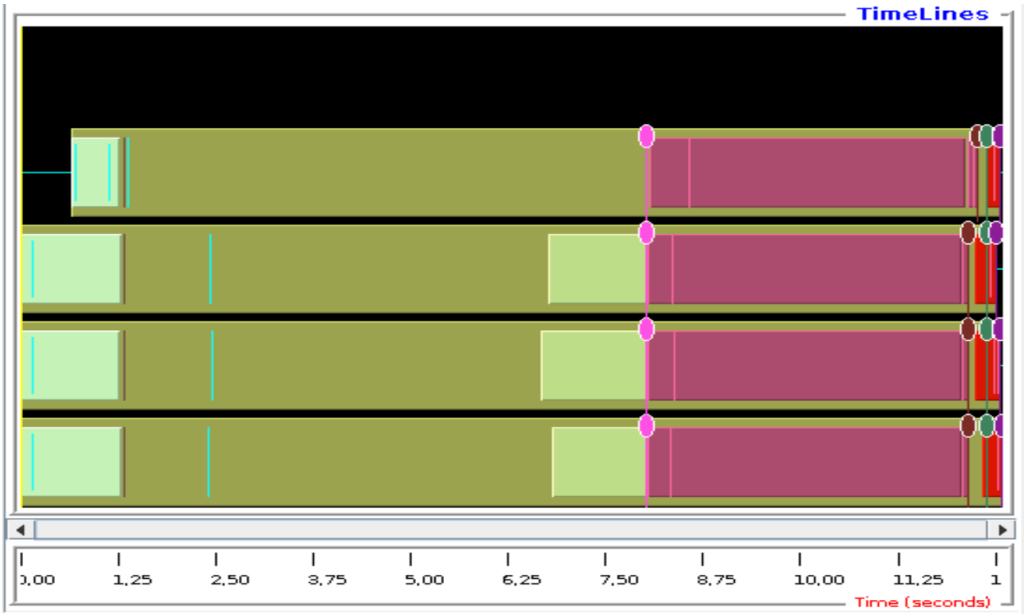


Figura 5.5: Representación de una ejecución en HELENNA para la red neuronal vgg16 con la reserva de memoria *Malloc()*

obtenidas al realizar el entrenamiento secuencial. Además, los experimentos con varios procesos realizando la comunicación con NCCL, MPI CUDA-aware y MPI.

Este conjunto de pruebas se realizarán en tres redes de diferente complejidad, estas son vgg1, vgg16 y resnet. Entre ellas, vgg1 sería la de menor complejidad estando conformada por un menor número de capas. Mientras que resnet sería aquella con un mayor grado de complejidad al estar formada por un número de capas mayor. Mientras que la red vgg16 sería la que presentaría una complejidad intermedia entre estos ejemplos. Se puede comprobar tanto el número de capas como las funciones de activación utilizadas por cada una en el apéndice A, para las redes vgg1 (Figura A.1), vgg16 (Figura A.2 y A.3) y resnet (Figura A.4 y A.5) .

Valor parámetro -avg	Número de llamadas *Allreduce*
1	1404000
2	698400
4	349200
8	174600
16	87264
32	43632
64	21816
128	10872
256	5472
512	2664
1024	1224
2048	648
4096	216
8192	0

Tabla 5.1: Número de llamadas a *Allreduce* para un tamaño de *batch* de 256, 8 procesos y un *epcoch* de 100 en una ejecución en la red vgg16.

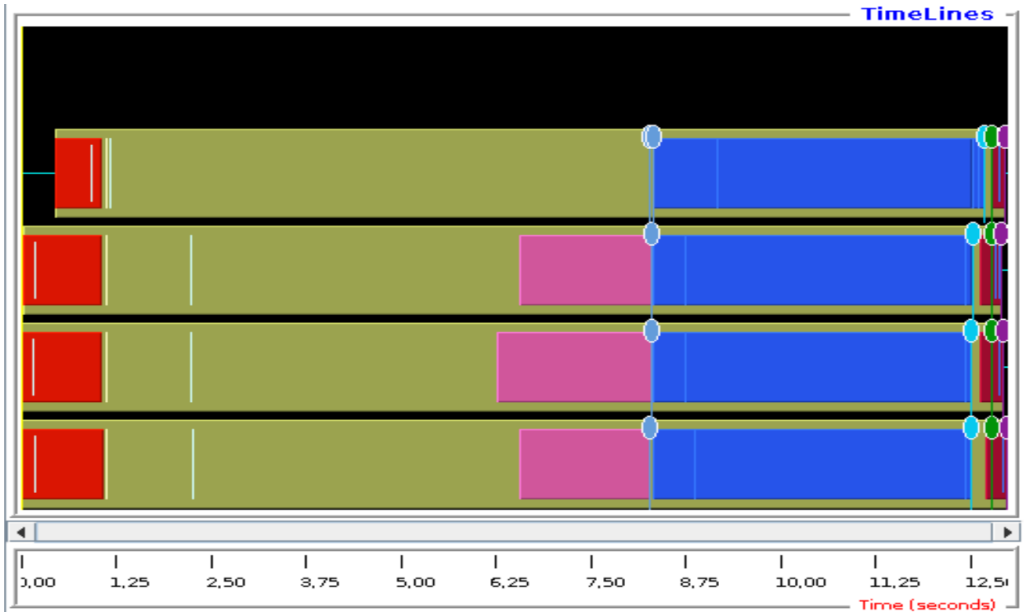


Figura 5.6: Representación de una ejecución en HELENNA para la red neuronal vgg16 con la reserva de memoria *MallocManaged()*

A la hora de realizar las pruebas para estas tres redes, se ha elegido un tamaño de *batch* de 256 y un *epoch* de 100. El número de *epoch* es 100, ya que este es lo suficientemente grande para obtener una precisión (*accuracy*) en el modelo medianamente alta. Hay que tener en cuenta que en vgg16 (en las otras redes no varía tanto) se obtiene una precisión muy baja con un número de *epoch* menor como podría ser 30-40. Por último, respecto al valor de "mpi-avg" su valor máximo tal y como se ha dicho anteriormente viene dado por el punto en el que se obtiene 0 en el número de llamadas a *Allreduce*. En este caso, el valor máximo para las 3 redes será 8192. Respecto al valor mínimo, su valor será 1. En este caso pese a que nuestro modelo de entrenamiento sea asíncrono, este se comportará como lo haría un entrenamiento síncrono al sincronizarse cada *batch*.

Una vez vistos los distintos parámetros de la ejecución analizaremos el comportamiento que presentan estas gráficas teniendo en cuenta que para cada una de ellas el eje vertical representa el tiempo de ejecución obtenido medido en segundos y el horizontal el valor del parámetro 'avg'. Además, otro factor a tener en cuenta para comprender adecuadamente estas gráficas es que en las gráficas que representan el tiempo de ejecución, el color negro representa la ejecución secuencial, los azules NCCL, los naranjas MPI CUDA-aware y los verdes MPI.

5.4.1. Análisis vgg1

En la figura 5.8 podemos observar el tiempo de ejecución para cada una de las combinaciones probadas. Como ya se ha dicho anteriormente, tanto para vgg1 como para las redes que analizaremos posteriormente, existen 3 factores que tendrán influencia en el tiempo de ejecución obtenido. Estos son: el valor "mpi-avg", el número de procesos y que librería se está utilizando para implementar las primitivas de comunicación (NCCL, MPI CUDA-aware o MPI). En este análisis se valorará que impacto ha tenido cada uno de estos factores en el tiempo de ejecución obtenido. En primer lugar tenemos el número de procesos. Un mayor valor de este factor en principio debería disminuir el tiempo de ejecución necesario para realizar el entrenamiento, ya que a mayor sea su valor mayor sera la capacidad de realizar los cálculos necesarios en paralelo y así acelerar la ejecución. Sin embargo, si observamos la comparativa para un "mpi-avg" de 1 en cualquiera de las tres

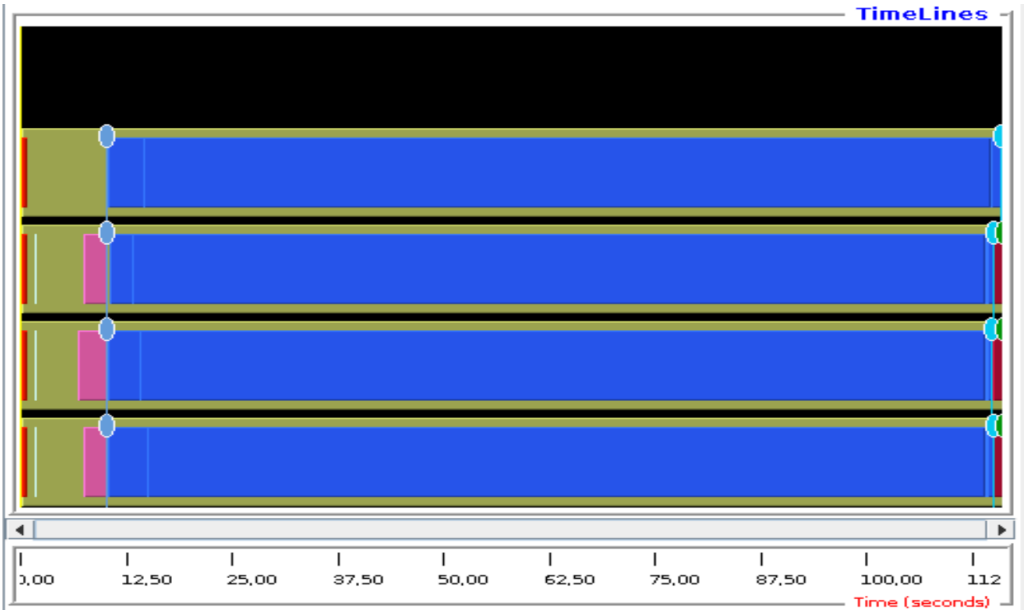


Figura 5.7: Representación de una ejecución en HELENNA para la red neuronal vgg16 con la reserva de memoria *MallocHost()*

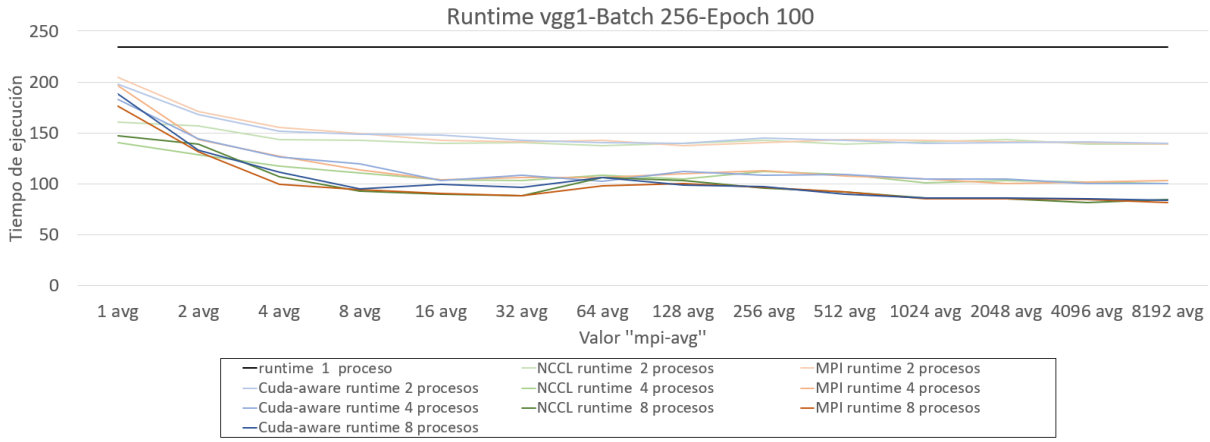


Figura 5.8: Comparativa del tiempo de ejecución de NCCL,MPI y MPI CUDA-aware con distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red vgg1, con un *batch* de 256 y un *epoch* de 100

librerías, observamos que esto no es así. Únicamente en MPI se produce esta relación de a mayor número de procesos menor tiempo de ejecución en las otras 2 incluso se obtiene un mayor tiempo de ejecución para 8 que para 4 procesos y para MPI pese a que si se cumpla pasamos de un tiempo de ejecución de 205 segundos con 2 procesos a uno de 175 segundos con 8 procesos pese a haber aumentado la capacidad de cálculo x4.

Por lo tanto podemos concluir, que este factor por sí mismo no permite obtener mejoras significativas en el tiempo de ejecución. El siguiente factor que tendremos en cuenta es el de "mpi-avg". En este caso, sí que se aprecia como el tiempo de ejecución obtenido desciende significativamente desde una ejecución con un "mpi-avg" de 1 en la que la sincronización sea muy elevada a una con un "mpi-avg" de 8192 en la que se produzca la sincronización al final del *epoch*. Podemos observar que, únicamente variando este factor de 1 a 8192 para 2 procesos en MPI o MPI CUDA-aware (en NCCL esto no se aplica, pero también es significativo el efecto de este factor) obtenemos mejores prestaciones que con 8 procesos y un "mpi-avg" de 1 por lo que podemos que considerar que este factor

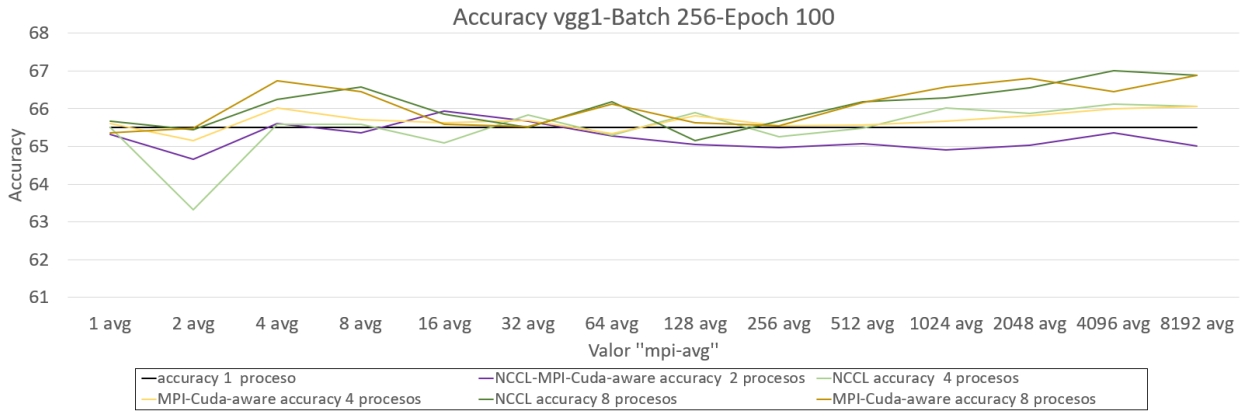


Figura 5.9: Comparativa del *accuracy* de NCCL,MPI y MPI CUDA-aware con distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red vgg1, con un *batch* de 256 y un *epoch* de 100

es incluso más importante que el número de procesos. Este efecto de "mpi-avg" para 2 procesos es incluso más notorio para 4 y 8 procesos, esto se debe a un mayor coste de las invocaciones a *Allreduce* en estas ejecuciones . Un "mpi-avg" medianamente alto (con esto nos referimos a valores a partir de la parte intermedia de la gráfica) sí que permite aprovechar el paralelismo que proporciona un mayor número de procesos, por lo que podemos decir que en este caso con un "mpi-avg" lo suficientemente grande sí que se da la relación de a mayor número de procesos menor tiempo de ejecución. Conforme va aumentando el número de procesos, la mejora obtenida es menor. Podemos observar que la mayor mejora la obtenemos al pasar de 1 proceso a 2, tras esto obtenemos una mejora algo menor al pasar de 2 a 4 procesos y aun menor al pasar de 4 a 8.

Tras estos dos factores, tendríamos un último que es la biblioteca utilizada para implementar las primitivas de comunicación. En primer lugar, a la hora de comparar las 3 bibliotecas hay que tener en cuenta varias cosas. Tal y como podemos observar en las gráficas, las prestaciones obtenidas por las 3 alternativas se va igualando conforme aumenta el "mpi-avg", esto se debe a que conforme este aumenta las llamadas a primitivas de comunicación se reducen, por lo que el peso de las comunicaciones en el tiempo de ejecución obtenido va decreciendo conforme el valor de este parámetro aumenta. Por esta razón, a la hora de analizar qué biblioteca es mejor para esta red y las siguientes redes a analizar, se tendrán en cuenta los valores más pequeños (como pueden ser 1,2 y 4) para "mpi-avg", ya que en estos es donde el peso de las comunicaciones es mayor e influye más el uso de una u otra biblioteca. Por tanto, si tenemos en cuenta estos valores más pequeños para "mpi-avg", podemos observar como NCCL claramente obtiene mejores prestaciones que las otras dos alternativas.

Respecto a las prestaciones obtenidas para cada una de las bibliotecas hay que decir varias cosas. En primer lugar, que se ha elegido la reserva de memoria *Malloc()* a la hora de realizar estas pruebas. Esto se debe a que este tipo de reserva de memoria es el estándar y se han obtenido peores resultados con una reserva de memoria *MallocHost()*, en las pruebas realizadas anteriormente para HELENNA comparados con los resultados obtenidos para una reserva de memoria mediante *Malloc()* o *MallocManaged()*, los cuales obtienen prestaciones similares. Podemos observar que entre MPI y MPI CUDA-aware existen ciertas diferencias en los resultados de uno y de otro, pero estas diferencias no son muy significativas y no parecen seguir ningún patrón especifico. Para comprobar esto se ha observado si, esta nula diferencia entre las prestaciones obtenidas entre ambas librerías existía para otras configuraciones. Para ello se han realizado pruebas con los mismos parámetros para la red vgg1, pero con distintos tamaños de *epoch*, las gráficas correspon-

dientes a estas pruebas para 2 (Figura B.6), 4 (Figura B.7) y 8 (Figura B.8) procesos las podemos encontrar en el apéndice B. En estas pruebas, podemos observar como el que MPI o MPI CUDA-aware obtienen unas mejores prestaciones no sigue ningún patrón especifico. Por lo tanto, podemos concluir que tanto MPI como MPI CUDA-aware obtienen prestaciones similares. Tras haber observado los resultados obtenidos para las 3 bibliotecas estudiadas, podemos deducir que NCCL será la biblioteca que proporciona unas mejores prestaciones y que sus prestaciones se igualan con las de las otras alternativas para los mayores tamaños de ”mpi-avg”.

