Facultad de Ciencias

Análisis de rendimiento de aplicaciones de Deep Learning sobre procesadores de propósito general

(Performance analysis of Deep Learning applications on general purpose processors)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Juan Luis Padilla Salomé

Director: Pablo Abad Fidalgo

Julio – 2021

RESUMEN

Las aplicaciones de *Deep Learning* se han extendido de manera significativa en los últimos años, obligando incluso a crear nuevas tecnologías específicas para este campo (TPU). Por ello, los fabricantes de procesadores de propósito general se ven en la necesidad de mejorar sus productos para que sean competentes. Y es ahí donde surge la necesidad de encontrar una herramienta que sea capaz de realizar un análisis del rendimiento del sistema. Haciendo más fácil entender las interacciones entre el hardware y software y facilitando las labores de optimización y desarrollo.

En este trabajo se ha diseñado una herramienta, de nombre *DL-Prof*, para evaluar el rendimiento de los procesadores de propósito general cuando se ejecutan aplicaciones de *Deep Learning*. Desde el punto de vista software se ha trabajado con los entornos de desarrollo más habituales en DL. Para ello, se ha hecho uso de los contadores existentes en este tipo de hardware de propósito general.

La herramienta se ha validado de manera exhaustiva y se han realizado experimentos sobre aplicaciones de Deep Learning donde se ha demostrado la utilidad de la herramienta.

Palabras clave: Deep Learning, Keras, PMU, DL-Prof.

ABSTRACT

Deep Learning applications have expanded significantly in recent years, even forcing the aparition of new technologies specific to this field (TPU). Therefore, manufacturers of general-purpose processors find it necessary to improve their products to make them proficient. And that is where the need arises to find a tool that is capable of performing an analysis of the performance of the system. Making it easier to understand the interactions between hardware and software and facilitating optimization and development tasks.

In this work, a tool, called DL-Prof, has been designed to evaluate the performance of general-purpose processors when running *Deep Learning* applications. From a software point of view, we have worked with the most common development environments in DL. To do this, the existing counters in this type of general-purpose hardware has been used.

The tool has been extensively validated and experiments have been carried out on *Deep Learning* applications, where the usefulness of the tool has been demonstrated.

Key words: Deep Learning, TensorFlow, PAPI, DL-Prof

Índice de contenidos

Índice de Figuras

Índice de Tablas

1 Introducción y objetivos

Desde la aparición del primer microprocesador hace medio siglo, el computador ha sido el motor de los grandes avances tecnológicos que se han sucedido hasta el día de hoy. En poco más de 50 años Intel ha pasado del *4004* en 1971 [1], con una frecuencia de 740 kHz y transistores de 10 µm, a microprocesadores que operan a frecuencias superiores a los 4 GHz [2] y transistores mil veces más pequeños (~14 nm). Estos avances han permitido que la integración que tienen los computadores en nuestro día a día sea total; convirtiendo al computador en una herramienta fundamental en cualquier aspecto de nuestra vida cotidiana (evidencia de ello es la cantidad de aparatos electrónicos de uso diario que incorporan un procesador: móviles, automóviles, cajero automático, etc.).

Los computadores son los responsables de que algunos avances conseguidos en la actualidad parezcan extraídos de novelas de ciencia ficción. Un claro ejemplo de ello es el área de la Inteligencia Artificial (IA), que ha experimentado un boom en los últimos 10 años gracias a las GPUs [3]. Este tipo de procesadores, cuya arquitectura permite aprovechar sus múltiples *cores* para ejecutar en paralelo miles de *threads*, son el hardware adecuado para aplicaciones con un elevado paralelismo de datos, como las relacionadas con el entorno IA [4]. Esto ha permitido un incremento sustancial en la potencia de cálculo disponible, haciendo posible el desarrollo de algoritmos y aplicaciones inimaginables hace tan solo unos años.

A pesar de que la relevancia actual de la Inteligencia Artificial nos puede hacer pensar que se trata de un campo que ha surgido recientemente, éste es un concepto antiguo y cuyos primeros pasos se remontan a los años 50. Década en la que el investigador británico Alan Turing ya se planteaba la posibilidad de que las máquinas pudieran pensar, así como mecanismos para comprobar su inteligencia [5]. Desde sus inicios, el campo del aprendizaje máquina (Machine Learning o ML) ha evolucionado de manera vertiginosa hasta la actualidad, donde la aplicación de técnicas de IA ha demostrado ser un éxito en campos tan dispares como la banca, la medicina o el entretenimiento. Un ejemplo de la situación actual se puede encontrar con el modelo de lenguaje de *OpenAI*: GPT-3 [6], que utiliza el procesamiento de lenguaje natural (NLP) para, entre otros ejemplos, desarrollar código en cualquier lenguaje de programación, con solo describir las características necesarias (el resto se hace de manera autónoma, sin intervención humana [7]. Otro ejemplo llamativo se puede encontrar en el mundo farmacéutico donde se utiliza la IA para predecir si un medicamento es seguro y efectivo para probar en personas; y encontrar nuevos usos para esos medicamentos [8].

La Inteligencia Artificial nos rodea, aunque no seamos conscientes de ello, y parece que esta dependencia se va a incrementar en un futuro cercano, motivo por el cual se ha elegido este campo de trabajo para desarrollar el presente TFG. Se trata de un campo muy amplio, englobando todo procedimiento para automatizar tareas realizadas por humanos (ver Figura 1-1). En su vertiente más clásica, la rama de IA conocida como *symbolic* tiene como objetivo automatizar la obtención de respuestas mediante un conjunto de datos y reglas conocidas.

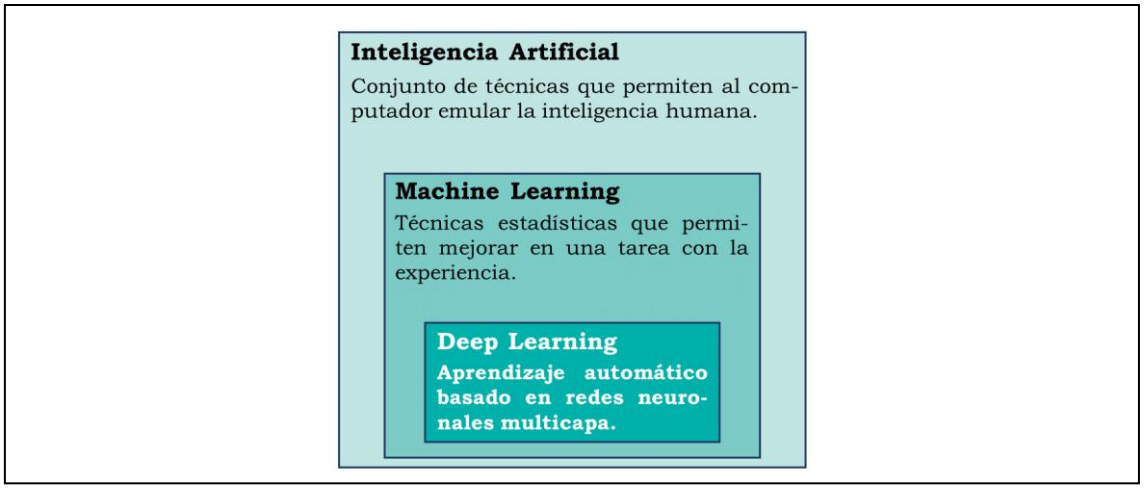

Figura 1-1: Descripción de los campos de la Inteligencia Artificial.

De forma complementaria, la rama conocida como Aprendizaje Automático (*Machine Learning* o ML) tiene un enfoque distinto, centrándose en el aprendizaje de las reglas a partir de conjuntos de entradas y salidas conocidas. El ML se ha convertido actualmente en la rama más popular dentro de la IA, y más especialmente el *Deep Learning* (DL). Este subcampo del ML se caracteriza por el tipo de estructuras utilizadas en el proceso de aprendizaje, consistente en capas de neuronas artificiales interconectadas entre sí y denominadas redes neuronales artificiales. Su uso está presente en aplicaciones tan diversas como la identificación y control de sistemas (detección de objetos, conducción de vehículos, predicción de trayectoria, etc.), filtrado de redes sociales y correo electrónico (spam), algoritmos de búsqueda y recomendación, predicciones meteorológicas y del mercado bursátil, entre muchas otras.

El procesador es el substrato tecnológico que ha posibilitado los numerosos avances en el área del *Machine Learning*. En ocasiones, la potencia de cálculo disponible ha sido el factor limitante que ha ralentizado los avances en el área [9]. De manera complementaria, en otras ocasiones ha sido el componente hardware el que ha posibilitado grandes saltos cualitativos. Como se ha mencionado previamente, éste es el caso de las GPUs, las cuales cobraron una mayor importancia en la última década, desde que el investigador *Alex Krizhevsky* vio que eran un hardware propicio para la ejecución de algoritmos basados en máquinas de Boltzmann [10] (un tipo de red neuronal estocástica), abriendo la posibilidad de utilizar redes aún más complejas (redes neuronales profundas) con un coste computacional realista [4]. Así, el ganador del concurso de *ImageNet* [11] en 2012 (competición donde los equipos evalúan sus algoritmos con un conjunto de datos determinado, y rivalizan por conseguir una mayor precisión en tareas de reconocimiento visual), se aprovechó de la velocidad de procesamiento que se obtiene con las GPUs para entrenar una red neuronal con más de un millón de imágenes. El tiempo que le costó fue de tan sólo 5-6 días, y no las semanas, o incluso meses, que se tardaba antes con otros medios. En menos de 10 años hemos asistido a una explosión del área del *Machine Learning* (y el *Deep Learning* más concretamente), con miles de publicaciones científicas cada año y un esfuerzo por parte de las grandes empresas (*Google*, *Amazon*, *Intel*, *NVIDIA*, etc.) por reposicionar parte de su negocio alrededor de esta tecnología. La relevancia actual de este

campo lo convierte en un punto de partida razonable para el desarrollo de un TFG. La siguiente sección profundiza sobre las motivaciones detrás de la elección de la temática del trabajo realizado.

1.1.Motivación

En general, los algoritmos de *Machine Learning*, y más particularmente del campo del *Deep Learning*, son capaces de obtener grandes mejoras de rendimiento derivadas del procesamiento paralelo. Es por esto por lo que las arquitecturas de tipo SIMD (*Single Instruction Multiple Data*) parecen la base más adecuada para su ejecución. Arquitecturas genéricas como las GPGPUs [3], o modelos específicos del área como las TPU [12] son el hardware sobre el que se entrenan y utilizan las redes neuronales profundas en muchos casos.

A pesar de parecer desplazada a un segundo plano, los procesadores de propósito general también desempeñan un papel relevante en esta área; y es que, en ocasiones, tanto GPUs como TPUs pueden suponer un modelo poco adecuado para este tipo de aplicaciones, haciendo necesario volver sobre arquitecturas de propósito general como las CPUs [13]:

- Las redes convolucionales 3D, e incluso las 2D con un tamaño de *batch* grande, requieren en ocasiones una gran cantidad de memoria. En GPUs, a veces esto supone un cuello de botella debido a la limitada capacidad de memoria con la que cuentan y la baja tasa de transferencia; pudiendo hacer recomendable el entrenamiento de dichas redes en CPUs.
- En redes neuronales de tipo recurrente (*Recurrent Neural Network*), donde la paralelización es difícil debido a la dependencia secuencial de los datos y, más en concreto, en aplicaciones cuyo acceso a memoria es irregular, la arquitectura donde se obtiene un mejor rendimiento es en las CPUs.
- El precio de desarrollo o adquisición de estos aceleradores es solamente asumible por las grandes compañías tecnológicas (*Google*, *Amazon*, *Facebook*, etc.). Por ejemplo, las TPUs de última generación están disponibles solamente a través de los servicios *cloud* de *Google*[1], limitando el acceso a este tipo de tecnología. Para poder utilizarla es necesario pagar las cuotas que pone esta compañía por horas o meses, llegando en este último caso a ser superiores a los mil dólares[2].

Por las razones descritas, los fabricantes de microprocesadores de propósito general están incorporando características a sus productos dedicadas a mejorar el rendimiento cuando se trabaja con aplicaciones de Inteligencia Artificial. Véase el caso de *Intel*, que ha lanzado esta tecnología con el nombre de *Deep Learning Boost* [14] y la cual dota a sus procesadores de la familia *Ice Lake* (en adelante) con soporte para instrucciones AVX-512 (instrucciones vectoriales de 512 bits capaces de explotar el paralelismo a nivel de datos existente en aplicaciones de DL). Es evidente que el rendimiento de las aplicaciones de *Deep Learning* es un factor clave en el diseño de las nuevas familias de procesadores de propósito general. Desafortunadamente, el desarrollo de este tipo de aplicaciones se

[1] https://cloud.google.com/compute
[2] https://cloud.google.com/tpu/pricing

lleva a cabo habitualmente mediante *frameworks* de alto nivel, con una estructura software compleja y de múltiples capas, haciendo complicado entender cómo va a interaccionar nuestro código con el hardware sobre el que se ejecuta. Este desconocimiento sobre la interacción hardware-software puede dificultar las labores de optimización y desarrollo, lo que hace atractivo el desarrollo de herramientas que faciliten dicha tarea.