Además del tiempo de ejecución otro factor importante a la hora de hacer un análisis, es la precisión o *accuracy* obtenida tras realizar un entrenamiento. Antes de comentar la gráfica correspondiente al *accuracy*, se van a hacer unas breves aclaraciones sobre estas que podrán ser aplicadas en análisis posteriores. Para 2 procesos NCCL, MPI y MPI CUDA-aware presentan un mismo *accuracy* por lo que este sera representado como una única recta de color morado, para 4 y 8 procesos NCCL se obtiene un *accuracy* diferente al de MPI y MPI CUDA-aware por lo que por una parte se representara el *accuracy* de NCCL de color verde y por otra el de MPI CUDA-aware de color amarillo, por último estaría la ejecución con 1 proceso, esta representada con el color negro, al no realizar ninguna comunicación no varía para los distintos valores de ”mpi-avg” ni se ve afectada por el uso de una u otra librería. La diferencia entre el *accuracy* de NCCL y el de las otras alternativas para un mismo número de procesos suele ser mínima, pero se representará igualmente para así observa como ambas varían de forma similar. Por último cabe apreciar que aunque para 4 y 8 procesos varíe el *accuracy* entre NCCL y las otras alternativas, para un ”mpi-avg” de 8192 se obtiene el mismo *accuracy* para ambas. Esto se debe a que en este caso no se hace ninguna llamada a *Allreduce* simplemente se sincronizan al completar el *epoch* por lo que las diferencias entre ambas alternativas se eliminan. En lo referente a las causas de esta diferencia entre los *accuracy* de NCCL y las alternativas que hacen uso de MPI, es posible que esto se debe a la naturaleza de las operaciones colectivas de NCCL, que al contrario de las operaciones punto a punto (en las que el *accuracy* es el mismo) se realizan de una forma especifica para comunicar eficientemente las GPU, por que es diferente a las operaciones colectivas de MPI y por esto dan un *accuracy* distinto pese a realizar una misma operación. En este caso, la operación colectiva a realizar es *Allreduce*, que a la hora de realizar la comunicación suma y divide. Una posible explicación sería que el orden de estas operaciones sea distinto en MPI y NCCL. Aunque el *accuracy* presentado por NCCL sea diferente, este suele ser similar al presentado por MPI y MPI CUDA-aware.

Respecto a esta gráfica que representa el *accuracy* para vgg1 (Figura 5.9), su comportamiento es curioso, ya que al contrario de lo que observaremos en gráficas posteriores el mayor *accuracy* lo obtenemos con 8 procesos y con el mayor tamaño de ”mpi-avg”. Para 4 procesos también se da que 8192 es el valor para el que obtiene un mayor *accuracy*. Esto no ocurre únicamente para este valor de ”mpi-avg” sino que hay una evolución ascendente en la que para 4 y 8 procesos aumenta el *accuracy*, conforme aumenta el ”mpi-avg”. En la ejecución con 2 procesos no se da esto, obteniéndose un accuracy menor al obtenido por 1 proceso para la mayoria de combinaciones y no mejorando dicho accuracy al aumentar el ”mpi-avg”.

Por lo tanto, para esta configuración, podríamos concluir que los valores de los parámetros que nos permitirían obtener unos mejores resultados en cuanto a tiempo de ejecución y *accuracy* serían un alto valor para el parámetro ”mpi-avg”, una ejecución con 8 o 4 procesos (mejores prestaciones para 8 procesos) y el uso de la librería NCCL. Pese a que para esta configuración podamos concluir esto, de momento no podemos concluir que esta elección de valores para los parámetros será adecuada para todas las configuraciones, ya que tal y como veremos para los siguientes casos, este comportamiento es algo

anormal, ya que lo más típico es que para 8 o 4 procesos se vea resentido el *accuracy* para los valores más altos para "mpi-avg". Esto es así incluso en otras combinaciones con la red vgg1 como las que podemos ver en el apéndice B en el que para una misma configuración, pero cambiando únicamente el número de *epoch* a 40 (Figura B.9), 35 (Figura B.10) y 30 (Figura B.11), el comportamiento mostrado no se corresponde con el mostrado para un *epoch* de 100. Para estas dos combinaciones el *accuracy* desciende conforme aumenta el valor para "mpi-avg" y el número de procesos. Analizaremos en más profundidad este tipo de comportamientos en las gráficas de *accuracy* para vgg16 y resnet18.

5.4.2. Análisis vgg16

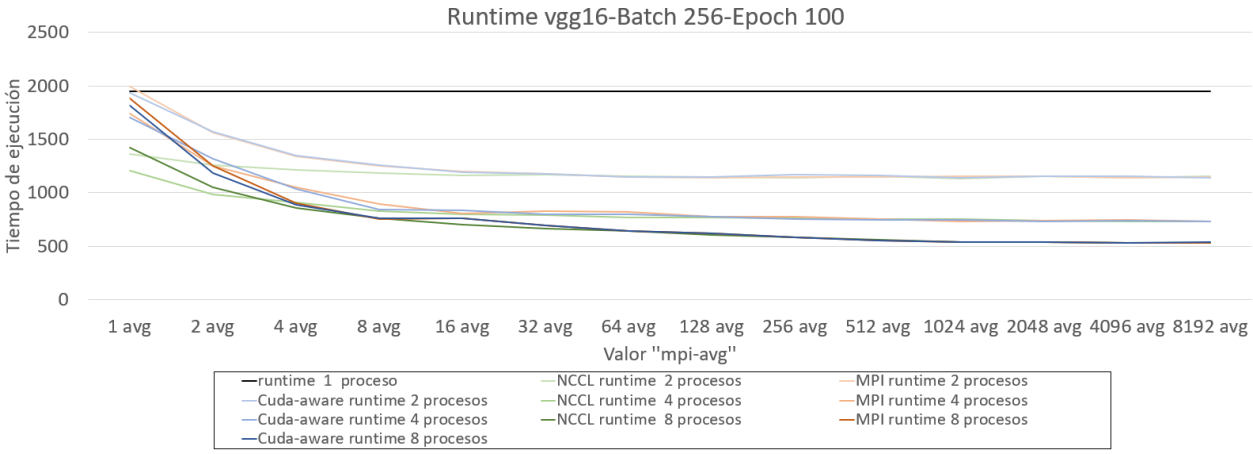


Figura 5.10: Comparativa del tiempo de ejecución de NCCL,MPI y MPI CUDA-aware con distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red vgg16, con un *batch* de 256 y un *epoch* de 100

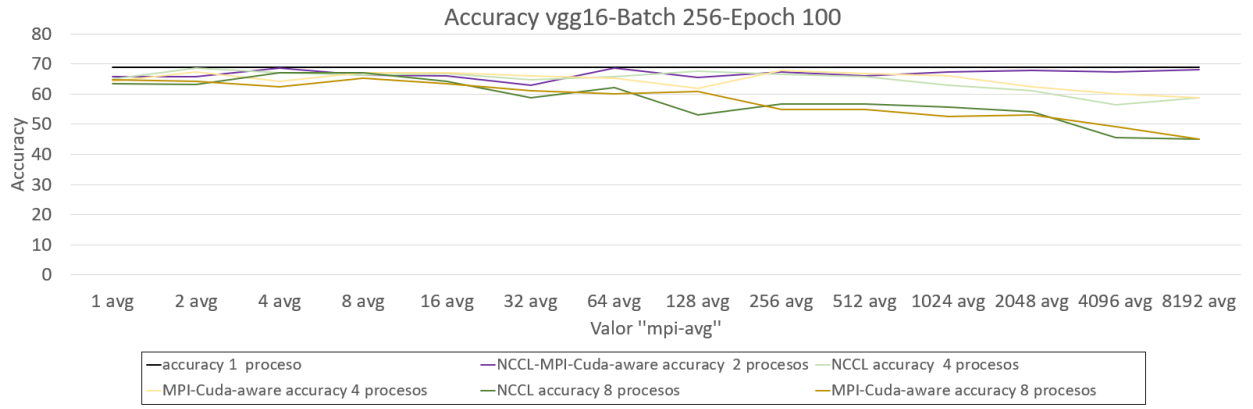


Figura 5.11: Comparativa del *accuracy* de NCCL,MPI y MPI CUDA-aware con distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red vgg16, con un *batch* de 256 y un *epoch* de 100

En la Figura 5.10, podemos observar los resultados obtenidos para la red vgg16. Podemos observar que exceptuando los mayores tiempos de ejecución obtenidos, los resultados son bastante parecidos. Tal y como se ha citado antes, la importancia de "mpi-avg" es muy alta al reducir este el elevado peso de las comunicaciones y no pudiendo aprovecharse los recursos proporcionados por un mayor número de procesos sin este factor. Además, NCCL sigue siendo la biblioteca que proporciona unas mejores prestaciones entre las 3 alternativas escogidas y para un valor alto de "mpi-avg" se obtiene una mejora cada vez menor al aumentar el número de procesos.

Este y otros aspectos son comunes a los comportamientos mostrados en la red vgg1. Para comenzar a encontrar diferencias podemos observar la figura 5.11 que representa el *accuracy* obtenido. Podemos observar que al contrario de lo que ocurría con vgg1 para 4 y 8 procesos se muestra una tendencia más o menos sostenida en la que para los valores más altos de "mpi-avg" a mayor sea este valor menor es el *accuracy* obtenido. Este comportamiento es más notorio para 8 que para 4 procesos, siendo el descenso más sostenido y alcanzando un *accuracy* más bajo. Para 2 procesos no encontramos este comportamiento. En primer lugar, no varía tanto el *accuracy*, ya que varía entre un 3 y 4 % del *accuracy* más alto al más bajo, al contrario de lo que pasa con 4 y 8 procesos en los que llega a variar un 10 % en uno y un 20 % en otro. Además, aparte de esto, podemos observar como para los valores más altos de "mpi-avg" es donde encontramos un *accuracy* más bajo de forma sostenida.

Para comprender este comportamiento hay que tener en cuenta el propósito de las comunicaciones en este modelo de entrenamiento asíncrono. Estas permiten a todos los procesos conocer los parámetros actualizados en cierto momento del entrenamiento permitiendo ajustar con más precisión al conocer estos parámetros actualizados. Sin embargo, al reducirse la frecuencia con la que se realizan estas sincronizaciones estos ajustes se realizan de forma menos precisa, por lo que se ve resentida la precisión del modelo final. Esto es más notorio cuanto mayor es el número de procesos, ya que cada uno de estos procesos "conocerá" menos información respecto al total de los datos de entrada por lo que al juntarse esto con una baja actualización causa que se obtenga un *accuracy* mucho menor. Por esta razón con 2 procesos, pese a reducir la frecuencia de sincronización como cada uno de ellos conoce el 50 % de datos de entrada, el impacto de reducir la frecuencia de las comunicaciones no es tan notorio como con 4 y 8 procesos en los que cada proceso "conoce" una porción mucho menor.

Hemos podido observar como se reduce el *accuracy* obtenido para un número alto de procesos y del valor de "mpi-avg". Sin embargo, esto nos hace plantearnos una pregunta ¿Qué valores de los parámetros estudiados nos permiten obtener un buen equilibrio entre *accuracy* y tiempo de ejecución? En la gráfica correspondiente al tiempo de ejecución hemos podido observar cómo las opciones que nos permiten reducir en mayor medida el tiempo de ejecución son aquellas con un "mpi-avg" y número de procesos alto, pero esto nos plantea problemas respecto al *accuracy* obtenido. Para solucionar esto hay que tener presente un aspecto de estas gráficas. Tal y como se ha mencionado anteriormente existe un punto en el que las 3 bibliotecas comienzan a obtener prestaciones parecidas, no se podría decir que las prestaciones se estancan a partir de este punto, ya que sigue descendiendo el tiempo de ejecución obtenido. Pese a que no podamos decir que se estancan, sí que podemos afirmar que a partir de este punto o incluso algo antes para las combinaciones con NCCL la mejora obtenida con cada incremento de "mpi-avg" comienza a ser baja.

Teniendo en cuenta esto, primeramente hay que analizar a partir de qué punto el descenso en el *accuracy* es muy elevado, para 8 procesos podríamos decir que a partir de un "mpi-avg" de 128 para MPI y MPI CUDA-aware y de 64 para NCCL este comienza a bajar significativamente no pasando de un 56 % a partir de este punto y presentando entre un 63-64 y un 60 % para valores menores. Respecto al comportamiento que presenta el tiempo de ejecución con 8 procesos para ese punto de "mpi-avg" 64-128 ya se ha producido el mayor descenso del tiempo de ejecución pasando de unos 1800 segundos en MPI y MPI CUDA-aware o 1400 en NCCL a unos 640-620 entre un "mpi-avg" de 64 o 128. Para valores mayores de "mpi-avg" se seguirán experimentando mejoras en el tiempo de ejecución obtenido, pero estas serán ya menores. Para un "mpi-avg" de 8192 obtendríamos un tiempo de ejecución de 536 reduciendo así 100 segundos el tiempo de ejecución, pero no se obtiene una mejora tan significativa. Si se quisiera reducir al máximo el tiempo de

ejecución pese a disminuir algo más el *accuracy* el punto adecuado sería un "mpi-avg" de 1024, ya que obtiene un tiempo de ejecución de 539 segundos y el *accuracy* se reduce a un 54-53 % no a un 45 % del "mpi-avg" 8192. Por último, respecto a las ejecuciones con 8 procesos hay que comentar que si queremos una ejecución que mantenga un *accuracy* lo más alto posible lo más adecuado sería un "mpi-avg" de 8, ya que en este punto el *accuracy* conseguido es mayor o igual al obtenido con un "mpi-avg" de 1 y el tiempo de ejecución es mucho menor al obtenido con este valor para un "mpi-avg" de 1.

En cuanto a 2 procesos, aunque en este caso el *accuracy* no se vea demasiado resentido por un mayor valor de "mpi-avg" tampoco recomendaría aumentar este mucho más allá de un valor de 128, ya que, en este caso, las diferencias sí que son mínimas y prácticamente solo se deben a las variaciones en el tiempo de ejecución que acarrean estas ejecuciones por naturaleza. Esto lo podemos observar en que para un "mpi-avg" de 128 obtenemos un tiempo de ejecución de 1147 segundos y para "mpi-avg" de 2048 o 4096 obtenemos 1150 segundos mientras que para un "mpi-avg" de 1024 o 8192 obtenemos un tiempo de ejecución de 1139 segundos, por lo que en este caso no podríamos afirmar que a partir de este punto se obtenga una mejora en las prestaciones.

Por último quedaría comentar el comportamiento para 4 procesos. En este caso exceptuando un pico negativo que se da para el valor 128 para "mpi-avg" en NCCL al igual que se da para 8 procesos, observamos que para valores mayores a un "mpi-avg" de 1024 comienza a descender de forma más sostenida y brusca el tiempo de ejecución, para un valor de "mpi-avg" de 1024 se obtiene un tiempo de ejecución de unos 730 segundos, muy similar a lo obtenido para valores mayores de "mpi-avg" por lo que podemos concluir que este será el valor que nos proporciona un tiempo de ejecución y un *accuracy* óptimo.

De este análisis podemos concluir varias cosas. La primera es que conforme mayor sea el número de procesos si se esta ejecutando con valores altos para "mpi-avg" el descenso en el *accuracy* es mayor. Otra conclusión a la que hemos podido llegar, es que cuantos más procesos estén ejecutándose, el estancamiento del tiempo de ejecución obtenido se producirá para valores más altos de "mpi-avg", esto quiere decir que para 2 procesos el valor de "mpi-avg" en el que las prestaciones se estancan será mayor que para 8 procesos. Además, también hemos podido observar como existen ciertos valores para "mpi-avg" que nos proporcionan un balance entre tiempo de ejecución como *accuracy*, así como que aunque no queramos conseguir un balance entre ambos resultados sí que existen puntos más óptimos que otros a la hora de obtener unas mejores prestaciones.

5.4.3. Análisis resnet

En la figura 5.12 podemos observar los resultados referentes al tiempo de ejecución para la red resnet18. En ella podemos observar comportamientos parecidos a los mostrados para las dos anteriores redes analizadas como pueden ser el cómo se reduce el tiempo de ejecución para valores de "mpi-avg" altos, cómo para estos valores la mejora obtenida al aumentar el número de procesos es cada vez menor y otros aspectos ya citados. La única diferencia que parece presentar es respecto a las prestaciones mostradas para NCCL con 8 procesos. En este caso el tiempo de ejecución obtenido se aproxima mucho más que en las redes anteriores a las otras dos bibliotecas. Exceptuando esta pequeño apunte NCCL sigue siendo la alternativa que proporciona unas mejores prestaciones.

El único aspecto en el que diverge algo más con vgg1 y vgg16 es respecto al *accuracy* mostrado en la figura 5.13. En esta gráfica encontramos varias diferencias respecto al comportamiento mostrado para la red vgg16. La primera y más significativa es la mostrada por las ejecuciones con 4 procesos en la que entre un "mpi-avg" de 64 y 128 hasta uno

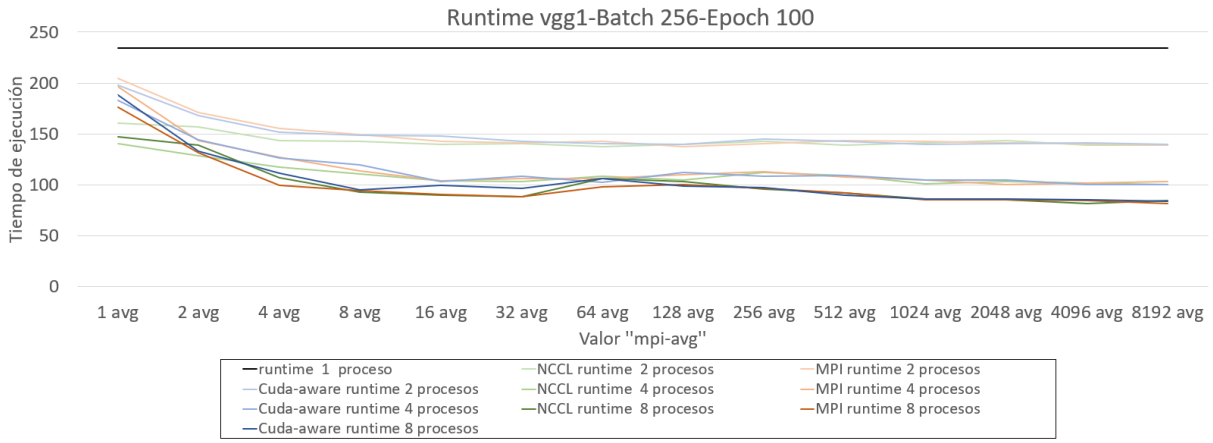


Figura 5.12: Comparativa del tiempo de ejecución de NCCL,MPI y MPI CUDA-aware con distintos ”mpi-avg” en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 256 y un *epoch* de 100

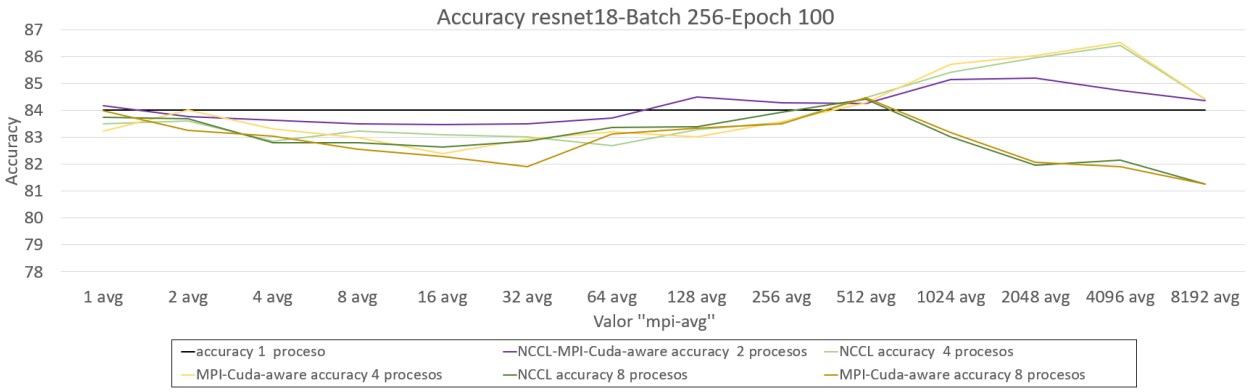


Figura 5.13: Comparativa del *accuracy* de NCCL,MPI y MPI CUDA-aware con distintos ”mpi-avg” en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 256 y un *epoch* de 100

de 4096 se experimenta una subida del *accuracy* conforme este valor aumenta, mientras que para un valor de 8192 comienza a descender este *accuracy*. Este comportamiento no se parece al mostrado por vgg16 y las pruebas en vgg1, ya que en este caso se da que una ejecución con 4 procesos obtiene un mayor *accuracy* para valores altos de ”mpi-avg” que una con un menor número de procesos como son 2 procesos. Además también se da que para 2 procesos, las ejecuciones con mayor *accuracy* tienen un valor de ”mpi-avg” medio alto (128-2048) empezando a descender este para los valores más grandes (4096-8192). Para 8 procesos sí que se da que el *accuracy* desciende para los tamaños más grandes de ”mpi-avg” a partir de un valor de 512 aunque también se experimentan valores más bajos para grados de sincronización mayores (como ocurre entre los valores 16-32 para MPI o MPI CUDA-aware).

A la hora de obtener las mejores prestaciones del binomio tiempo ejecución-*accuracy*, para 4 procesos parece claro que el valor 4096 para ”mpi-avg” es el más adecuado, ya que es el que obtiene las mejores prestaciones tanto en *accuracy* como en tiempo de ejecución. Para 2 procesos, sucede una cosa similar. Escogeríamos un ”mpi-avg” de 2048, ya que permite obtener el mayor *accuracy* para este número de procesos además de un tiempo de ejecución que en este punto no se reduce al aumentar el ”mpi-avg”. Por último, respecto a las ejecuciones con 8 procesos elegiría un ”mpi-avg” de 512, ya que este es el punto en el que se alcanza un mayor *accuracy* para 8 procesos y a partir de este comienza a descender,

en este punto tendríamos un tiempo de ejecución de 872 segundos, que podría bajar hasta 840 segundos para un "mpi-avg" de 8192 con su consecuente bajada en el *accuracy*. Finalmente, la diferencia entre el *accuracy* más alto y el más bajo presentado por el total de los procesos es de un 5 %.

5.4.4. Comparativa diferentes *epoch*

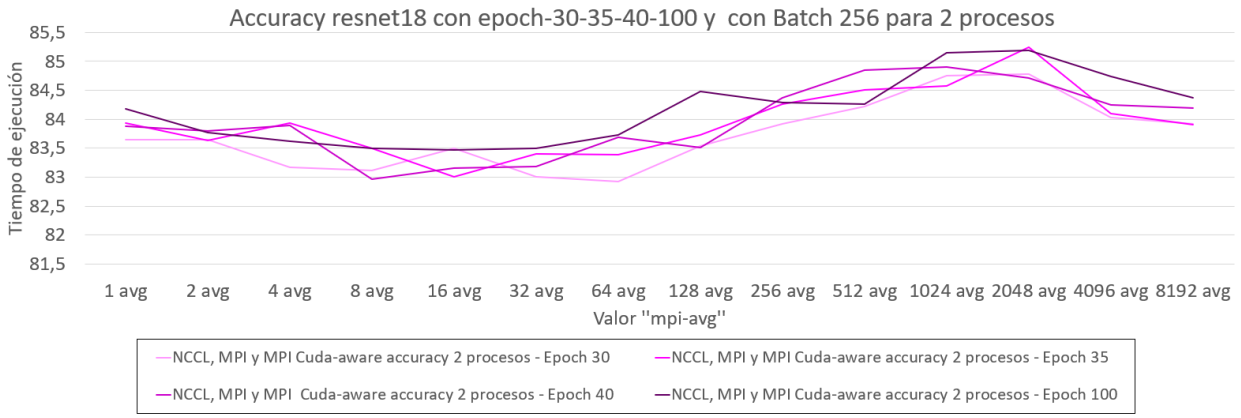


Figura 5.14: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 256 y un *epoch* de 100,40,35 o 30 para 2 procesos

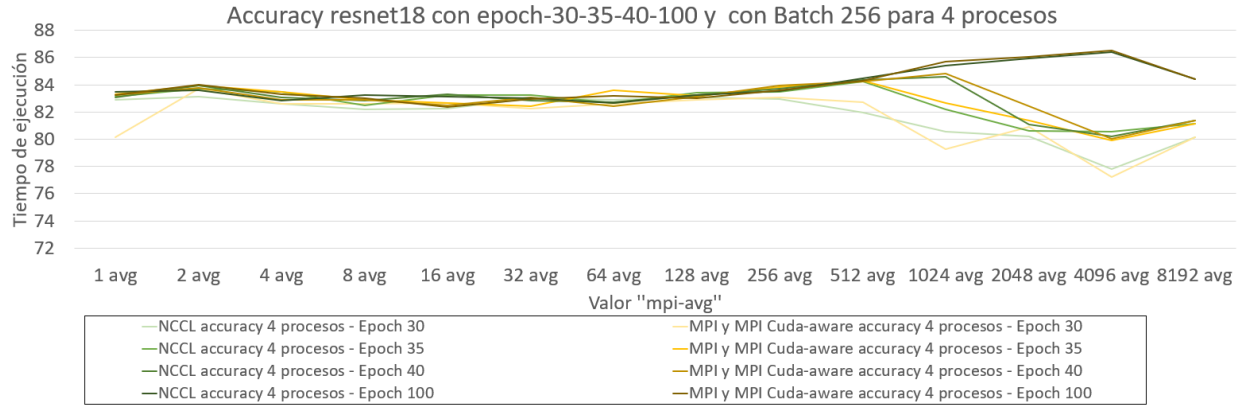


Figura 5.15: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 256 y un *epoch* de 100,40,35 o 30 para 4 procesos

Como ya se ha comentado en el análisis de vgg1, además de las pruebas con un *epoch* de 100 se han hecho otra serie de pruebas con un *epoch* de 30,35 y 40. Estas pruebas se han realizado para las 3 redes analizadas, pero en este apartado únicamente encontraremos aquellas realizadas para resnet18. El resto las podremos encontrar en el Apéndice B, para evitar ser excesivamente repetitivos. Únicamente nos referiremos a las gráficas con estas pruebas para hacer algún apunte específico que reafirme ciertas conclusiones a las que hayamos llegado. Se ha elegido resnet18 para estas pruebas, ya que vgg16 presenta un *accuracy* muy bajo para los *epoch* analizados y vgg1 es una prueba mucho más ligera en la que variará menos el tiempo de ejecución obtenido al variar el *epoch*.

El propósito de realizar estas pruebas es comprobar en que punto será más adecuado aumentar el *epoch* a la hora de obtener unas mejores prestaciones en cuanto *accuracy*, además de comprobar cómo difiere el comportamiento de estas ejecuciones respecto a

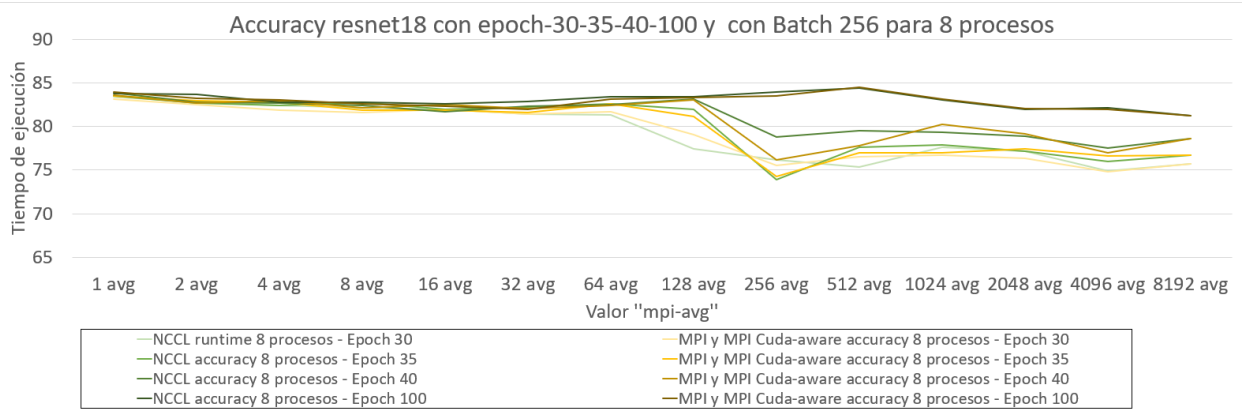


Figura 5.16: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 256 y un *epoch* de 100,40,35 o 30 para 8 procesos

las analizadas para un *epoch* de 100. Para llevar a cabo este análisis se han realizado tres gráficas en las que se muestra el *accuracy* obtenido para ejecuciones con un *epoch* de 30, 35, 40 y 100, realizando una gráfica para las ejecuciones con 2 procesos (Figura 5.14) ,otra para aquellas con 4 procesos (Figura 5.15) y otra para las de 8 procesos (Figura 5.16). Únicamente se muestra el *accuracy* y no el tiempo de ejecución, ya que estas gráficas nos permitirían observar como un mayor número de épocas implica un mayor tiempo de ejecución. Al igual que con las otras redes, en el Apéndice B pueden encontrarse las gráficas de *accuracy* y tiempo de ejecución para resnet18.

En estas gráficas podemos comprobar dos cosas. Principalmente la diferencia que presenta el *accuracy* al variar el *epoch* en una cantidad no muy grande como puede ser de 5 o de 10 o los efectos de una variación más grande de 60-70. Tras observar los efectos de estas variaciones podemos observar varias cosas. Para 2 procesos las ejecuciones con un *epoch* de 100, pese a superar en gran parte de las combinaciones de "mpi-avg" a las ejecuciones con un *epoch* menor, únicamente obtiene una mejora del orden del 1 % en el *accuracy* como máximo, siendo esta mejora menor o incluso presentándose mejores prestaciones para ejecuciones con un *epoch* menor en algunas ejecuciones. Respecto al comportamiento mostrado para los *epoch* de 30, 35 y 40 cabe decir que, al igual que con un *epoch* de 100 la diferencia obtenida es mínima. En este caso la máxima diferencia entre estas 3 ejecuciones es de un 0.5 %. Además entre las ejecuciones con un *epoch* de 35 y 40 no existe casi diferencia, no pudiendo apreciarse claramente si una tiene un mayor *accuracy* que la otra a lo largo de la ejecución cosa que ocurre con un *epoch* de 30, que pese a superar en algún punto a *epochs* mayores sí que se puede apreciar que obtiene un *accuracy* menor.

Respecto a las gráficas para 4 y 8 procesos podemos apuntar varias cosas. Primero que en ambas existe un valor para "mpi-avg" a partir del cual el *accuracy* obtenido entre los distintos *epochs* comienza a divergir significativamente. Este es el punto que habíamos apuntado en el análisis realizado anteriormente. Aquí podemos observar que tal y como sucede para un *epoch* de 100 esto sucede para *epochs* menores, solo que esto es incluso más significativo para estos *epochs* menores. Además, sucede para valores menores de "mpi-avg" y alcanza un *accuracy* menor para 8 procesos. Para comprender por qué sucede esto primero me gustaría apuntar lo que sucede antes de este punto de "disrupción". Tanto para 4 como para 8 procesos antes de este punto la diferencia entre estas ejecuciones con distintos *epochs* era mínima, para 4 procesos era de alrededor de un 1 % y para 8 procesos algo más entre un 1 y un 2 %, en este punto de la ejecución antes de la "disrupción"

incluso sucede que ejecuciones con un menor *epoch* que otra obtengan un mayor *accuracy* (incluso en el caso de *epoch* 100).

Una vez comentado esto procedamos a explicar las causas de este comportamiento. Hemos observado que para unos niveles de comunicación medio-altos ("mpi-avg" bajo) en esta red podemos obtener un *accuracy* medianamente similar para *epochs* como 100 o 30, por lo que las prestaciones se estancan y no se obtiene una mejora significativa al aumentar el número de *epochs*. Esto cambia con valores elevados para "mpi-avg" en los que la comunicación se reduzca, ya que esto causa que las actualizaciones de pesos al entrenar el modelo sean menos precisas causando un descenso de las prestaciones. Un *epoch* alto como el de 100 compensa esta menor precisión al realizar un mayor número de ajustes en los parámetros al recorrer los datos de entrada en un mayor número de ocasiones.

Lo importante de esto es resaltar que en un caso como este en el que las prestaciones en cuanto *accuracy* se aproximan tanto para diferentes *epoch*, teniendo en cuenta que un mayor *epoch* causa un mayor tiempo de ejecución, será más adecuado reducir el número de *epochs* utilizados y aumentar la sincronización (valores de "mpi-avg" más bajos) que ejecutar un *epoch* alto con bajos niveles de sincronización. Si bien esto reduce el tiempo de su ejecución, tambien reduce su *accuracy* de tal manera que con un *epoch* más bajo y unos niveles de sincronización mayores hubiéramos obtenido tanto un tiempo de ejecución menor como un *accuracy* mayor. Habrá que ajustar adecuadamente todos estos parámetros para optimizar al máximo el entrenamiento realizado. Hay que tener en cuenta que para resnet18 las pruebas con diferentes *epochs* la diferencia en cuanto *accuracy* no es muy elevada para ejecuciones con una comunicación elevada, pero existen otras redes en las que bien no existe este estancamiento o en las que se necesita de un *epoch* más elevado para obtener un *accuracy* alto como puede ser el caso vgg16. En este caso, que para un *epoch* de 30-35-40 obtiene un *accuracy* de alrededor del 15 % y con un *epoch* de 100 de un 60-70 %. Esto lo podemos observar en las gráficas del Apéndice B que muestran el *accuracy* y tiempo de ejecución de vgg16 con un *epoch* de 30-35-40.

Un último aspecto que me gustaría comentar respecto a estas gráficas es respecto a que el máximo *accuracy* alcanzado llegue para niveles de sincronización bajos y no para niveles altos con un *epoch* de 100. En cuanto a esto, primero decir que este comportamiento también estaba presente para la red vgg1 y no para la red vgg16 con un *epoch* de 100. Es posible que tanto para vgg1 como para resnet18 se esté produciendo el fenómeno conocido como *overfitting* en el entrenamiento de inteligencia artificial. Este fenómeno consiste en que al sobreentrenar una red neuronal, puede ocurrir que este modelo se ajuste demasiado a las características específicas de los datos de entrada respondiendo incorrectamente con otros datos diferentes. Por lo tanto en las redes vgg16 y resnet18 al realizar los ajustes de una forma más precisa con un "mpi-avg" bajo nos estamos ajustando demasiado a los datos de entrada, mientras que con un "mpi-avg" alto al realizarse los ajustes de forma menos precisa existe una mayor dispersión que permite aumentar el *accuracy*. Eso si, en el caso de resnet18 existe un punto en el que esta menor precisión sí que causa un descenso en el *accuracy*.

5.4.5. Comparativa diferentes *batch*

Por último, se realizará un análisis del impacto del tamaño de un *batch* en las prestaciones de resnet18 variando el tamaño de éste y agrupando estas pruebas en 3 gráficas, una para 2 procesos (Figura 5.17), otra para 4 procesos (Figura 5.18) y una última para 8 procesos (Figura 5.19) al igual que se hizo en la comparativa anterior. Estas pruebas se realizarán con un *epoch* de 35 y los tamaños de *batch* probados serán 128, 256 y 512.