1.2. Objetivos

El objetivo final de este TFG es el desarrollo de una herramienta que nos ayude a comprender cómo interaccionan las aplicaciones de *Deep Learning* con los elementos principales de la arquitectura del procesador. Para el desarrollo de la herramienta, de nombre *DL-Prof*, se han seguido los pasos descritos a través de las secciones de este documento, que se resumen a continuación:

- En el capítulo 2 se describen las herramientas que se han utilizado para el desarrollo del *framework*. Tanto las nociones teóricas necesarias para comprender el funcionamiento de una aplicación de *Deep Learning* como el software elegido.
- En el capítulo 3 se desarrollan los pasos que se han llevado a cabo, con las herramientas comentadas en el capítulo anterior, para elaborar el *framework*, así como las dificultades encontradas durante el proceso.
- En el siguiente capítulo se detallan las pruebas de validación para comprobar que la herramienta desarrollada funciona de manera correcta. Se diseñan pequeños experimentos que nos ayudan en dicha tarea.
- A continuación, se describe el grueso de los experimentos planteados. En el capítulo 5 se usa *DL-Prof* para su propósito inicial de medir y observar el comportamiento del sistema al ejecutar aplicaciones de aprendizaje profundo.
- Para terminar, se dedica el último capítulo a detallar las conclusiones que se han extraído de los experimentos y sumergirnos en el futuro que se abre con esta herramienta.

Otro objetivo de este TFG es que todo el código desarrollado, así como los resultados de las pruebas y experimentos, que aparecen en los distintos capítulos, sean accesibles públicamente a través de un repositorio sobre el que se trabajará.

2 Concepto y herramientas utilizadas

Con los principales objetivos de la herramienta en mente, es necesario comenzar con una breve descripción de los diferentes componentes del *framework* completo con el que se pretende trabajar, desde la interfaz en el que se programan los modelos de red neuronal (*Keras, Tensorflow*) hasta los componentes hardware disponibles para el *profiling* de la microarquitectura (PMU, PAPI).

2.1. Deep Learning

Como se ha comentado en la sección anterior, DL es un subcampo del *Machine Learning* que se caracteriza por el tipo de estructuras utilizadas en el proceso de aprendizaje. Las redes neuronales artificiales (RNA) son la esencia del DL, donde su potencia, versatilidad y escalabilidad las hacen ideales para resolver tareas grandes y complejas de *Machine Learning*.

Inspiradas por el funcionamiento de las redes neuronales biológicas, presentes en nuestro cerebro, una RNA está compuesta por un conjunto de neuronas artificiales, también conocidas como *linear threshold unit* (LTU), interconectadas entre sí formando capas (*layers*). Las neuronas artificiales, en cada capa, aprenden cada vez más representaciones abstractas de los datos. Cada una de ellas recibe uno o varios parámetros de entrada en forma de números reales ($\mathbb{R}$) que se multiplican por unos pesos (valores asignados a cada entrada de parámetro). Se calcula la suma de las multiplicaciones y se aplica una función de activación (*activation function*) para obtener un parámetro de salida del LTU.

Para hablar de una RNA con un modelo secuencial, ésta debe tener una capa de entrada, otra de salida, y al menos una capa oculta. La capa de entrada (*input layer*) es la encargada de recibir los datos de entrada y comenzar a difundirlas a las capas posteriores contiguas. Las cuales tienen el nombre de capas ocultas (*hidden layer*), o capas intermedias, y son las responsables de procesar los datos hasta llegar a las de salida, las encargadas de tomar una predicción; en la Figura 2-1 se muestra un esquema de cómo están representadas.

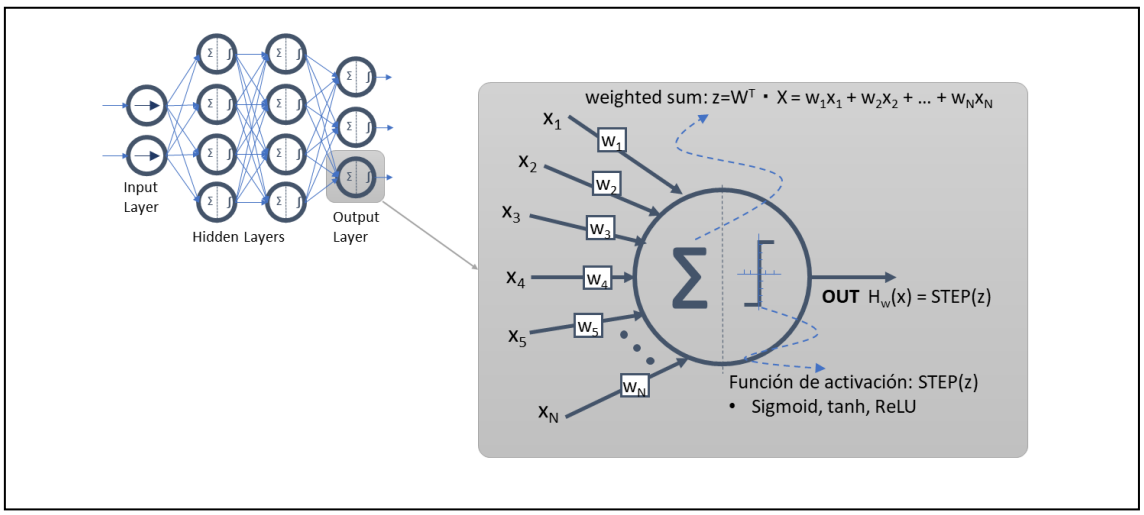

Figura 2-1: Estructura de una RNA y un LTU.

Las funciones de activación están presentes en cada capa y son las que definen cómo la suma de los pesos es transformada a un parámetro de salida de un nodo (LTU) a otro. Todas las capas ocultas, normalmente, tienen la misma función de activación, y es la capa de salida la que sí tiene otra función que depende del tipo de predicción a realizar. Los pesos en cada capa empiezan con valores aleatorios que son mejorados de manera reiterada con la intención de mejorar la precisión de la red. Una función de pérdida (*loss function*) es usada para medir cómo de imprecisa es la red; y, se usa el algoritmo de propagación para atrás (*backpropagation*) para determinar si cada peso debe aumentarse o disminuirse para reducir la pérdida.

2.1.1. Entornos de desarrollo

En la actualidad existen varios entornos de desarrollo para crear modelos de *Deep Learning*, los más importantes son:

- *Pytorch*. Es una librería desarrollada, en un principio, por Facebook y que actualmente es software libre y gratuito. Está destinada principalmente para su uso en *Python*, pero también tiene una interfaz en *C++*. Provee calculo tensorial (como *NumPy*) con optimizaciones para GPUs.
- *Caffe*. Framework desarrollado originalmente por la universidad de California, escrita en C++ con una interfaz en Python. Destinado para aplicaciones de Deep Learning tales como clasificación y segmentación de imágenes. Soporta optimizaciones para la GPU y CPU (*NVIDIA cuDNN* e *Intel MKL*).
- *Tensorflow*. Desarrollado y mantenido por Google, es una librería que permite su fácil despliegue a través de distintas plataformas (CPUs, GPUs, TPUs), y desde ordenadores de escritorio hasta clústeres de servidores y dispositivos móviles. Está enfocado principalmente en el entrenamiento e inferencia de redes neuronales profundas.

De ellas, la más extendida es *Tensorflow* y la cual se ha elegido para realizar las pruebas en este TFG. Las ventajas que ofrece sobre las otras es que se trata de una plataforma de extremo a extremo (*end-to-end*); es decir, se encarga de todo el proceso desde el principio al final y no necesita de ninguna otra herramienta de terceros. Por su sencillez, es la más adecuada para experimentar/empezar con el mundo del DL. Utiliza una API de alto nivel llamada *Keras*, la cual se describe a continuación.