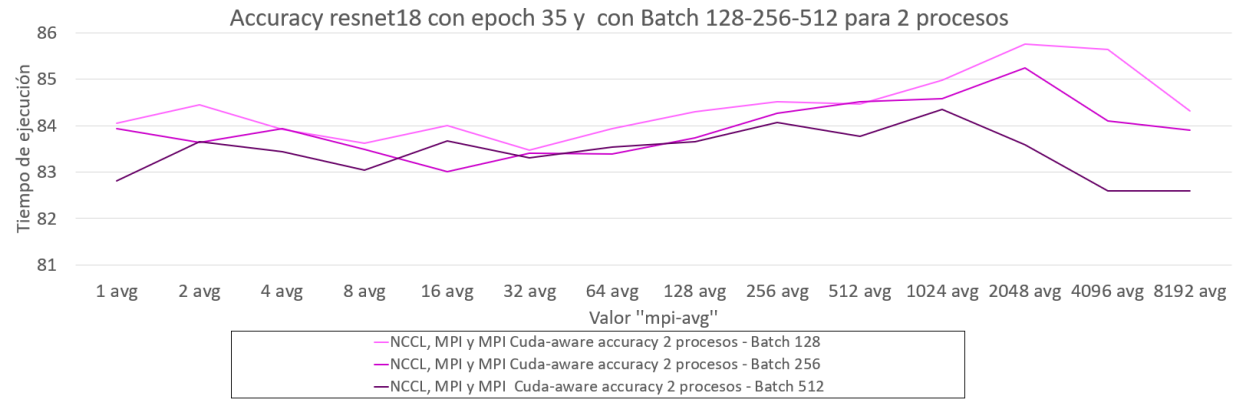


Figura 5.17: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 128,256 o 512 y un *epoch* de 35 para 2 procesos

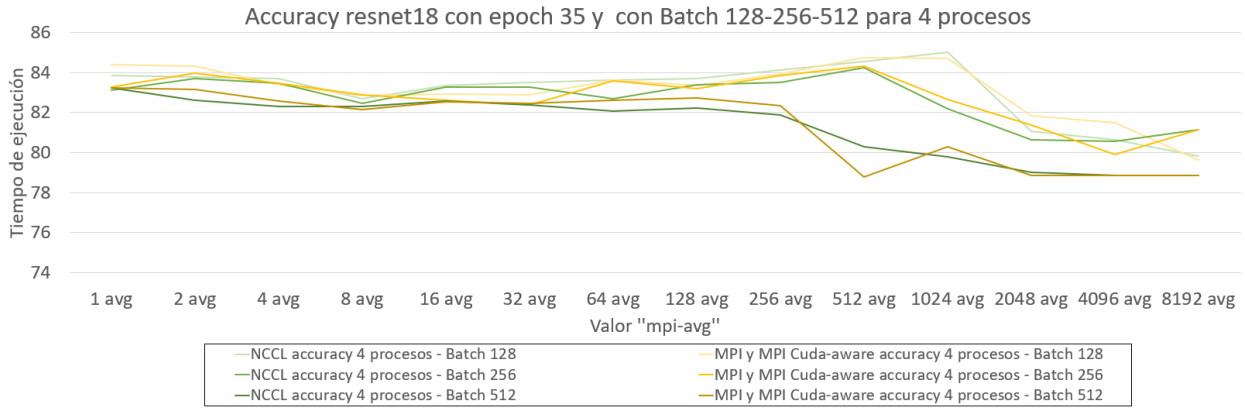


Figura 5.18: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 128,256 o 512 y un *epoch* de 35 para 4 procesos

En primer lugar, cabe decir que al igual que un mayor *epoch* conllevaba un mayor tiempo de ejecución, un menor tamaño de *batch* ocasiona que el tiempo de ejecución obtenido sea mayor al necesitarse de más iteraciones para recorrer el total de los datos de entrada. Por tanto, habrá que analizar el impacto de este tamaño de *batch* en el *accuracy* para comprobar hasta qué punto es conveniente aumentar este parámetro para obtener unas mejores prestaciones en cuanto a tiempo de ejecución y *accuracy*. Este comportamiento se puede apreciar en unas gráficas que se encuentran en el apéndice B. En ellas se muestran los tiempos de ejecución de configuraciones con un tamaño de *batch* de 128,256 o 512 para 2 (Figura B.27), 4 (Figura B.28) y 8 (Figura B.29) procesos.

En cuanto a las ejecuciones con 2 procesos podemos apreciar claramente que a menor es el tamaño de *batch*, mayor es el *accuracy* obtenido además de apreciarse que el máximo *accuracy* obtenido para cada una de estas ejecuciones es para tamaños de "mpi-avg" altos en los que la sincronización es baja. Las máximas variaciones de *accuracy* obtenida entre estos distintos tamaños de *batch* es de un 1-2 %. Tanto para 4 como para 8 procesos sucede algo parecido, salvo para puntos específicos. A mayor tamaño de *batch* menor *accuracy* variando este alrededor de un 1 % para 4 procesos y de un 0.7 % para 8 procesos.

En este caso sí que parece apreciarse que un menor tamaño de *batch* corresponde en la mayoría de casos con un mayor *accuracy*. Eso si, esto no es así para todas las combinaciones y además la mejora obtenida en cuanto a *accuracy* no es muy elevada, si exceptuamos los citados casos en los que la sincronización es demasiado baja y comienza a descen-

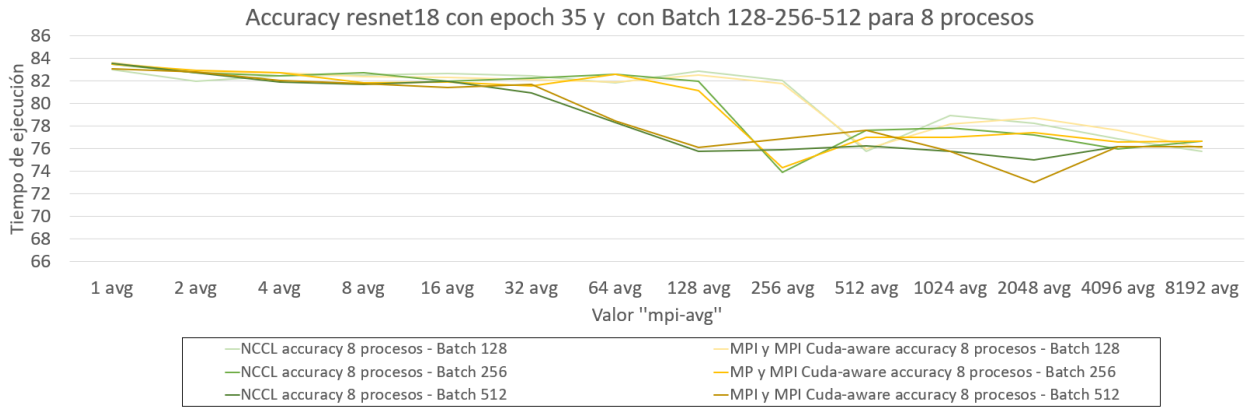


Figura 5.19: Comparativa del *Accuracy* para distintos "mpi-avg" en un entrenamiento asíncrono para el *dataset* CIFAR-10, la red resnet18, con un *batch* de 128,256 o 512 y un *epoch* de 35 para 8 procesos

der significativamente. En general, habrá que variar los valores de este parámetro para buscar la mejor combinación tiempo de ejecución-*accuracy*.

Conclusiones

A lo largo de los distintos análisis llevados a cabo en este capítulo hemos podido extraer distintas conclusiones sobre el efecto de distintos parámetros en el entrenamiento de una red neuronal en HELENNA. Estos parámetros son: el número de procesos. "mpi-avg", la biblioteca utilizada, red neuronal utilizada, el número de *epoch*, el tamaño de *batch*, como se reserva la memoria, el uso de CPU o GPU y qué tipo de entrenamiento realizar.

Acerca de estos análisis, decir que en primer lugar se ha concluido que un entrenamiento asíncrono realizado mediante GPU es el más adecuado entre los probados, ya que las GPU proporcionan unas mejores prestaciones que las CPU al hacer uso del paralelismo de sus muchos núcleos para realizar las operaciones. Por otra parte, un entrenamiento asíncrono permite aprovechar todas las unidades de cálculo disponibles al contrario que *parameter server* que debe utilizar una únicamente para actualizar los parámetros. Además, respecto a un entrenamiento síncrono permite reducir el elevado número de comunicaciones mejorando así las prestaciones obtenidas.

Respecto a la reserva de memoria utilizada, se ha elegido *Malloc*, ya que *MallocHost* presenta unas peores prestaciones al tener que reservarse mucha memoria fija en el *buffer*, por lo que de momento en HELENNA para estas redes probadas no puede aprovecharse la mejora que este tipo de memoria puede llegar a proporcionar. La biblioteca que proporciona un mejor rendimiento entre MPI, MPI CUDA-aware y NCCL es NCCL presentando esta unos tiempos de ejecución significativamente menores que las otras dos bibliotecas. Para valores de "mpi-avg" bajos, MPI y MPI CUDA-aware presentan unas prestaciones similares.

En cuanto al parámetro "mpi-avg", éste proporciona un menor tiempo de ejecución cuanto mayor es su valor, ya que al aumentar éste disminuye el número de comunicaciones realizadas. Esta mejora en las prestaciones va relacionada en parte con muchos otros factores, ya que cuanto mayor es el número de procesos mayor es la mejora que provoca el aumento del valor de este parámetro "mpi-avg". Sin embargo, la mejora obtenida para combinaciones con un "mpi-avg" al aumentar el número de procesos disminuye conforme este va aumentando, siendo la mejora obtenida al pasar de 1 a 2 procesos mayor

que la obtenida al pasar de 2 a 4 y esta mayor que al pasar de 4 a 8. Además, la mejora obtenida al aumentar este parámetro "mpi-avg" se estanca en cierto punto que suele ser mayor (nos referimos a un mayor valor de "mpi-avg") cuanto mayor es el número de procesos. Por último, decir que NCCL pese a obtener mejores prestaciones que las otras 2 alternativas llega un punto que al reducirse el número de invocaciones a primitivas de comunicación iguala sus prestaciones con las de MPI y MPI CUDA-aware.

En lo referente a la red utilizada, el uso de una u otra varía tanto el tiempo de ejecución como el *accuracy*, estando relacionado con esto el número de epoch que necesita cada red para obtener un *accuracy* elevado o si este se estanca no aumentando a partir de cierto punto. Respecto a este estancamiento también se ha visto que se produce un *overfitting* que causa un estancamiento del *accuracy* obtenido. Una menor comunicación entre procesos al causar que se ajusten los parámetros de forma menos precisa causa que en ciertos puntos se obtenga un mejor *accuracy* que con una mayor comunicación. Hablando de los efectos en el *accuracy* que causa "mpi-avg", para valores muy altos de este parámetro se produce una disminución en el *accuracy* obtenido que es más notable conforme se aumenta el número de procesos. Este comportamiento únicamente no se da en configuraciones con un *epoch* alto y con un *accuracy* estancado. Respecto a la variabilidad del *accuracy* obtenido, cabe decir que además del *overfitting*, esta puede deberse la distribución del *dataset* o la variabilidad estocástica del proceso de entrenamiento.

En cuanto al *epoch*, en ciertos casos un aumento de este parámetro no implica necesariamente un mayor *accuracy*, especialmente para valores de "mpi-avg" bajos. Teniendo en cuenta que al aumentar el número de *epochs* se incrementa el tiempo de ejecución, podrán realizarse ejecuciones con un *epoch* más reducido que otras que mantengan o se acerquen al *accuracy* obtenido por ejecuciones con un *epoch* mayor. El caso del tamaño de *batch* es parecido, ya que al disminuir este valor aumenta el tiempo de ejecución. En las pruebas estudiadas, parece tener mayor impacto en la obtención de un *accuracy* más elevado que el que tenía el número de *epochs* para redes como vgg1 o resnet18, por lo que en este caso habría que tener en cuenta una combinación especifica para concluir si en ese caso un tamaño de *batch* más o menos elevado es conveniente.

En conclusión, el ajustar cada uno de estos parámetros correctamente nos permite construir un entrenamiento distribuido mucho más eficiente y que en algunos casos además de mejorar el tiempo de ejecución también pueda mejorar el *accuracy* obtenido.

CAPÍTULO 6
Conclusiones

A partir de los distintos análisis llevados a cabo a lo largo del desarrollo de este trabajo se pueden extraer las siguientes conclusiones:

- Para primitivas de comunicación en las que el tamaño del mensaje enviado sea pequeño o mediano MPI, CUDA-aware junto a una reserva de memoria a través de *cudaMallocHost()* es la opción que proporciona un mayor ancho de banda. Mientras que para tamaños de mensaje grandes o muy grandes NCCL junto a una reserva de memoria a través de *cudaMalloc()* o *cudaMallocManaged()* es la opción que proporciona un mayor ancho de banda.

- La reserva de memoria mediante *Malloc()* y *MallocManaged()* presentan prestaciones similares.

- El ancho de banda de NCCL para tamaños de mensaje muy grandes cae en picado para 2 procesos a partir de cierto tamaño. Sin embargo para 4 y 8 procesos se produce una estabilización del ancho de banda a partir de cierto tamaño debido a las características propias de NCCL.

- NCCL experimenta unas prestaciones peores o iguales con la reserva de memoria *cudaMallocHost()* respecto a los otros dos tipos de reserva de memoria probados.

- Entre los entrenamientos síncronos, asíncronos y *parameter server* los entrenamientos asíncronos son aquellos que nos permiten obtener unas mayores prestaciones.

- Al reducir la frecuencia de sincronización en un entrenamiento asíncrono a través del parámetro "mpi-avg" (en la herramienta HELENNA) podemos reducir el uso de las comunicaciones en dicho entrenamiento, mejorando el tiempo de ejecución. Esta reducción en el número de comunicaciones además de causar una mejora en las prestaciones obtenidas en cuanto al tiempo de ejecución, también causa que en muchos casos al disminuir esta sincronización, la precisión o *accuracy* del modelo disminuya respecto a un entrenamiento con una mayor sincronización. Esta disminución del *accuracy* aumenta conforme mayor es el número de procesos.

- Aunque lo normal es que para una sincronización muy baja en un sistema con un número de procesos medianamente alto (4-8) disminuya el *accuracy* existen casos en los que el modelo experimenta *overfitting* que causa un estancamiento en las prestaciones y que con una menor sincronización obtiene un mayor *accuracy* que en ejecuciones con una mayor sincronización. Esto se debe a la menor precisión en los ajustes de los pesos del modelo que proporciona mayor varianza al modelo.

- Para un entrenamiento de redes neuronales en HELENNA la librería NCCL proporciona unas mejores prestaciones a la hora de realizar entrenamientos distribuidos de redes neuronales. Esta mejora en las prestaciones es principalmente observable para valores de "mpi-avg" bajos en los que hay una alta sincronización. Para valores más altos de "mpi-avg" en los que la sincronización se reduce estas diferencias entre NCCL y las otras bibliotecas se van reduciendo hasta desaparecer. Pese a esto NCCL, es la alternativa que proporciona mejores prestaciones si tenemos en cuenta todas las posibles combinaciones por lo que se recomienda el uso de esta biblioteca para la versión de HELENNA actual.
- Las pruebas realizadas en HELENNA fueron realizadas para *Malloc()*, ya que *MallocHost()* presentaba unas prestaciones deficientes respecto a *Malloc()* y *MallocManaged()* y entre estas dos se hace uso de *Malloc()*, ya que presentan prestaciones similares y lo más habitual es emplear *Malloc()*. Estas malas prestaciones para *MallocHost()* se deben a que, en las redes probadas, el *buffer* creado por *MallocHost()* (o *Malloc()* en su caso) reserva una cantidad de memoria fija demasiado grande que lastra las prestaciones. Como trabajo futuro, se plantea buscar alternativas que permitan aprovechar las características de *MallocHost()* junto a MPI CUDA-aware observadas en el capítulo 4. Para esto se plantea dividir los mensajes enviados para que sean de un tamaño menor y así no se vean lastradas sus prestaciones o fijen un tamaño a partir del cual se reserve mediante la memoria *MallocHost()* y MPI CUDA-aware en vez de con NCCL y *Malloc()*.
- El número de *epochs* utilizado en un entrenamiento es un hiperparámetro importante que nos permite aumentar el *accuracy* obtenido con el coste de aumentar el tiempo de ejecución. Existen ciertos valores para "mpi-avg" en los cuales el aumento del número de *epochs* no ocasiona que se obtenga un mayor *accuracy* por lo que reduciendo este valor según las circunstancias podremos obtener un *accuracy* similar con un tiempo de ejecución menor. Esto dependerá de cada caso, ya que al igual que una variación en el número de *epochs* puede no presentar un impacto muy elevado en el *accuracy* obtenido puede ocurrir que al aumentar éste se produzca un incremento muy significativo en el *accuracy* presentado. El tamaño de *batch* tiene un comportamiento similar a este parámetro solo que en este caso se ha detectado una mayor variación en el *accuracy* presentado relacionada con el tamaño de *batch*. Por tanto, tanto el número de *epochs* como el tamaño de *batch* serán dos parámetros que al realizar un entrenamiento distribuido tendremos que tener en cuenta junto a la sincronización aplicada a estos observando así si es conveniente aplicar una mayor o menor sincronización como número de *epochs* y de *batch*.

De todas estas conclusiones podemos decir finalmente que existe una serie de parámetros en la ejecución de un entrenamiento distribuido que nos permitirán mejorar la eficiencia de este, entre estos parámetros el uso de una biblioteca como NCCL para implementar las primitivas de comunicación así como un modelo asíncrono que nos permita reducir el número de comunicaciones y nos permiten aprovechar las prestaciones proporcionadas por el número de procesos ejecutados. Además de estos parámetros que nos permiten optimizar el entrenamiento distribuido, existen otros parámetros que, si se fijan correctamente, también nos permiten obtener una mejora en las prestaciones obtenidas. Estos parámetros son el número de *epochs* y tamaño de batch y el tipo de reserva de memoria.

Como una vía para trabajo futuro además del ya planteado para aprovechar las prestaciones de MPI CUDA-aware junto a *MallocHost()* se plantea tanto el estudio de diferentes *datasets* y redes más exigentes computacionalmente como pueden ser el *dataset* Imagenet y la red Alexnet así como el estudio del impacto de otros parámetros en las

prestaciones que en este trabajo no se han tratado como pueden ser el *learning rate* o el *momentum*.