2.1.2. Keras

Escrita en *Python*, es una API de la librería de *Tensorflow* que permite crear y manejar modelos con topología no lineal, capas compartidas, e incluso múltiples entradas y salidas. Algunas funciones, básicas, con las cuales se puede operar con los modelos de DL se comentan a continuación:

- *compile()*. Permite configurar el modelo para el entrenamiento y elegir parámetros relevantes tales como las funciones de optimización y de pérdida, así como los parámetros a evaluar durante la fase de entrenamiento y la fase de pruebas.
- *fit()*. Entrena la red neuronal para un número fijo de iteraciones en un conjunto de datos, también llamado épocas (*epochs*).

- *evaluate()*. Devuelve el valor de pérdida y los valores de las métricas para el modelo en modo de prueba
- *predict()*. Genera predicciones para las muestras que se le pasan.

Como se puede ver en la Figura 2-2, las capas software que subyacen a un modelo de DL son muchas, lo que dificulta entender cómo interaccionan con el hardware, haciendo el *profiling* necesario para las labores de optimización y desarrollo.

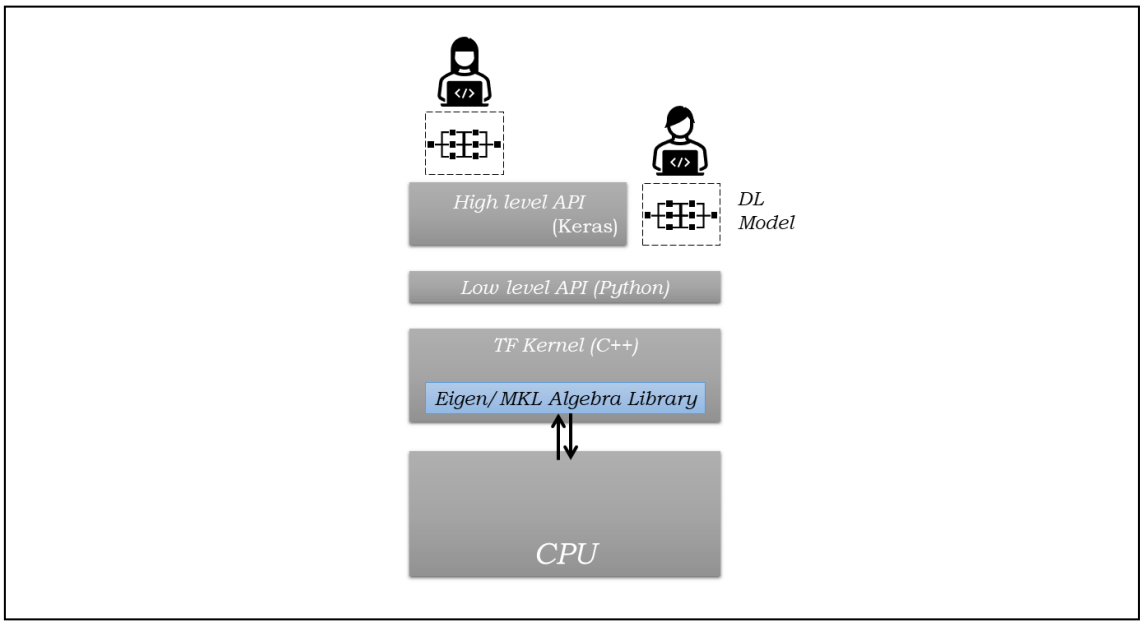

Figura 2-2: Capas software para el desarrollo de modelos de DL.

2.2.*Profiling* de aplicaciones

El aumento en las cotas de rendimiento ha sido una constante en la evolución del procesador. Parámetros como latencia y *throughput* son críticos en las aplicaciones, y diferencias de rendimiento mínimas pueden traducirse en grandes pérdidas o ganancias económicas. Véase el caso del *High-Frequency Trading* (HFT) [15], método que se utiliza en el mercado financiero utilizando potentes computadores y complejos algoritmos para realizar transacciones en milésimas de segundos. Para poder sacar el mayor partido del procesador, los desarrolladores de software deben ser capaces de entender cómo interacciona su código con el *hardware* sobre el que se ejecuta. Con este propósito existen multitud de herramientas[3] [16][17][18], basadas en diferentes mecanismos como la cuenta de eventos, *sampling* a intervalos regulares (uso estadístico), instrumentalización de código, etc.

En los últimos años, las herramientas de *profiling* utilizan la instrumentalización mediante la monitorización de eventos hardware [19][20](*VTUNE*, *Nsight*), pues permiten comprender los efectos que tienen sobre el rendimiento aspectos importantes del sistema, como la jerarquía de memoria, los protocolos de coherencia y la predicción de saltos,

[3] En *C* es común utilizar la herramienta *gprof* para ver el tiempo que se gasta en cada función al ejecutar un programa. *Valgrind*, en cambio, permite ver los problemas que tiene un código en relación con cómo se trata la memoria (accesos inválidos, no liberación de punteros, etc.). *VTune Profiler* permite analizar el rendimiento de máquinas basadas en x86 con *Windows* o *Linux* como SO.

entre otros. Nuestra herramienta hará uso de éstos, por lo que explicamos en las siguientes subsecciones algunos aspectos básicos sobre su implementación a bajo nivel (registros) y sobre la interfaz escogida para su utilización (librería *C/C++*)

2.2.1. PMU y Contadores Hardware

Actualmente, la mayoría de los procesadores cuentan con hardware específico para la monitorización de ciertos eventos micro-arquitecturales, tales como el número de ciclos transcurridos, aciertos y fallos en los distintos niveles de la cache, etc. Este proceso de monitorización se lleva a cabo a través de un conjunto de registros especiales, que tienen por nombre *Performance Monitoring Unit* (PMU). En el caso de los procesadores de *Intel*, que serán los utilizados en este TFG, dichos registros se agrupan en tres categorías diferentes de acuerdo con su función. Las siguientes secciones describen cada una de las categorías.

2.2.1.1. Registros de control global y estado.

Este primer conjunto de registros está a cargo del control de los registros contadores fijos y programables, aportando además información del estado de la PMU en general. Estos registros globales son:

- *IA32_PERF_CAPABILITIES*. Registro de sólo lectura que proporciona información acerca de las distintas opciones que puede soportar la máquina.
- *IA32_DEBUGCTL*. Este registro proporciona distintas opciones para depurar las distintas PMU.
- *IA32_PERF_GLOBAL_CTRL*. Permite controlar globalmente los contadores fijos y programables.
- *IA32_PERF_GLOBAL_STATUS*. Si ha ocurrido un desbordamiento (*overflow*) en cualquiera de los contadores (fijos y programables), indica cual ha sido.
- *IA32_PERF_GLOBAL_OVF_CTRL*. Registro de solo escritura que permite limpiar el bit si ha ocurrido un *overflow* en el registro anterior (*IA32_PERF_GLOBAL_STATUS*).
- *IA32_FIXED_CTR_CTRL*. Controla los contadores fijos y determina si tienen permitido el generar interrupciones.