6.1 Relación del trabajo desarrollado con los estudios cursados

Para la realización del trabajo han sido de especial relevancia los contenidos tratados en las siguientes asignaturas:

- **Computación Paralela (CPA) y Lenguajes y Entornos de Programación Paralela (LPP)**: El objetivo de CPA es realizar una introducción a los modelos de programación paralela, estudiándose para ello los modelos de memoria compartida y distribuida más extendidos. En la asignatura de LPP, a partir de los conocimientos adquiridos en CPA se profundizará más en estas tecnologías. En la realización de este trabajo se han aprovechado los conocimientos de modelos de memoria distribuida, entre estos modelos hay que destacar los conocimientos en el uso de MPI.
- **Arquitectura e Ingeniería de Computadores (AIC) y Arquitecturas Avanzadas (AAV)**: Estas asignaturas tienen dos objetivos principales, proporcionar un conocimiento sólido sobre el funcionamiento de los procesadores actuales y estudiar el funcionamiento de las redes de interconexión y su impacto en las prestaciones. Los conceptos adquiridos en estas asignaturas han sentado las bases para conocer y comprender el funcionamiento de las GPU, los sistemas compuestos por múltiples nodos de cálculo y su aplicación a la realización de entrenamiento distribuido de redes neuronales. En AAV, al ser una asignatura más avanzada se cubren ciertos conocimientos adicionales respecto a AIC.
- **Sistemas Inteligentes (SIN)**: El objetivo de esta asignatura es introducir a los sistemas inteligentes desde una orientación práctica. Se desarrollan los contenidos alrededor del aprendizaje automático y la representación del conocimiento y búsqueda. Estos conocimientos nos han permitido sentar las bases para comprender el funcionamiento de los algoritmos de Inteligencia Artificial y más en concreto de las redes neuronales y *Deep Learning*.

Bibliografía

[1] *"A Brief Introduction to Distributed Training with Gradient Descent"*. URL: https://medium.com/@shivvidhyut/a-brief-introduction-to-distributed-training-with-gradient-descent-a4ba9faefcea.

[2] *"Activation Functions in Neural Networks"*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[3] *"Advantages and disadvantages of each activation function in deep learning"*. URL: https://www.programmersought.com/article/2709872916/.

[4] *"An Introduction to CUDA-Aware MPI"*. URL: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.

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

[6] *"Computer AI passes Turing test in 'world first"*. URL: https://www.bbc.com/news/technology-27762088#:~:text=The%2065%2Dyear%2Dold%20Turing,London%20that%20it%20was%20human.&text=A%20glimpse%20at%20one%20of%20the%20conversations..

[7] *"CuBLAS Developer zone"*. URL: https://developer.nvidia.com/cublas.

[8] *"CUDA:cudaMalloc vs cudaMallocHost"*. URL: ttps://www.programmersought.com/article/36327486720/.

[9] *"cuDNN NVIDIA Developer zone"*. URL: https://developer.nvidia.com/cudnn.

[10] *"Deep Learning.Introducción práctica con keras(primera parte)"*. URL: https://torres.ai/deep-learning-inteligencia-artificial-keras/.

[11] ""Distributed Deep Learning training: Model and Data Parallelism in Tensorflow"". En: (). URL: https://theaisummer.com/distributed-training/.

[12] *"Do we really need GPU for Deep Learning? - CPU vs GPU"*. URL: https://medium.com/@shachishah.ce/do-we-really-need-gpu-for-deep-learning-47042c02efe2.

[13] *"Documentacion de CUDA"*. URL: https://docs.nvidia.com/cuda/.

[14] *"Documentación slurm workload managed"*. URL: https://slurm.schedmd.com/documentation.html.

[15] *"Ethernet"*. URL: https://www.ecured.cu/Ethernet.

[16] *"Everything You Need to Know About GPU Architecture and How It Has Evolved"*. URL: https://blog.cherryservers.com/everything-you-need-to-know-about-gpu-architecture.

[17] *"GPU vs CPU Computing: What to choose?"* URL: https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4.

[18] *"GPU vs CPU: What Are The Key Differences?"* URL: https://blog.cherryservers.com/gpu-vs-cpu-what-are-the-key-differences.

[19] *"H20 AI web"*. URL: https://www.h2o.ai/es/.

[20] *"ImageNet"*. URL: https://www.image-net.org/update-mar-11-2021.php.

[21] *"Infiniband"*. URL: https://www.ecured.cu/InfiniBand.

[22] *"Intel® oneAPI Math Kernel Library "*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html#gs.52fw9s.

[23] *"Long short-term memory"*. URL: https://en.wikipedia.org/wiki/Long_short-term_memory#History.

[24] *"MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE"*. URL: https://mvapich.cse.ohio-state.edu/.

[25] *"Neurociencia computacional"*. URL: https://es.wikipedia.org/wiki/Neurociencia_computacional.

[26] *"NVIDIA NCCL Developer zone"*. URL: https://developer.nvidia.com/nccl.

[27] *"Open MPI:Open Source High Performance Computing"*. URL: https://www.open-mpi.org/.

[28] *"Repositorio ibdump"*. URL: https://github.com/Mellanox/ibdump.

[29] *"TAU COMMANDER Intuitive Performance Engineering "*. URL: http://taucommander.paratools.com/.

[30] *"Tcpdump.org"*. URL: https://www.tcpdump.org/.

[31] *"TensorFlow.org "*. URL: https://www.tensorflow.org/.

[32] *"The difference and usage of CUDA-global memory"*. URL: https://www.programmersought.com/article/60145534441/.

[33] *"The Evolution of Deep Learning"*. URL: https://towardsdatascience.com/the-deep-history-of-deep-learning-3bebeb810fb2.

[34] *"The Future is Hybrid Trends in Computing Hardware"*. URL: https://medium.com/@shivvidhyut/a-brief-introduction-to-distributed-training-with-gradient-descent-a4ba9faefcea.

[35] *"The History of Artificial Intelligence"*. URL: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/.

[36] *"THE MNIST DATABASE of handwritten digits"*. URL: http://yann.lecun.com/exdb/mnist/.

[37] *"Todo lo que Necesitas Saber sobre el Descenso del Gradiente Aplicado a Redes Neuronales"*. URL: https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78.

[38] *'Introduction to InfiniBand"*. URL: https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[39] *'Wireshark.org"*. URL: https://www.wireshark.org/.

[40] *"Pagina web tutorial MPI"*. URL: https://mpitutorial.com/.

[41] Ammar Ahmad Awan y col. "Optimized large-message broadcast for deep learning workloads: MPI, MPI+ NCCL, or NCCL2?" En: *parallel computing* 85 (2019), págs. 141-152.

[42] MPI Furum. *"Document for a standard message passing interface"*. Inf. téc. Tech Rep. CS-93-214, University of Tennessee, 1993.

[43] John L Hennessy y David A Patterson. *"Computer architecture: a quantitative approach"*. Elsevier, 2011.

[44] Shijie Joaquin Obregon Cobo. "Computación Heterogénea (Potencia GPU+CPU)". En: *ResearchGate* (2013).

[45] Matthias Langer y col. "Distributed training of deep learning models: A taxonomic perspective". En: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), págs. 2802-2818.

[46] X. Li y col. "Chapter 4 - Deep Learning and Its Parallelization". En: *Big Data*. Morgan Kaufmann, 2016, págs. 95-118.

[47] Dejiao Niu y col. "The Asynchronous Training Algorithm Based on Sampling and Mean Fusion for Distributed RNN". En: *IEEE Access* 8 (2019), págs. 62439-62447.

[48] Shuo Ouyang y col. "Communication optimization strategies for distributed deep learning: A survey". En: *arXiv preprint arXiv:2003.03009* (2020).

[49] David A Patterson y John L Hennessy. *"Estructura y diseño de computadores"*. Reverte, 2018.

[50] Sreeram Potluri y col. "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs". En: *2013 42nd International Conference on Parallel Processing*. IEEE. 2013, págs. 80-89.

[51] Yufei Ren y col. "irdma: Efficient use of rdma in distributed deep learning systems". En: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2017, págs. 231-238.

[52] Piero Scaruffi. *"Intelligence is not Artificial - Expanded Edition: A History of Artificial Intelligence and Why the Singularity is not Coming any Time Soon"*. CreateSpace Independent Publishing Platform, 2018.

[53] Alan Turing y J Haugeland. *"Computing machinery and intelligence"*. MIT Press Cambridge, MA, 1950.

[54] Michael Wooldridge. *"A Brief History of Artificial Intelligence: What It Is, Where We Are, and Where We Are Going"*. Macmillan USA, 2021.

APÉNDICE A
Redes Neuronales

En este apartado se ilustran algunas de las redes neuronales presentes en HELENNA, exponiéndose el código que las compone, mostrándose así qué funciones de activación y capas están presentes en cada red neuronal.

Un primer apunte a la hora de comprender el código que compone estas redes neuronales es que en cada linea primeramente se define el nombre de la capa o función de activación, posteriormente cual se va a utilizar y por último los parámetros de esta que podemos encontrarlos en el apartado 2.6 de esta memoria donde tratamos las distintas capas y funciones de activación de HELENNA con sus parámetros. Como ya hemos dicho el segundo parámetro es el que indica el tipo de función de activación o capa utilizado para cada capa sus significados son:

Parametro	Significado
conv	capa convulocional
maxpooling	capa maxpooling
fc	capa *fully connected*
batch_normalization	capa *Batch Normalization*
dropout	capa *Dropout*
relu	función de activación ReLU
softmax	función de activación Softmax
split	permite que la salida de una capa vaya a varias capas
add	suma la salida de dos capas previas

Tabla A.1: Significado tienen los distintos posibles parámetros de una red neuronal

network and model summary

layer	name	layer type	neurons	params	in_layer	Format	configuration	memory
0	input	input layer	3072	0		IHWB	inputs 3072, outputs 3072	0.00 GB
1	conv1_0	convolutional	32768	896	(prev)	IHWB	IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32	0.07 GB
2	relu1_0	relu	32768	0	(prev)	IHWB		0.05 GB
3	conv2_0	convolutional	32768	9248	(prev)	IHWB	IN: 32x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 32x32x32	0.33 GB
4	relu2_0	relu	32768	0	(prev)	IHWB		0.05 GB
5	maxp1_0	maxpooling	8192	0	(prev)	IHWB	IN: 32x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 32x16x16	0.02 GB
6	fc1_0	fully connected	128	1048704	(prev)	IHWB	inputs 8192, outputs 128	0.02 GB
7	relu3_0	relu	128	0	(prev)	IHWB		0.00 GB
8	fc2_0	fully connected	10	1290	(prev)	IHWB	inputs 128, outputs 10	0.00 GB
9	sfmax1_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	139540	1060138			Memory (no layers): 0.00 GB	0.54 GB

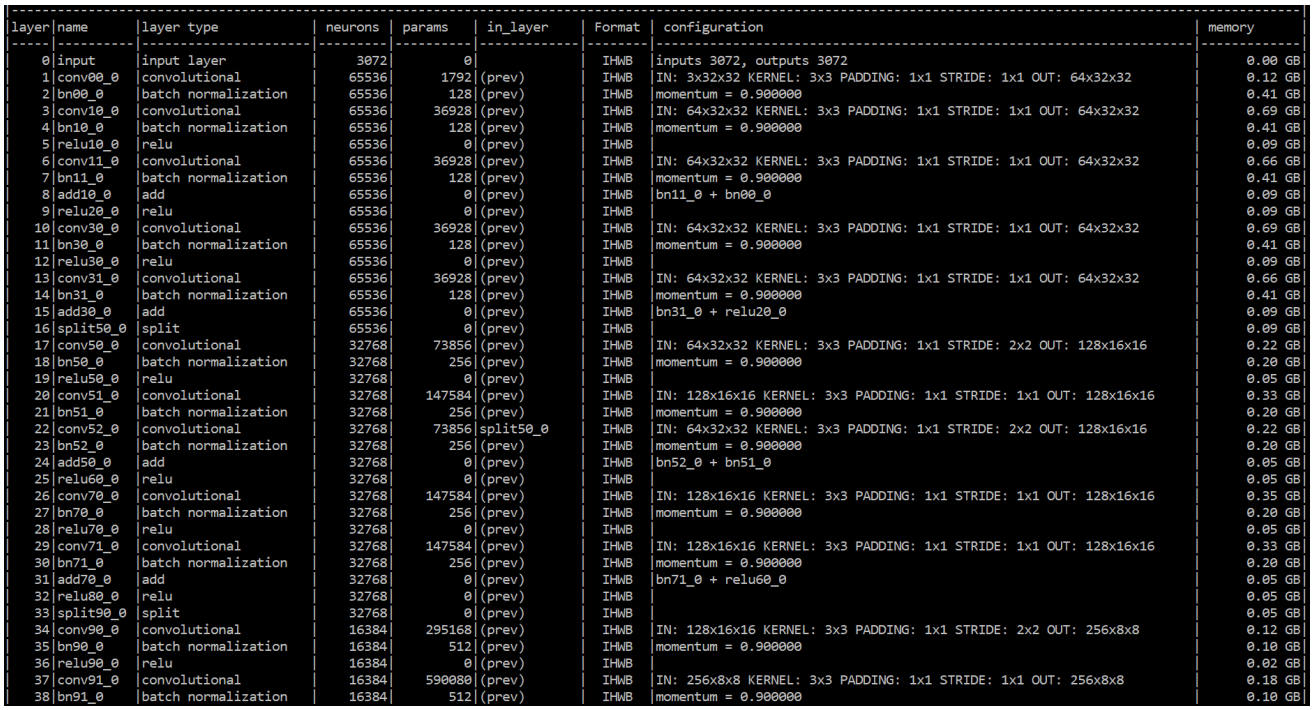
Figura A.1: Capas y funciones de activación que conforman la red vgg1

layer	name	layer type	neurons	params	in_layer	Format	configuration	memory
0	input	input layer	3072	0		IHWB	inputs 3072, outputs 3072	0.00 GB
1	conv1_0	convolutional	65536	1792	(prev)	IHWB	IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.12 GB
2	relu1_0	relu	65536	0	(prev)	IHWB		0.09 GB
3	bn1_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.000000	0.41 GB
4	do1_0	dropout	65536	0	(prev)	IHWB	RATE: 0.300000	0.13 GB
5	conv2_0	convolutional	65536	36928	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.66 GB
6	relu2_0	relu	65536	0	(prev)	IHWB		0.09 GB
7	bn2_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.000000	0.41 GB
8	maxp1_0	maxpooling	16384	0	(prev)	IHWB	IN: 64x32x32 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 64x16x16	0.03 GB
9	conv3_0	convolutional	32768	73856	(prev)	IHWB	IN: 64x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x16x16	0.19 GB
10	relu3_0	relu	32768	0	(prev)	IHWB		0.05 GB
11	bn3_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.000000	0.20 GB
12	do2_0	dropout	32768	0	(prev)	IHWB	RATE: 0.400000	0.06 GB
13	conv4_0	convolutional	32768	147584	(prev)	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x16x16	0.33 GB
14	relu4_0	relu	32768	0	(prev)	IHWB		0.05 GB
15	bn4_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.000000	0.20 GB
16	maxp2_0	maxpooling	8192	0	(prev)	IHWB	IN: 128x16x16 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 128x8x8	0.02 GB
17	conv5_0	convolutional	16384	295168	(prev)	IHWB	IN: 128x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.10 GB
18	relu5_0	relu	16384	0	(prev)	IHWB		0.02 GB
19	bn5_0	batch normalization	16384	512	(prev)	IHWB	momentum = 1.000000	0.10 GB
20	do3_0	dropout	16384	0	(prev)	IHWB	RATE: 0.400000	0.03 GB
21	conv6_0	convolutional	16384	590080	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.18 GB
22	relu6_0	relu	16384	0	(prev)	IHWB		0.02 GB
23	bn4_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.000000	0.10 GB
24	do4_0	dropout	16384	0	(prev)	IHWB	RATE: 0.400000	0.03 GB
25	conv7_0	convolutional	16384	590080	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.18 GB
26	relu7_0	relu	16384	0	(prev)	IHWB		0.02 GB
27	bn7_0	batch normalization	16384	512	(prev)	IHWB	momentum = 1.000000	0.10 GB
28	maxp3_0	maxpooling	4096	0	(prev)	IHWB	IN: 256x8x8 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 256x4x4	0.01 GB
29	conv8_0	convolutional	8192	1180160	(prev)	IHWB	IN: 256x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.07 GB
30	relu8_0	relu	8192	0	(prev)	IHWB		0.01 GB
31	bn8_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 1.000000	0.05 GB
32	do5_0	dropout	8192	0	(prev)	IHWB	RATE: 0.400000	0.02 GB
33	conv9_0	convolutional	8192	2359808	(prev)	IHWB	IN: 512x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.13 GB
34	relu9_0	relu	8192	0	(prev)	IHWB		0.01 GB
35	bn9_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.000000	0.05 GB
36	do6_0	dropout	8192	0	(prev)	IHWB	RATE: 0.400000	0.02 GB
37	conv10_0	convolutional	8192	2359808	(prev)	IHWB	IN: 512x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.13 GB
38	relu10_0	relu	8192	0	(prev)	IHWB		0.01 GB

Figura A.2: Capas y funciones de activación que conforman la red vgg16 (Parte 1)

39	bn10_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 1.000000	0.05 GB
40	maxp4_0	maxpooling	2048	0	(prev)	IHWB	IN: 512x4x4 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 512x2x2	0.00 GB
41	conv11_0	convolutional	2048	2359808	(prev)	IHWB	IN: 512x2x2 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x2x2	0.06 GB
42	relu11_0	relu	2048	0	(prev)	IHWB		0.00 GB
43	bn11_0	batch normalization	2048	1024	(prev)	IHWB	momentum = 1.000000	0.01 GB
44	do7_0	dropout	2048	0	(prev)	IHWB	RATE: 0.400000	0.00 GB
45	conv12_0	convolutional	2048	2359808	(prev)	IHWB	IN: 512x2x2 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x2x2	0.06 GB
46	relu12_0	relu	2048	0	(prev)	IHWB		0.00 GB
47	bn12_0	batch normalization	2048	1024	(prev)	IHWB	momentum = 0.000000	0.01 GB
48	do8_0	dropout	2048	0	(prev)	IHWB	RATE: 0.400000	0.00 GB
49	conv13_0	convolutional	2048	2359808	(prev)	IHWB	IN: 512x2x2 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x2x2	0.06 GB
50	relu13_0	relu	2048	0	(prev)	IHWB		0.00 GB
51	bn13_0	batch normalization	2048	1024	(prev)	IHWB	momentum = 1.000000	0.01 GB
52	maxp3_0	maxpooling	512	0	(prev)	IHWB	IN: 512x2x2 KERNEL 2x2 PADDING: 0x0 STRIDE 2x2 OUT: 512x1x1	0.00 GB
53	do9_0	dropout	512	0	(prev)	IHWB	RATE: 0.500000	0.00 GB
54	fc1_0	fully connected	512	262656	(prev)	IHWB	inputs 512, outputs 512	0.01 GB
55	relu14_0	relu	512	0	(prev)	IHWB		0.00 GB
56	bn14_0	batch normalization	512	1024	(prev)	IHWB	momentum = 2.000000	0.00 GB
57	do10_0	dropout	512	0	(prev)	IHWB	RATE: 0.500000	0.00 GB
58	fc2_0	fully connected	10	5130	(prev)	IHWB	inputs 512, outputs 10	0.00 GB
59	smax_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	1014804	14991946			Memory (no layers): 0.00 GB	4.74 GB

Figura A.3: Capas y funciones de activación que conforman la red vgg16 (Parte 2)



layer	name	layer type	neurons	params	in_layer	Format	configuration	memory
0	input	input layer	3072	0		IHWB	inputs 3072, outputs 3072	0.00 GB
1	conv00_0	convolutional	65536	1792	(prev)	IHWB	IN: 3x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.12 GB
2	bn00_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.900000	0.41 GB
3	conv10_0	convolutional	65536	36928	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.69 GB
4	bn10_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.900000	0.41 GB
5	relu10_0	relu	65536	0	(prev)	IHWB		0.09 GB
6	conv11_0	convolutional	65536	36928	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.66 GB
7	bn11_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.900000	0.41 GB
8	add10_0	add	65536	0	(prev)	IHWB	bn11_0 + bn00_0	0.09 GB
9	relu20_0	relu	65536	0	(prev)	IHWB		0.09 GB
10	conv30_0	convolutional	65536	36928	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.69 GB
11	bn30_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.900000	0.41 GB
12	relu30_0	relu	65536	0	(prev)	IHWB		0.09 GB
13	conv31_0	convolutional	65536	36928	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 64x32x32	0.66 GB
14	bn31_0	batch normalization	65536	128	(prev)	IHWB	momentum = 0.900000	0.41 GB
15	add30_0	add	65536	0	(prev)	IHWB	bn31_0 + relu20_0	0.09 GB
16	split50_0	split	65536	0	(prev)	IHWB		0.09 GB
17	conv50_0	convolutional	32768	73856	(prev)	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 128x16x16	0.22 GB
18	bn50_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.900000	0.20 GB
19	relu50_0	relu	32768	0	(prev)	IHWB		0.05 GB
20	conv51_0	convolutional	32768	147584	(prev)	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x16x16	0.33 GB
21	bn51_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.900000	0.20 GB
22	conv52_0	convolutional	32768	73856	split50_0	IHWB	IN: 64x32x32 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 128x16x16	0.22 GB
23	bn52_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.900000	0.20 GB
24	add50_0	add	32768	0	(prev)	IHWB	bn52_0 + bn51_0	0.05 GB
25	relu60_0	relu	32768	0	(prev)	IHWB		0.05 GB
26	conv70_0	convolutional	32768	147584	(prev)	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x16x16	0.35 GB
27	bn70_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.900000	0.20 GB
28	relu70_0	relu	32768	0	(prev)	IHWB		0.05 GB
29	conv71_0	convolutional	32768	147584	(prev)	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 128x16x16	0.33 GB
30	bn71_0	batch normalization	32768	256	(prev)	IHWB	momentum = 0.900000	0.20 GB
31	add70_0	add	32768	0	(prev)	IHWB	bn71_0 + relu60_0	0.05 GB
32	relu80_0	relu	32768	0	(prev)	IHWB		0.05 GB
33	split90_0	split	32768	0	(prev)	IHWB		0.05 GB
34	conv90_0	convolutional	16384	295168	(prev)	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 256x8x8	0.12 GB
35	bn90_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.900000	0.10 GB
36	relu90_0	relu	16384	0	(prev)	IHWB		0.02 GB
37	conv91_0	convolutional	16384	590080	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.18 GB
38	bn91_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.900000	0.10 GB

Figura A.4: Capas y funciones de activación que conforman la red resnet18 (Parte 1)

39	conv92_0	convolutional	16384	295168	split90_0	IHWB	IN: 128x16x16 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 256x8x8	0.12 GB
40	bn92_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.900000	0.10 GB
41	add90_0	add	16384	0	(prev)	IHWB	bn92_0 + bn91_0	0.02 GB
42	relu100_0	relu	16384	0	(prev)	IHWB		0.02 GB
43	conv110_0	convolutional	16384	590080	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.18 GB
44	bn110_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.900000	0.10 GB
45	relu110_0	relu	16384	0	(prev)	IHWB		0.02 GB
46	conv111_0	convolutional	16384	590080	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 256x8x8	0.18 GB
47	bn111_0	batch normalization	16384	512	(prev)	IHWB	momentum = 0.900000	0.10 GB
48	add110_0	add	16384	0	(prev)	IHWB	bn111_0 + relu100_0	0.02 GB
49	relu120_0	relu	16384	0	(prev)	IHWB		0.02 GB
50	split130_0	split	16384	0	(prev)	IHWB		0.02 GB
51	conv130_0	convolutional	8192	1180160	(prev)	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 512x4x4	0.08 GB
52	bn130_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.900000	0.05 GB
53	relu130_0	relu	8192	0	(prev)	IHWB		0.01 GB
54	conv131_0	convolutional	8192	2359808	(prev)	IHWB	IN: 512x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.13 GB
55	bn131_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.900000	0.05 GB
56	conv312_0	convolutional	8192	1180160	split130_0	IHWB	IN: 256x8x8 KERNEL: 3x3 PADDING: 1x1 STRIDE: 2x2 OUT: 512x4x4	0.08 GB
57	bn132_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.900000	0.05 GB
58	add130_0	add	8192	0	(prev)	IHWB	bn132_0 + bn131_0	0.01 GB
59	relu140_0	relu	8192	0	(prev)	IHWB		0.01 GB
60	conv150_0	convolutional	8192	2359808	(prev)	IHWB	IN: 512x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.13 GB
61	bn150_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.900000	0.05 GB
62	relu150_0	relu	8192	0	(prev)	IHWB		0.01 GB
63	conv151_0	convolutional	8192	2359808	(prev)	IHWB	IN: 512x4x4 KERNEL: 3x3 PADDING: 1x1 STRIDE: 1x1 OUT: 512x4x4	0.13 GB
64	bn151_0	batch normalization	8192	1024	(prev)	IHWB	momentum = 0.900000	0.05 GB
65	add150_0	add	8192	0	(prev)	IHWB	bn151_0 + relu140_0	0.01 GB
66	relu160_0	relu	8192	0	(prev)	IHWB		0.01 GB
67	pool160_0	maxpooling	512	0	(prev)	IHWB	IN: 512x4x4 KERNEL 4x4 PADDING: 0x0 STRIDE 1x1 OUT: 512x1x1	0.00 GB
68	fc170_0	fully connected	512	262656	(prev)	IHWB	inputs 512, outputs 512	0.01 GB
69	relu170_0	relu	512	0	(prev)	IHWB		0.00 GB
70	fc2_0	fully connected	10	5130	(prev)	IHWB	inputs 512, outputs 10	0.00 GB
71	softmax_0	softmax	10	0	(prev)	IHWB		0.00 GB
		TOTAL	2016788	12817674			Memory (no layers): 0.00 GB	10.52 GB

Figura A.5: Capas y funciones de activación que conforman la red resnet18 (Parte 2

APÉNDICE B
Gráficas adicionales

B.1 Leyendas TAU

En este apartado encontramos las figuras B.1 (”mpi-avg” 64), B.2 (”mpi-avg” 2048), B.3 (Malloc), B.4 (MallocManaged), B.5 (MallocHost).

B.2 Gráficas vgg1

.

B.2.1. Comparativa MPI, MPI CUDA-aware

En este apartado encontramos las figuras B.6 (2 procesos), B.7 (4 procesos), B.8 (8 procesos).

B.2.2. Gráficas *accuracy*

En este apartado encontramos las figuras B.11 (*Epoch* 30), B.10 (*Epoch* 35), B.9 (*Epoch* 40).

B.2.3. Gráficas tiempo de ejecución

En este apartado encontramos las figuras B.14 (*Epoch* 30), B.13 (*Epoch* 35), B.12 (*Epoch* 40).

B.3 vgg16

B.3.1. Gráficas *accuracy*

En este apartado encontramos las figuras B.17 (*Epoch* 30), B.16 (*Epoch* 35), B.15 (*Epoch* 40).

B.3.2. Gráficas tiempo de ejecución

En este apartado encontramos las figuras B.20 (*Epoch* 30), B.19 (*Epoch* 35), B.18 (*Epoch* 40).

B.4 resnet18

B.4.1. Gráficas *accuracy*

En este apartado encontramos las figuras B.23 (*Epoch* 30), B.22 (*Epoch* 35), B.21 (*Epoch* 40).

B.4.2. Gráficas tiempo de ejecución

En este apartado encontramos las figuras B.26 (*Epoch* 30), B.19 (*Epoch* 35), B.24 (*Epoch* 40), B.27 (comparativa tamaño *batch* 2 procesos), B.28 (comparativa tamaño *batch* 4 procesos), B.29 (comparativa tamaño *batch* 8 procesos).

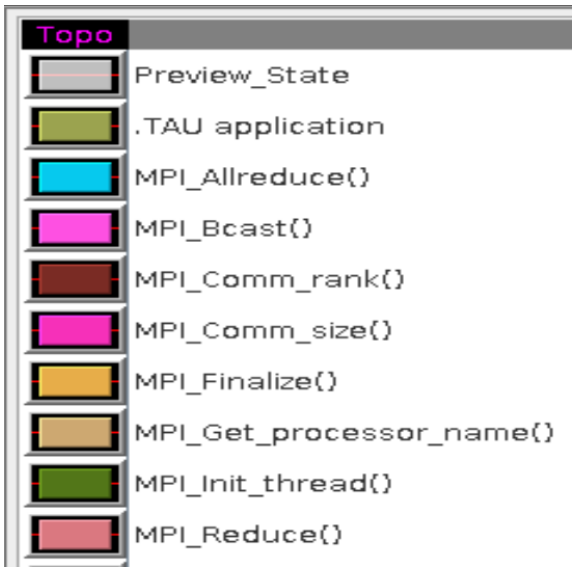


Figura B.1: Leyenda de la figura 5.2

Figura B.2: Leyenda de la figura 5.3

Figura B.3: Leyenda de la figura 5.5

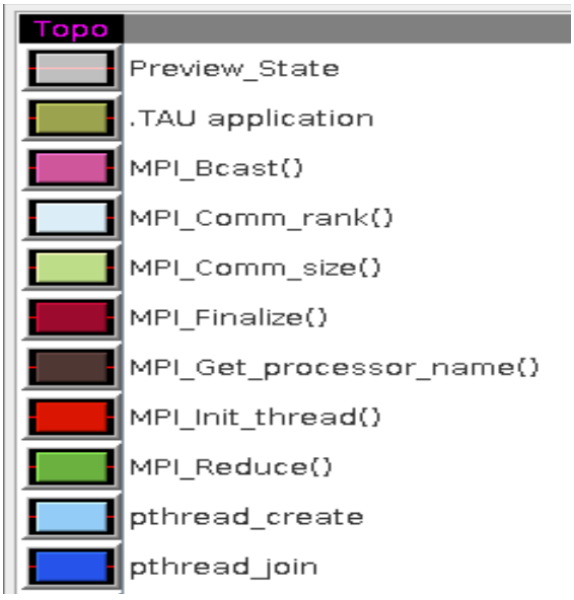


Figura B.4: Leyenda de la figura 5.6

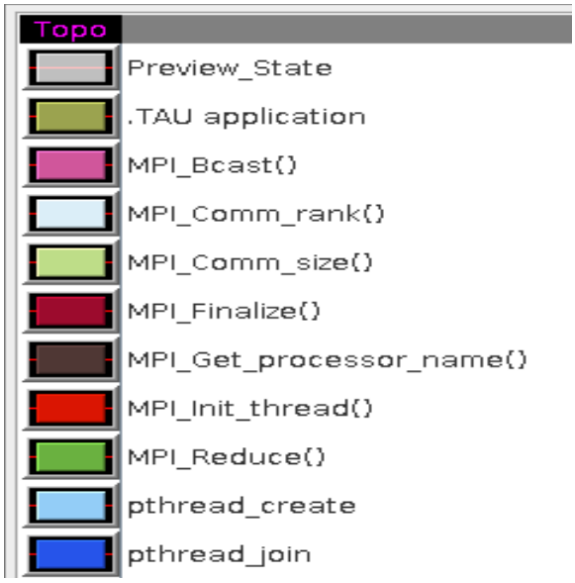


Figura B.5: Leyenda de la figura 5.7

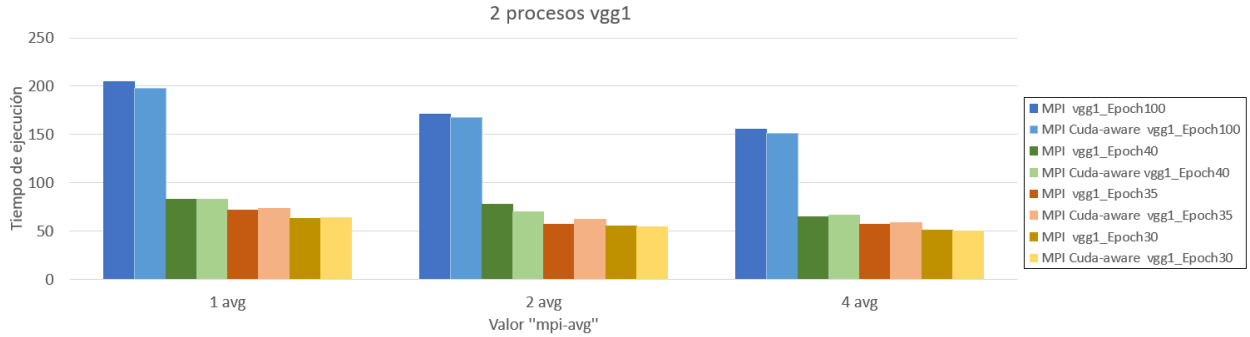


Figura B.6: Ejecuciones con la red vgg1 para el "mpi-avg" de 1,2 o 4 para un *epoch* de 100,40,35 y 30 con 2 procesos

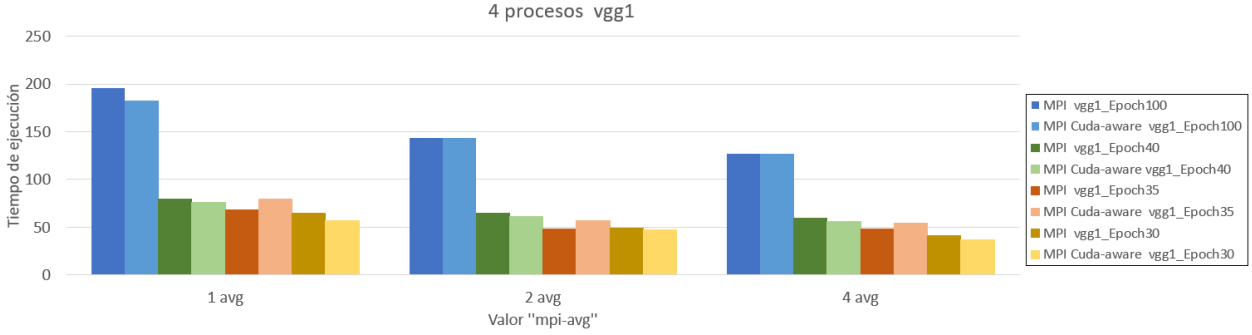


Figura B.7: Ejecuciones con la red vgg1 para el "mpi-avg" de 1,2 o 4 para un *epoch* de 100,40,35 y 30 con 4 procesos

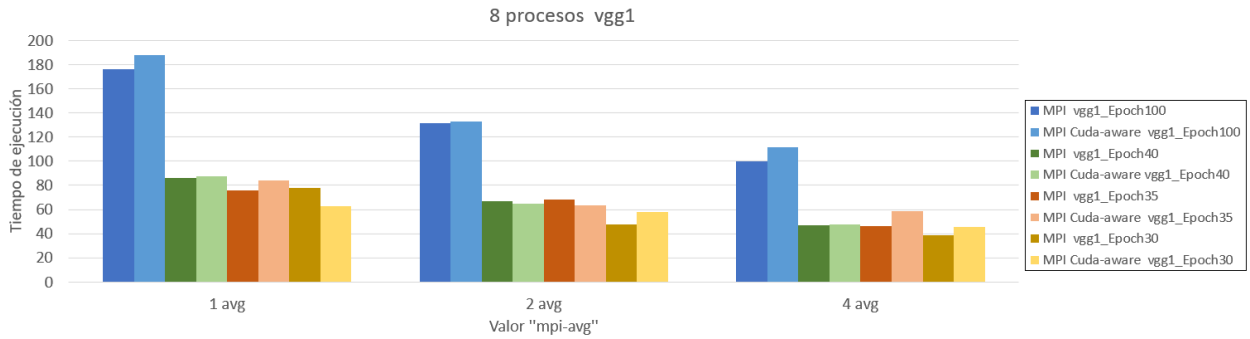


Figura B.8: Ejecuciones con la red vgg1 para el "mpi-avg" de 1,2 o 4 para un *epoch* de 100,40,35 y 30 con 8 procesos