2.2.1.2. Registros de control para los contadores programables.

Son los encargados del proceso de configuración de los contadores programables. Mediante estos registros se lleva a cabo la elección de los eventos que se desean medir. Para ello, se tiene que activar cada contador por este registro local y globalmente. En la Figura 2-3 se muestra un esquema de los campos que componen un registro de control de 64 bits. La funcionalidad de cada campo se describe en la Tabla 2-1.

Tabla 2-1: Campos asociados al registro de control.

Etiqueta	Bits	Descripción
EVTSEL	0-7	Selección del evento micro-arquitectural a detectar
EVTMSK	8-15	Condiciones que cumplir para medir el evento especificado en EVTSEL
USR	16	El evento solamente se cuenta si se está ejecutando en modo usuario (niveles 1,2,3)
OS	17	El evento solamente se Cuenta si se está ejecutando en modo *kernel* (nivel 0)
E	18	Encargado de incrementar el contador cuando ocurre una coincidencia en alguna de las condiciones de este registro.
INT	20	Permite generar una interrupción cuando se detecta un *overflow* en algún contador
AnyThr	21	Indica si se desea medir un único core lógico o todo el core físico
EN	22	Activación / desactivación del contador localmente
INV	23	Indica el tipo de comparación (mayor o igual vs. menor) que se tiene que realizar con respecto al campo CMASK
CMASK	24-31	Cuando este campo es distinto de cero, se utiliza para compararlo con los contadores de eventos en cada ciclo.

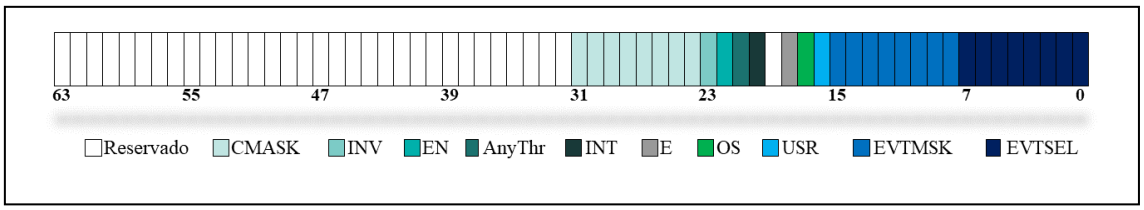

Figura 2-3: Descripción del MSR de control PerfEvtSeIX.

2.2.1.3. Registros de cuenta de eventos.

Finalmente, esta categoría hace referencia a los registros contadores fijos programables; todos ellos tienen un tamaño de 48 bits y su número depende de la familia del procesador, siendo habitual en los procesadores actuales encontrar tres contadores fijos (por core físico) y cuatro programables (por core lógico). Para modificar los cuatro registros programables se tiene que hacer con la instrucción/comando *wrmsr*.

Los registros de la PMU son de tipo MSR (*Model-Specific Registers*), un conjunto de registros de control utilizados para tareas de configuración, monitorización y *profiling*. El repertorio de instrucciones, que *Intel* proporciona para interaccionar con este tipo de registros, se compone por dos: RDMSR y WRMSR; pero trabajar con estas instrucciones de bajo nivel directamente es muy engorroso ya que sería necesario conocer los campos de cada registro para cada modelo y gestionar problemas adicionales (de permisos, de *overflow*, etc.) que pueden surgir. Esto es algo que no tiene mucho sentido pues existen herramientas que hacen el trabajo con estos registros mucho más sencillos, véase el caso de *perf tools* **¡Error! No se encuentra el origen de la referencia.** si se trabaja a través de la línea de comandos (o simplemente *perf*) o PAPI [21] para instrumentalizar código *C/C++*.

2.2.2. Performance Application Programming Interface (PAPI)

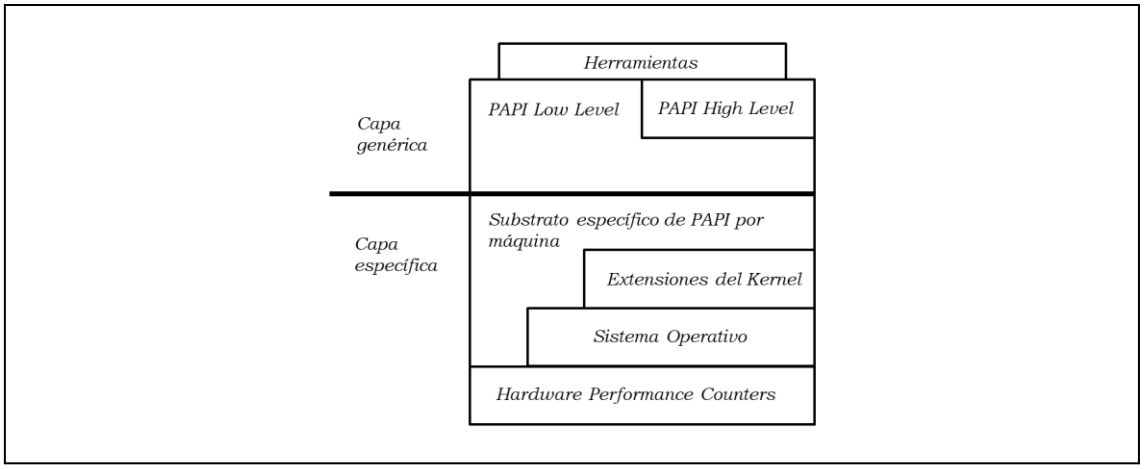

Figura 2-4: Diseño interno de la arquitectura de PAPI.

PAPI [21]es una herramienta, en forma de interfaz, que permite acceder a los contadores hardware presentes en los procesadores actuales. Desarrollado y mantenido por el *Innovative Computing Laboratory* (ICL) de la universidad de Tennessee, está escrita en el lenguaje *C* y viene en forma de librería (tanto estática como dinámica). Esta herramienta permite medir los contadores hardware no sólo de la CPU, sino también de la GPU, sistemas de entrada/salida, interconexiones, energía/potencia, etc. y está disponible para prácticamente todas las arquitecturas: *AMD*, *ARM*, *CRAY*, *IBM*, *Intel* y *Nvidia*, entre otras. En la Figura 2-4 se muestra la estructura en capa en la que se basa el diseño de PAPI. La capa superior se corresponde con las interfaces de alto y bajo nivel, así como las funciones independientes de la arquitectura de la máquina. La capa restante, de nombre Capa específica, está relacionada con las funciones y estructura de datos específicos de cada máquina sobre la que se usa. PAPI utiliza extensiones del *kernel*, llamadas al SO o el lenguaje ensamblador, dependiendo de cuál sea el más eficiente en cada arquitectura, para acceder a los contadores hardware.

Los eventos que se pueden medir en PAPI se clasifican en dos grupos:

- Eventos nativos (*native events*). Son todos aquellos eventos que son contables por la CPU. Este tipo de eventos son dependientes de la plataforma, de tal manera que es posible que algún tipo de evento solamente funcione en un subconjunto de las arquitecturas soportadas. Por ello, siempre que se pretenda utilizar PAPI es convenientes conocer la lista completa de eventos nativos disponibles. Esta información está disponible a través del comando *papi_native_avail*, proporcionado junto con las librerías de PAPI.
- Eventos preestablecidos (*preset events*). Son un conjunto de eventos que se encuentran en la mayoría de CPUs y que se consideran relevantes y útiles para medir el rendimiento de la aplicación. Estos pueden ser derivados; es decir, que pueden derivarse de otras métricas. Por ejemplo, el evento preestablecido *Total L1 Cache Misses* (PAPI_L1_TCM) se calcula como la suma de los eventos nativos *L1 Data Misses* y *L1 Instruction Misses*. Para acceder a la lista completa de eventos es necesario ejecutar el comando *papi_avail* en la terminal.

Como se ha mencionado previamente, PAPI proporciona dos niveles de acceso a la interfaz, de bajo nivel y alto nivel.

- *Low-level interface*. Permite manejar los eventos (a medir) en grupos llamados *"Event Sets"*, que deben ser definidos por el usuario. Su uso está recomendado si se desea medir con detalle. Proporciona más de 80 funciones que permite customizar la medida como desee el programador, además de soportar el uso de los dos grupos de eventos comentados anteriormente (*native events* y *preset events*).
- *High-level interface*. Está pensada para ser usada en aplicaciones de un solo *thread* y, a diferencia de la interfaz a bajo nivel, está limitada para la cuenta de eventos del tipo *preset events*. La ventaja que aporta es la de ser fácil de usar, pues la configuración antes de medir es implícita. Así, se basa en apenas cuatro funciones: *PAPI_hl_region_begin(const char* region)* y *PAPI_hl_region_end(const char* region)* permiten definir el inicio y final, respectivamente, de una región donde se han de medir los eventos. *PAPI_hl_read(const char* region)* empieza a medir la región elegida y *PAPI_hl_stop()* que detiene la ejecución de un *Event Set* determinado (opcional y sólo necesario si se combina con la interfaz de bajo nivel).

En resumen, PAPI permite, en prácticamente tiempo real, ver la relación que existe entre el rendimiento del software y los eventos hardware en todo el sistema. Es por ello por lo que *DL-Prof* hará un uso intensivo de la funcionalidad proporcionada por PAPI, tanto de su interfaz de alto nivel como de las funciones de bajo nivel por las ventajas que ofrece (y porque las aplicaciones de DL suelen ser *multithread*).

3 Herramienta *DL-Prof*

La optimización de rendimiento de una aplicación es una tarea habitual para la que existen múltiples herramientas [16][17][18][19][20][21]. Este tipo de *profiling* es más habitual cuando se usan lenguajes de bajo nivel (*C/C++*) pues se vuelve complicado a medida que las interfaces de software aumentan. Ante la carencia de herramientas que permita medir el comportamiento del sistema (a bajo nivel) cuando se ejecutan aplicaciones de *Deep Learning*, surge la necesidad de desarrollar un framework que realice dichas funciones.

Para la recopilación de información sobre la ejecución entran en juego los contadores hardware, registros de medida de eventos micro-arquitecturales descritos en la sección 2.2.1. Esto eventos permiten conocer al detalle lo que sucede en los componentes clave para el rendimiento del sistema. Por ejemplo, al ejecutar un programa, permite conocer el número de instrucciones que se han ejecutado, qué tipo de instrucciones son y cuántos ciclos de CPU ha tardado en ejecutarse el programa, qué uso de la jerarquía de memoria se ha hecho, cómo han funcionado mecanismos como la predicción de saltos o el *prefetching* entre otros muchos datos. El abanico de medidas que abren estos contadores es extenso, y algo de lo que se va a sacar mucho partido. El acceso a los contadores del procesador se hace a través de instrucciones específicas del repertorio de instrucciones, pero afortunadamente existen librerías que evitan trabajar a tan bajo nivel, como PAPI, interfaz de gestión de contadores descrita en la sección 2.2.2 y desarrollada como una librería en *C*.

En cuanto a la elección del lenguaje apropiado para el desarrollo, el objetivo será facilitar la integración de nuestra herramienta con los entornos habituales en el área. A pesar de que se pueden desarrollar modelos de red neuronal en cualquier lenguaje de programación (se pueden encontrar herramientas destinadas a *C/C++*, *Java*, *Fortran*, etc.), *Python* es actualmente el lenguaje principal en el cual se desarrollan la mayor cantidad de aplicaciones. Éste es el caso de *Keras*, la API desarrollada sobre la biblioteca de código abierto *TensorFlow* descrita en la sección 2.1.2 y que está diseñada para ser una interfaz amigable en la creación de redes neuronales artificiales. Dado que el objetivo es que la interfaz diseñada se utilice con modelos desarrollados con *Keras*, el lenguaje de programación elegido para desarrollar la herramienta es *Python*.

Para conjugar librerías de medida de eventos en *C* y modelos desarrollados en una API de *Python* y poder hacer un *profiling* del sistema cuando se ejecutan aplicaciones de DL, es necesario crear un framework en *Python* que se comunique con *C* (y por tanto con PAPI) para medir los contadores hardware del sistema. A dicha herramienta se la ha bautizado con el nombre de ***DL-Prof* (*Deep Learning Profiling*)**, y cuya arquitectura consta de los elementos que se indican a continuación y se describen en las secciones posteriores de este capítulo:

1. Librería en *C*.
2. Interfaz *C-Python*.
3. Implementación con *Keras (callbacks)*

La Figura 3-1 representa la arquitectura que tiene *DL-Prof* y cómo interactúa con las distintas herramientas para poder acceder a los contadores.