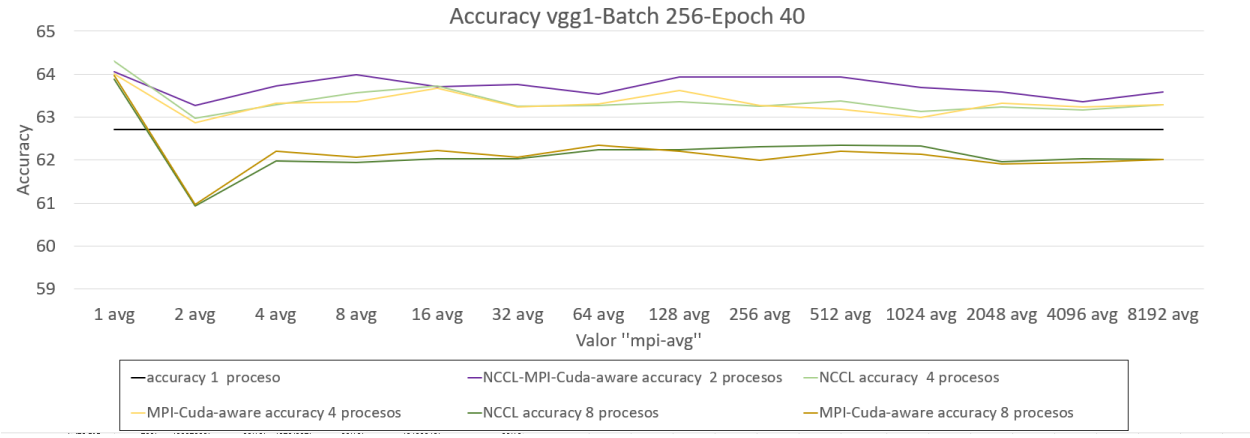


Figura B.9: *Accuracy* con la red vgg1 para un *epoch* de 40 y un *batch* de 256

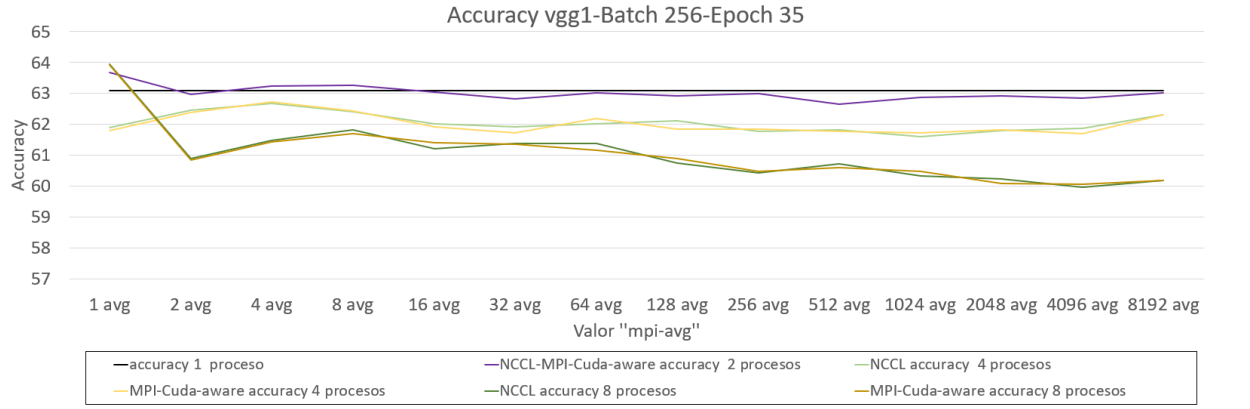


Figura B.10: *Accuracy* con la red vgg1 para un *epoch* de 35 y un *batch* de 256

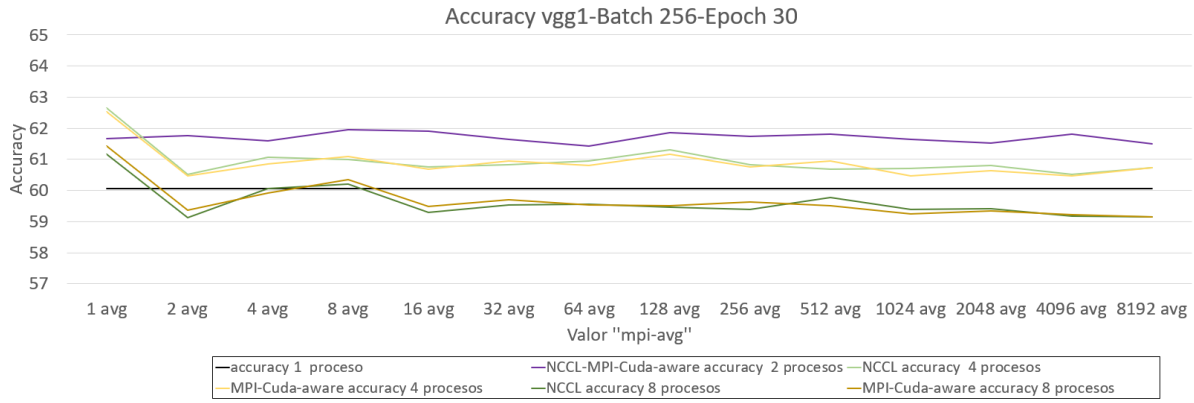


Figura B.11: *Accuracy* con la red vgg1 para un *epoch* de 30 y un *batch* de 256

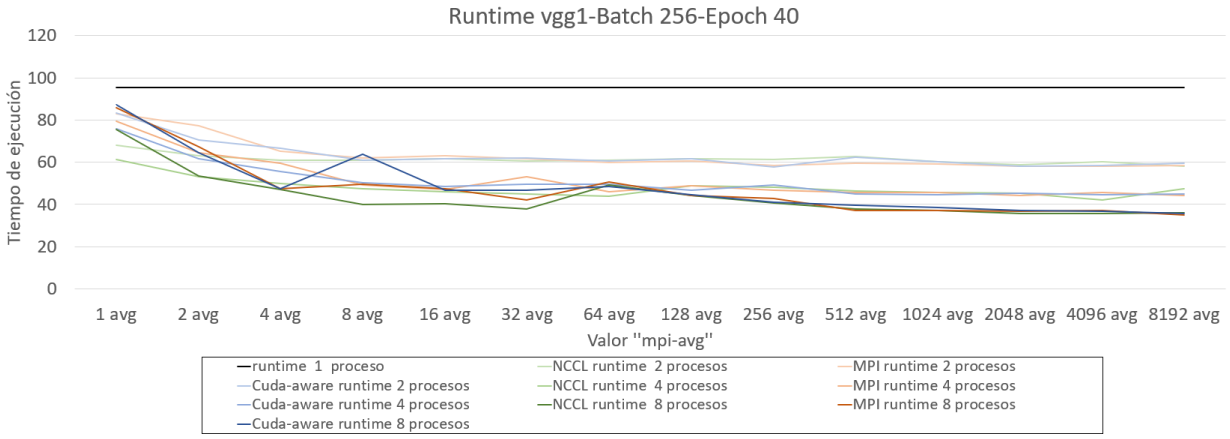


Figura B.12: Tiempo de ejecución con la red vgg1 para un *epoch* de 30 y un *batch* de 256

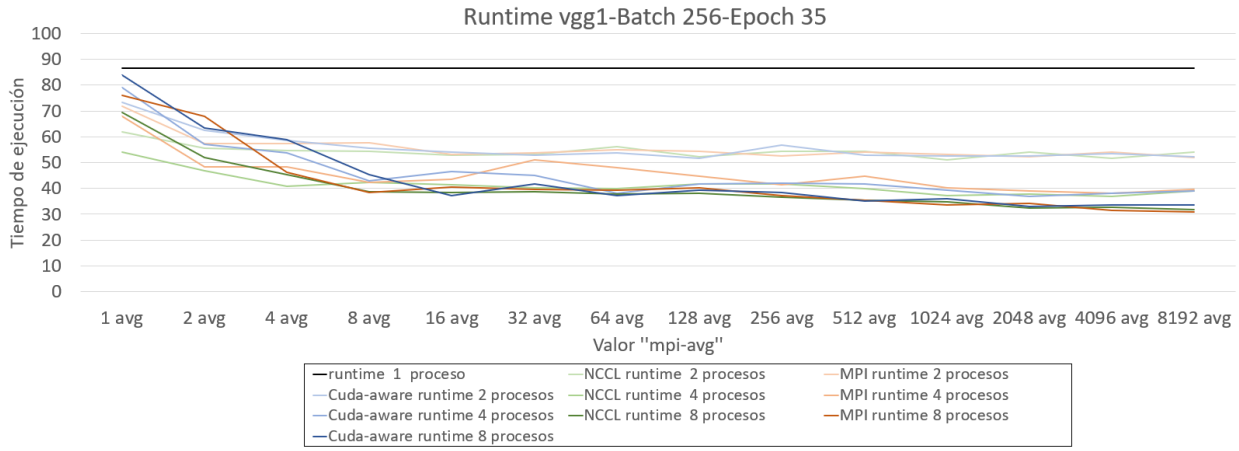


Figura B.13: Tiempo de ejecución con la red vgg1 para un *epoch* de 35 y un *batch* de 256

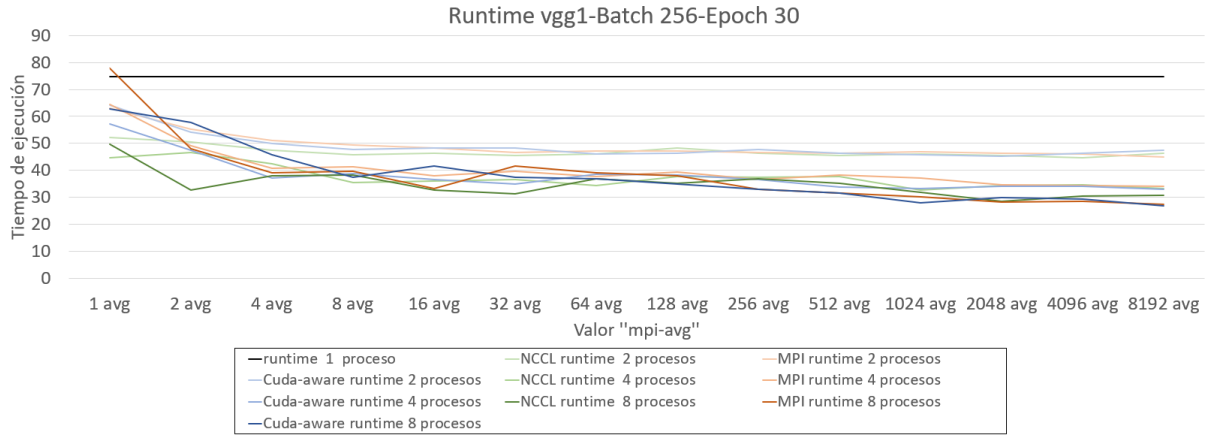


Figura B.14: Tiempo de ejecución con la red vgg1 para un *epoch* de 30 y un *batch* de 256

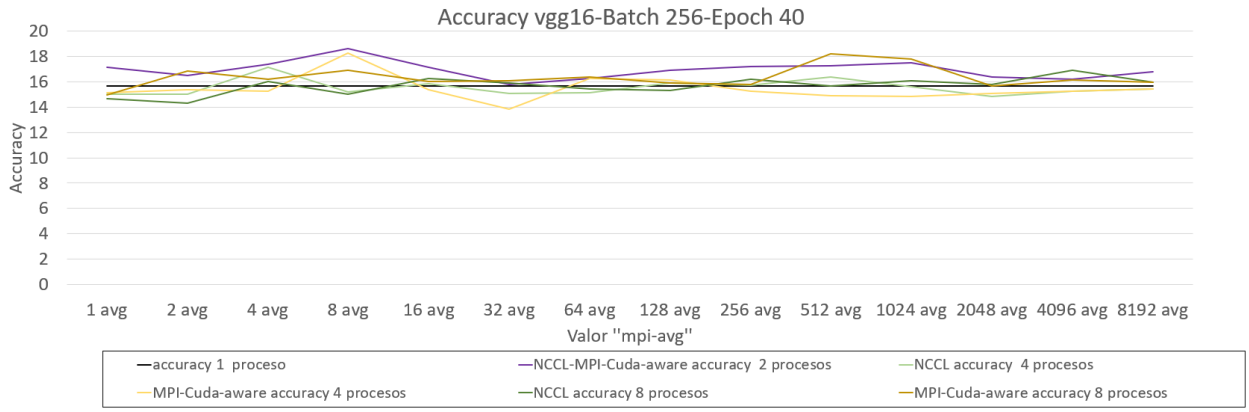


Figura B.15: *Accuracy* con la red vgg16 para un *epoch* de 40 y un *batch* de 256

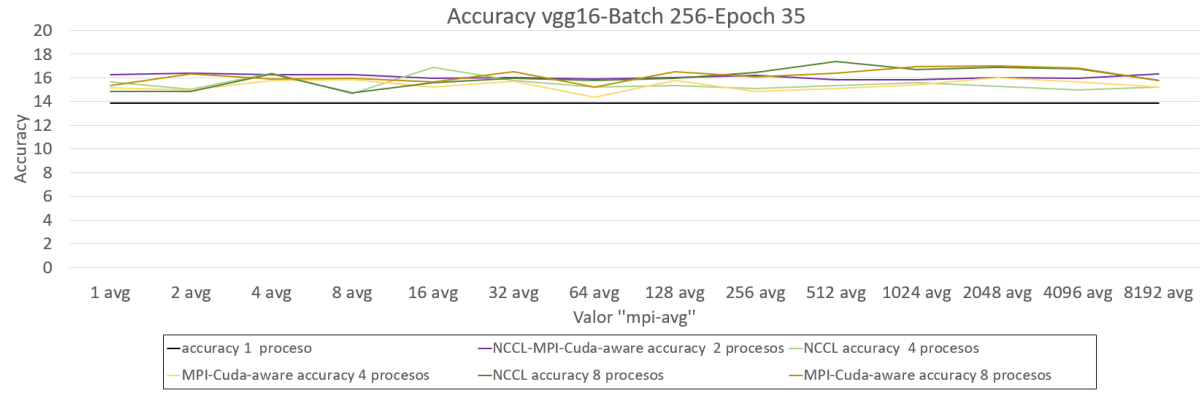


Figura B.16: *Accuracy* con la red vgg16 para un *epoch* de 35 y un *batch* de 256

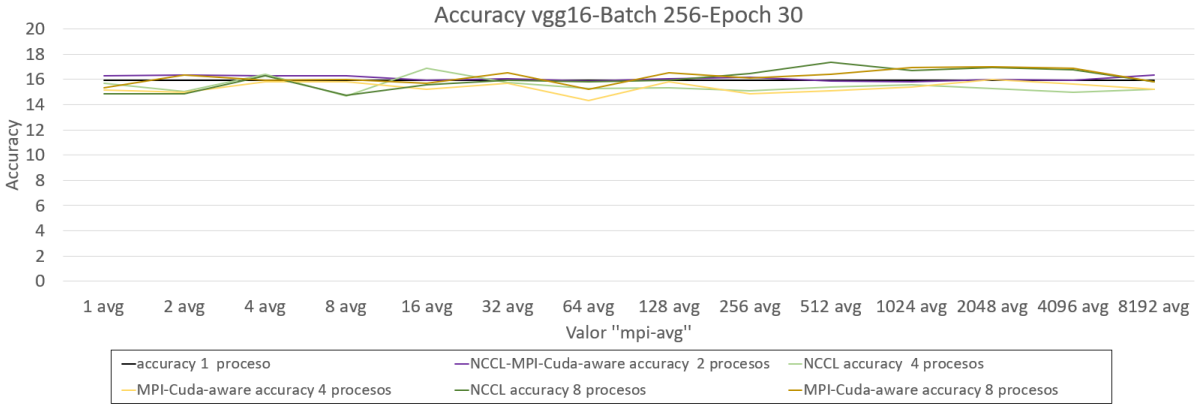


Figura B.17: *Accuracy* con la red vgg16 para un *epoch* de 30 y un *batch* de 256

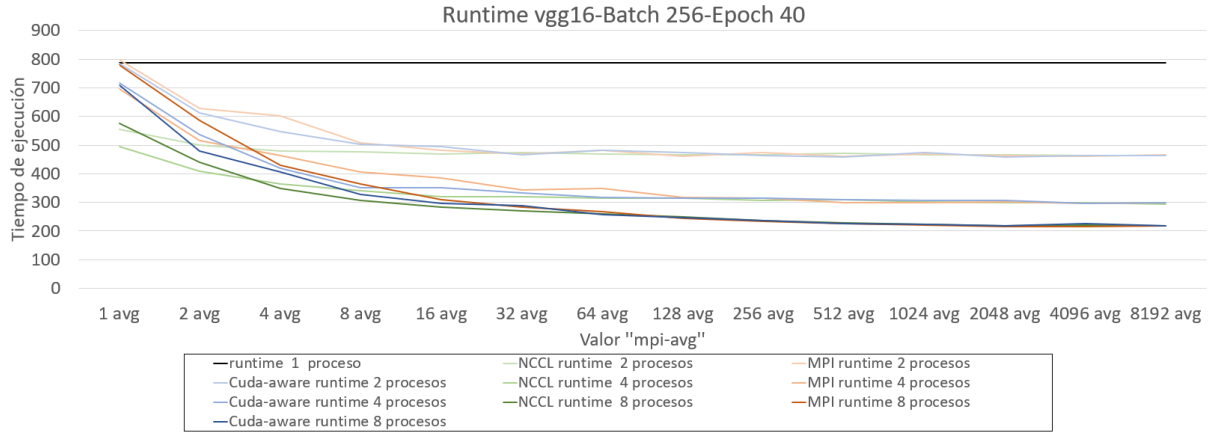


Figura B.18: Tiempo de ejecución con la red vgg16 para un *epoch* de 40 y un *batch* de 256

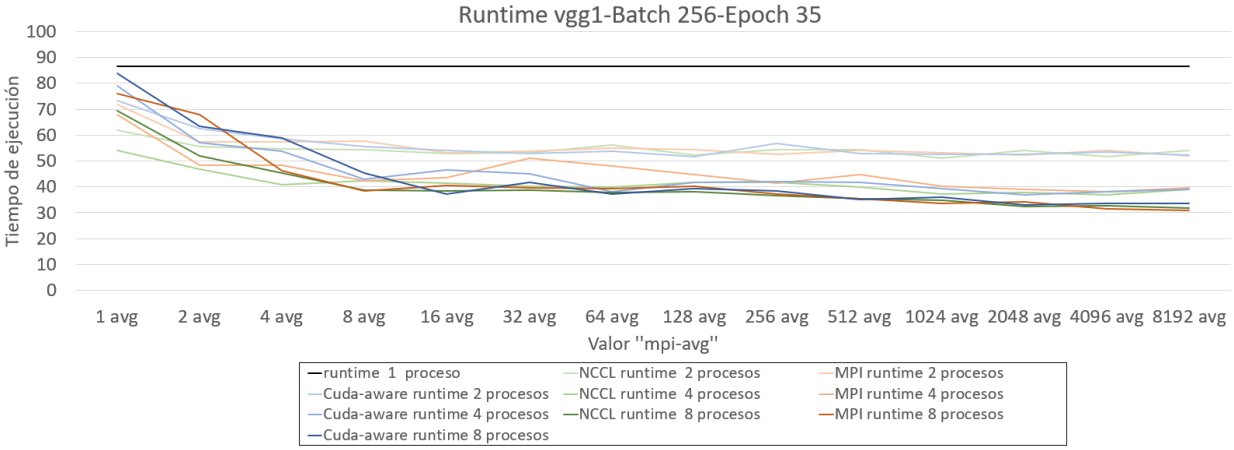


Figura B.19: Tiempo de ejecución con la red vgg16 para un *epoch* de 35 y un *batch* de 256

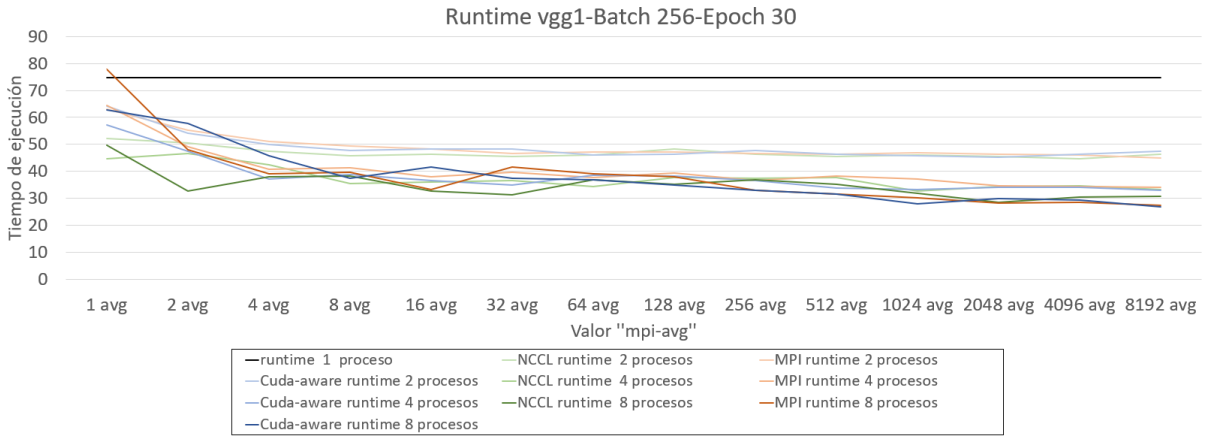


Figura B.20: Tiempo de ejecución con la red vgg16 para un *epoch* de 30 y un *batch* de 256

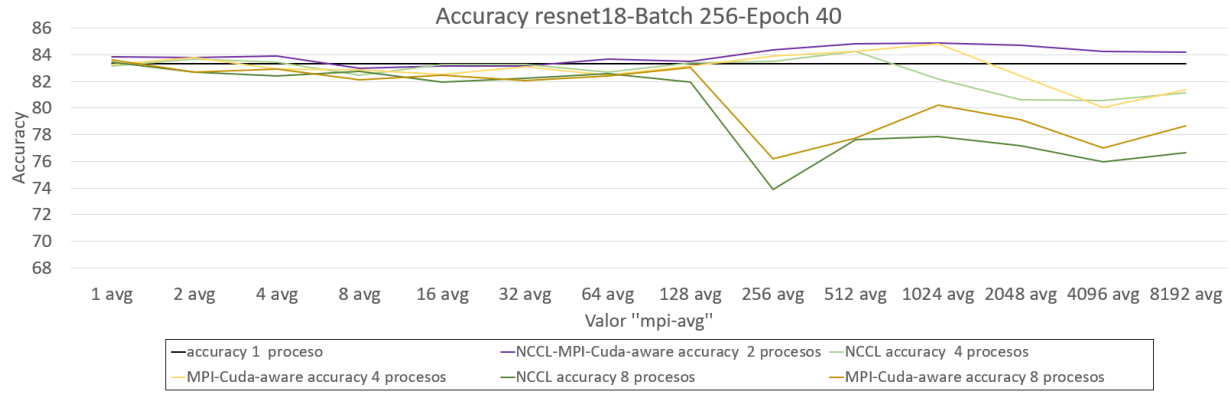


Figura B.21: *Accuracy* con la redr esnet18 para un *epoch* de 40 y un *batch* de 256

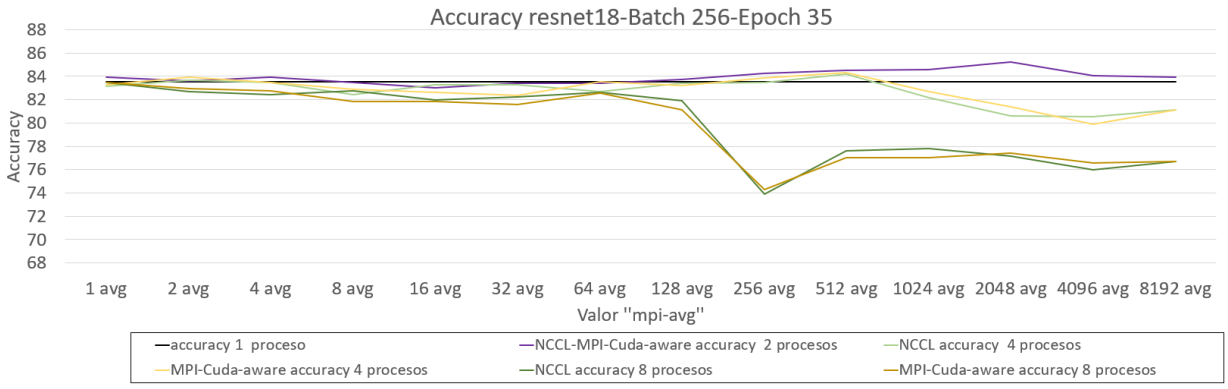


Figura B.22: *Accuracy* con la red resnet18 para un *epoch* de 35 y un *batch* de 256

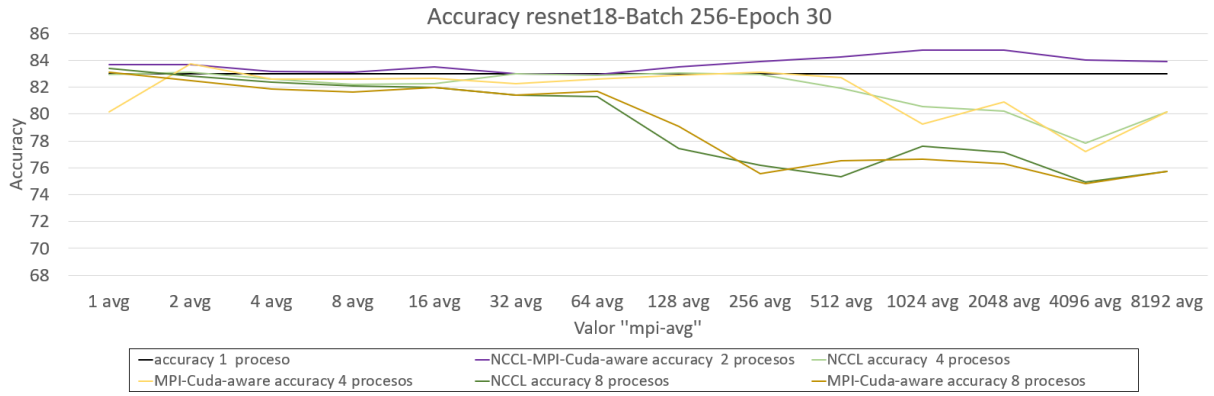


Figura B.23: *Accuracy* con la red resnet18 para un *epoch* de 30 y un *batch* de 256

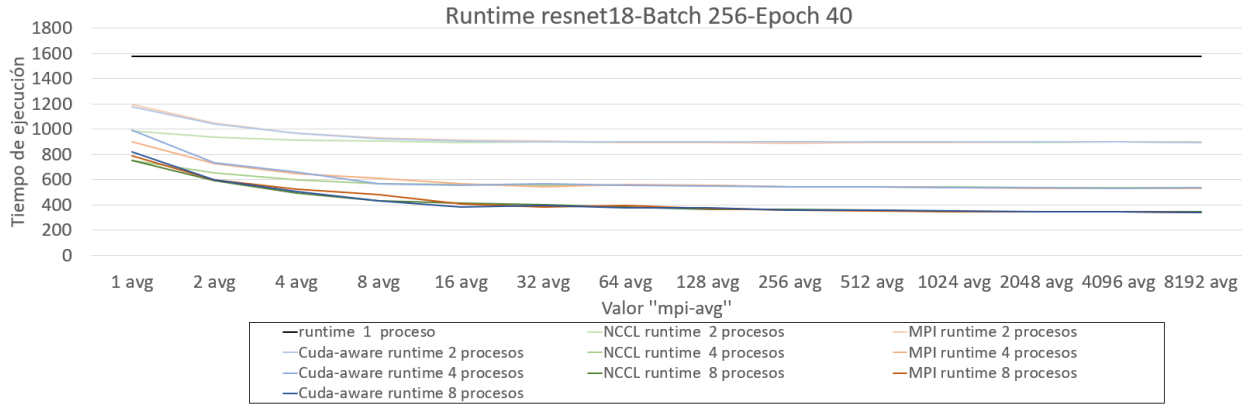


Figura B.24: Tiempo de ejecución con la red resnet18 para un *epoch* de 40 y un *batch* de 256

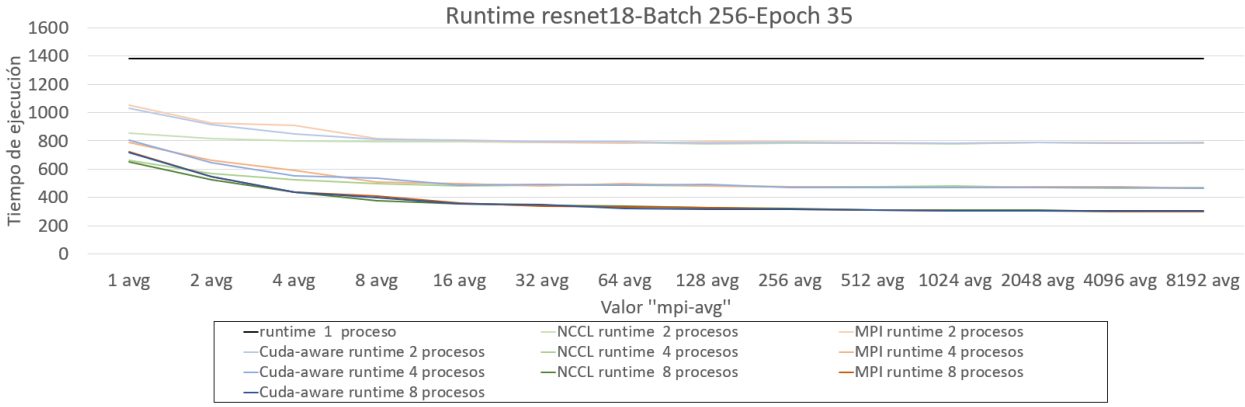


Figura B.25: Tiempo de ejecución con la red resnet18 para un *epoch* de 35 y un *batch* de 256

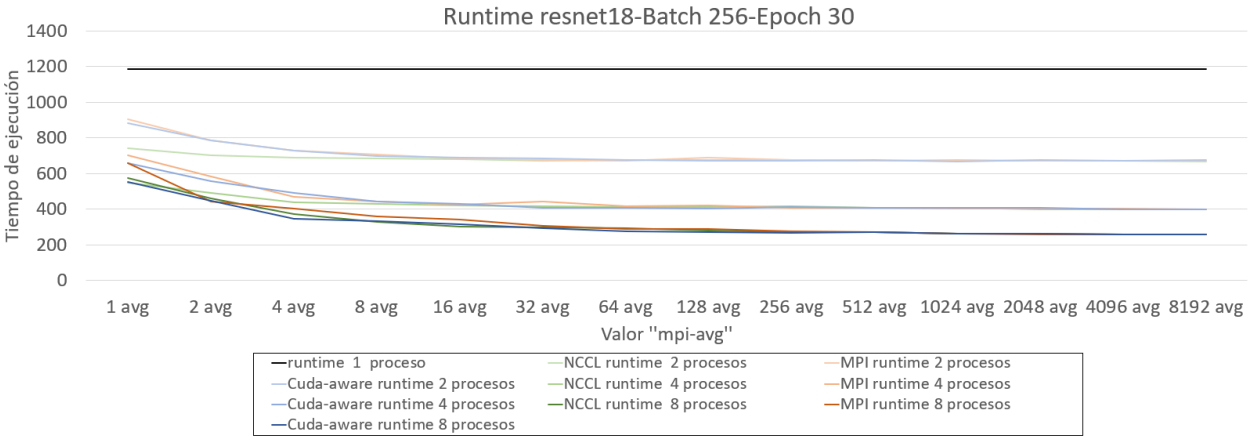


Figura B.26: Tiempo de ejecución con la red resnet18 para un *epoch* de 30 y un *batch* de 256

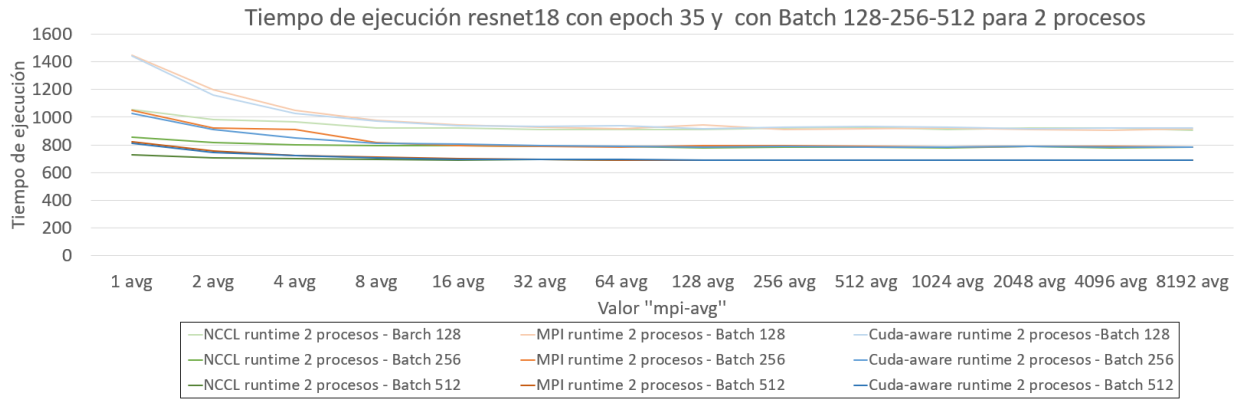


Figura B.27: Comparativa tiempo de ejecución con la red resnet18 para un *epoch* de 35 y un *batch* de 128-256-512 para 2 procesos

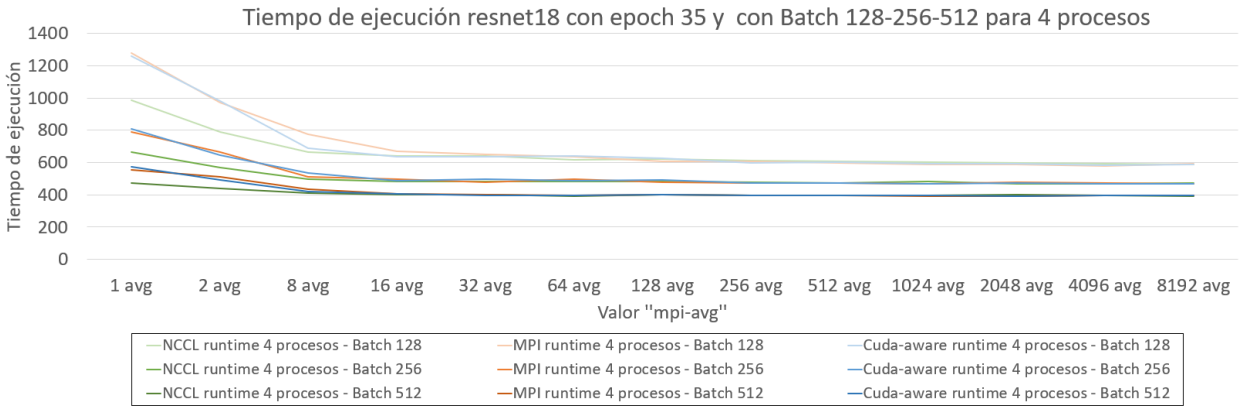


Figura B.28: Comparativa tiempo de ejecución con la red resnet18 para un *epoch* de 35 y un *batch* de 128-256-512 para 4 procesos

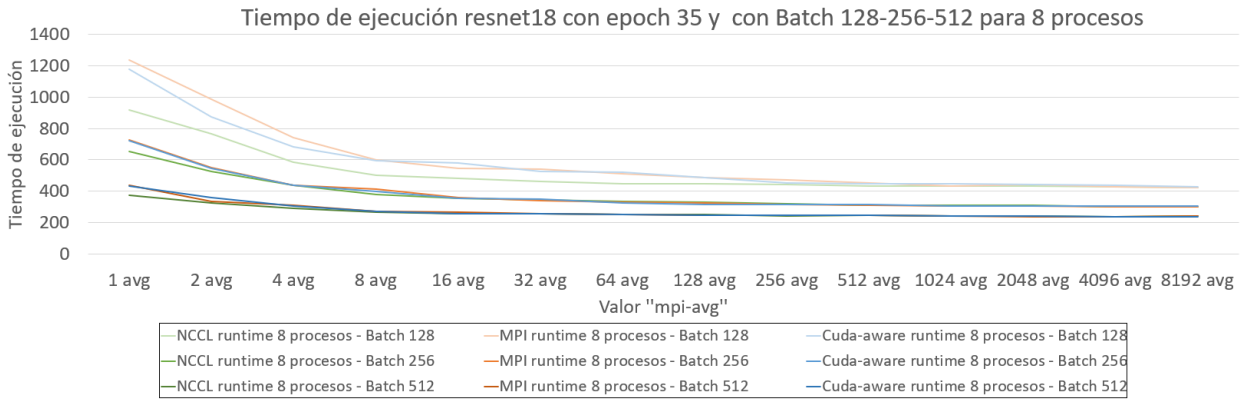


Figura B.29: Comparativa tiempo de ejecución con la red resnet18 para un *epoch* de 35 y un *batch* de 128-256-512 para 8 procesos