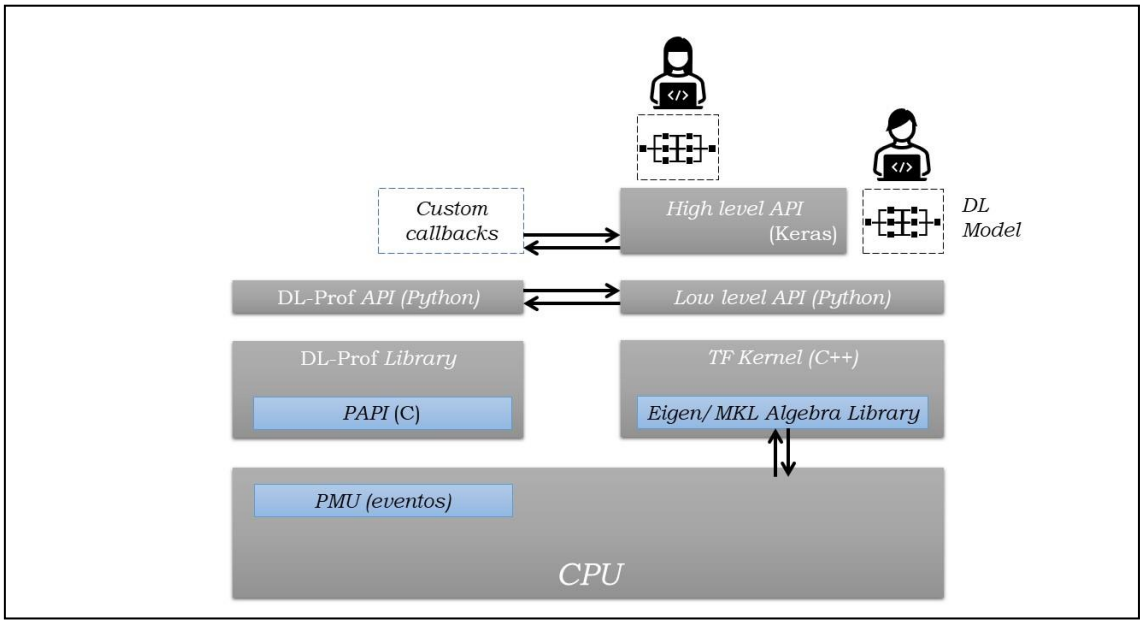

Figura 3-1: Arquitectura de la herramienta DL-Prof.

3.1.Librería en *C*

Para poder medir los contadores hardware del sistema partiremos de la librería PAPI. Esta herramienta cuenta con cientos de funciones y variables de utilidad, nuestra herramienta hará uso solamente de un subconjunto de ellas. Gran parte de la funcionalidad de PAPI requiere la modificación del código sobre el que se desea trabajar. Por ejemplo, para medir aplicaciones *multithread*, PAPI proporciona las funciones *PAPI_register_thread()* y *PAPI_unregister_thread()*; las cuales permiten añadir y quitar *threads* a la medida que se está realizando. Esto es algo impensable en el framework de *Deep Learning* con el que pretendemos trabajar, pues añadiría un nivel de complejidad muy elevado teniendo en cuenta que lo que se busca es que la herramienta sea fácil de utilizar, transparente al usuario y no afecte a las aplicaciones que se quieran medir. Razón por la cual, se optará por realizar medidas a nivel de *core*, utilizando las herramientas adecuadas para garantizar que los *cores* medidos son aquellos donde se ejecutan los *threads* bajo análisis (la sección 4.1 describe los mecanismos que se emplean con este fin). Para medir aplicaciones *multithread*, se hará uso de la función *PAPI_set_opt()* que permite asignar la medida a un *core* específico.

Los pasos seguidos para poder realizar una medida con PAPI son los siguientes:

1. Inicialización: antes de utilizar cualquier función, hay que cargar la librería con la llamada a *PAPI_library_init(version)*; la cual recibe por parámetro la versión de PAPI actual. Si se escribiese una versión incorrecta, el programa terminaría su ejecución lanzando un error.
2. Declaración del identificador del grupo de eventos: PAPI utiliza un entero como identificador para un conjunto de eventos (*Event Set*); el cual se usa para conocer

el estado de la medida (pausada, en marcha, finalizada, etc.) y lo que se está midiendo. Este identificador tiene que ser inicializado con la constante *PAPI_NULL* y pasar un puntero de él a la función *PAPI_create_eventset(*eventSet)*, para que se registre y se le asigne un valor. Si no se inicializase con la constante, la función retornaría un error.

3. Definición de eventos: posteriormente, a dicho "contenedor" de eventos hay que asignarle los eventos a medir (*PAPI_add_named_event(eventSet, events)*). Estos eventos son del grupo *native events*.
4. Medida: para empezar la medida y terminarla hay que llamar a las funciones *PAPI_start(eventSet)* y *PAPI_stop(eventSet, results)*, respectivamente.
5. Liberación: por último, para terminar de usar PAPI y liberar todos los recursos utilizados, se llama a la función *PAPI_shutdown()*.

Después de llamar a dichas funciones es necesario hacer una comprobación de que todo ha salido correctamente o al menos, eso es lo que se recomienda por parte del desarrollador. Como se puede comprobar, para realizar una única medida, los pasos a seguir no son triviales, y se requiere tener un cierto conocimiento de las funciones de la librería de PAPI.

Con el objetivo de que la herramienta libere al usuario de conocer ciertos detalles internos, se han creado unas funciones que harán de intermediarias, gestionando internamente la comprobación de errores y definición de constantes (por ejemplo, definir la versión actual de PAPI). En la Figura 3-2 se muestra, como ejemplo, una función de la librería *DL-Prof* que, para crear un conjunto de eventos, llama a la función de PAPI y comprueba que el resultado es correcto comparándolo con las constantes adecuadas (líneas 3-4).

```
1  int my_PAPI_create_eventset(int *EventSet)
2  {
3    int retval;
4    if ((retval = PAPI_create_eventset(EventSet)) != PAPI_OK)
5      ERROR_RETURN(retval);
6    return retval;
7  }
```

Figura 3-2: Comprobación de errores de la función PAPI_create_eventset() de PAPI.

El mismo procedimiento se ha repetido con todas aquellas funciones que han sido necesarias para poder conseguir una interfaz limpia y medidas correctas. El código correspondiente a esta funcionalidad se puede encontrar en los ficheros *MyPapi.c* y *MyPapi.py* del repositorio público. Estas funciones consiguen hacer el uso de la librería PAPI transparente para el usuario, que no necesita tener ningún conocimiento de su funcionamiento para llevar a cabo medidas con contadores. La interacción con la librería PAPI se gestiona a través de cinco funciones, siendo las únicas que el usuario debe conocer y manejar. Dichas funciones agrupan toda la gestión necesaria para las medidas y la funcionalidad de cada una se describe en la Tabla 3-1.

Tabla 3-1: Funciones de la librería en C.

Nombre	Inicialización
Declaración	*int my_prepare_measure(char *input_file_name, int num_cpus, int *cpus)*
Descripción	Se invoca antes de empezar una medida. Agrupa la llamada a las funciones de PAPI que cargan la librería, crean los conjuntos de eventos, asignan las medidas a cores y los eventos a medir. Todas las funciones que se hacen por separadas en PAPI, en esta función se juntan para que el usuario únicamente tenga que pasarle tres parámetros: el nombre de un fichero, que contendrá los eventos a medir (uno por línea), el número de cores que se van a medir y el identificador de éstos.
Nombre	Medida
Declaración	*int my_start_measure() / int my_stop_measure()*
Descripción	Las llamadas a estas funciones ponen en marcha/detienen la medida de eventos.
Nombre	Resultados
Declaración	*int my_print_measure(char *output_file_name)*
Descripción	Esta función recibe por parámetro una cadena de caracteres que se corresponde con la ruta al fichero donde se han de guardar las medidas realizadas. El fichero tendrá un formato CSV. Sin embargo, si el valor pasado fuese NULL, los resultados se mostrarán por pantalla en forma de tabla.
Nombre	Fin
Declaración	*int my_finalize_measure()*
Descripción	Se detiene la librería PAPI y se libera la memoria utilizada durante la ejecución del programa

Con estas funciones, se consigue que el usuario no necesite adquirir ningún conocimiento sobre el funcionamiento a bajo nivel de la librería PAPI, consiguiendo además una librería similar a la interfaz de alto nivel que cuenta PAPI y que no se usa por los motivos explicados en el capítulo anterior. Se ha elegido crear una librería dinámica compuesta por las funciones y variables que se han utilizado. Con esta elección, se consigue que *DL-Prof* sea más ligero y no tenga que hacer una copia de las funciones que se usan en el momento de enlace y compilación. Además, se consigue que la herramienta no esté sujeta a una versión determinada de PAPI. Las futuras actualizaciones afectarán únicamente a nuestra herramienta si se modifican los parámetros de entrada y salida de las funciones que se usan, algo que, normalmente, no se suele hacer por motivos de compatibilidad con versiones anteriores. Aparte las ventajas descritas que ofrece la implementación como librería dinámica, la conexión entre *C* y *Python* resultará más sencilla, tal y como se explica en el siguiente apartado.

3.2. Interfaz *C-Python*

La librería desarrollada tiene que poder ser llamada por aplicaciones de *Deep Learning*; o lo que es lo mismo, desde *Python*. Sin embargo, esto no es algo trivial dado que ambos lenguajes almacenan los datos de diferentes maneras. *C* reserva espacio en memoria de acuerdo con el tipo de dato declarado; sin embargo, en *Python* todo se gestiona como objetos y un tipo de dato equivalente puede tener un espacio de memoria reservado de tamaño diferente.

En vista de todos estos detalles se ha buscado una manera de crear una conexión entre ambos lenguajes, y la forma más fácil es mediante el uso de extensiones o módulos de *Python*. A ellos se les conoce como “enlaces de *Python*” (*Python bindings*) y permiten

llamar a funciones de *C*. De entre los módulos que existen, se han evaluado los dos que se describen en las siguientes secciones.

3.2.1. Cython

Tal y como es definido por sus desarrolladores [22], *Cython* es un *Python* con los tipos de datos de *C*. Se puede instalar mediante el administrador de paquetes de *Python* (*pip*) y requiere del compilador *GNU C Compiler* (*gcc*) pues, a diferencia de *Python*, éste debe ser compilado. Para su ejecución se usan dos tipos de archivos y se recomienda la inclusión de un tercero a cargo de las labores de compilación (una especie de *Makefile* para *Python)*. Además de poder llamar funciones de *C/C++* desde *Python*, permite hacer las mismas llamadas en sentido inverso (funciones de *Python* desde *C/C++*).
Tras implementar un programa de prueba (*Hola mundo*) y con todas las características que tiene el módulo se ha decidido no continuar y pasar a buscar otra opción. Antes de traducir el código fuente de *C* hay un proceso de optimización que no resulta adecuado en nuestro contexto, porque se quiere conocer qué es lo que se está ejecutando en todo momento con la herramienta. Es aquí cuando se ha empezado a ver que este conector no es el adecuado para nuestros fines.

Tabla 3-2: Funciones de la librería en Python.

Nombre	Creación
Declaración	MyPapi(lib_path)
Descripción	En la creación de un objeto de MyPapi es necesario indicarle la ruta de la librería compartida en C. Se crea un objeto de tipo CDLL (pues se está ejecutando el programa en *Linux* [esto podría cambiarlo para crear uno genérico. TODO para más adelante porque hay que modificar el código]) y se guarda como atributo de la clase. Además, se definen los parámetros de entrada y salida de las funciones de la librería en C.
Nombre	Inicialización
Declaración	*prepare_measure(events_file, cpus=None)*
Descripción	Se invoca antes de empezar una medida. Recibe por parámetro el fichero con los eventos a medir y una lista con las cpus donde medir. En caso de no pasarle dicha lista, se medirán todas las cpus del sistema donde se esté ejecutando.
Nombre	Medida
Declaración	*start_measure() / stop_measure()*
Descripción	Las llamadas a estas funciones ponen en marcha/detienen la medida de eventos.
Nombre	Resultados
Declaración	*print_measure(output_file=None)*
Descripción	Recibe por parámetro la ruta del fichero donde se han de guardar las medidas realizadas. El fichero tendrá un formato CSV. Sin embargo, si el valor pasado fuese *None* (o no se pasase ningún valor), los resultados se mostrarán por pantalla en forma de tabla a medida que se van obteniendo.
Nombre	Fin
Declaración	*finalize_measure()*
Descripción	Función que da por finalizada la medida de eventos.

3.2.2. *ctypes*

Es una librería que viene incluida al instalar *Python* y no necesita otro software adicional. Proporciona tipos de datos que son compatibles con los de *C*, además de permitir llamar a funciones de librerías compartidas (o *DLLs*). Para ello, primero se ha de crear un objeto de la clase *CDLL* (en *Linux*) o *WinDLL* (en *Windows*). Opcionalmente, se pueden definir los parámetros de entrada y salida de las funciones; y, por último, llamarlas como si se tratase de una función de otro objeto de *Python*.
Existen más conectores que pueden ser interesantes (y también más complejos), sin embargo, se ha elegido esta última opción porque no es necesario instalar ningún software adicional, y por su sencillez a la hora utilizarlo. Además, no aplica ninguna optimización/modificación sobre el código original de *C*; el papel que desarrolla es la de ser un mero intermediario entre la librería compartida y *Python*, justo lo que se busca.

Haciendo uso de la librería *ctypes*, el interfaz *C-Python* diseñado se reduce a cinco funciones con las que el usuario tendrá control total sobre el comportamiento de la medición de eventos desde una aplicación escrita en *Python*. La Tabla 3-2 define cada una de estas funciones y describe de forma breve su cometido.

3.3. *DL-Prof Callback*, integración con *Keras*

La interfaz descrita en la sección anterior permite interactuar con modelos de DNN desarrollados con la API de bajo nivel de *TensorFlow*. Como uno de los objetivos es minimizar los conocimientos necesarios sobre el desarrollo de este tipo de modelos, se han investigado las características que ofrece la API de *Keras*, encontrando un encaje interesante en la funcionalidad que desarrollan los *callbacks*. Estos pueden ser pasados a los métodos de *Keras* tales como *fit*, *evaluate* y *predict*, comentados en la sección 2.1.2, para conectarse con las distintas etapas del entrenamiento e inferencia del modelo, por las que pasa.

Keras permite la creación de unos *callbacks* personalizados que pueden realizar las operaciones que el programador estime oportuno. De tal manera, nos permite llamar a *DL-Prof* y medir eventos durante las distintas etapas de las aplicaciones de DL. Para crear un callback personalizado, es necesario crear una subclase que herede de la clase abstracta *keras.callbacks.Callback* y modificar las funciones que se correspondan con la etapa de interés, todas ellas se muestran en la Tabla 3-3.

Tabla 3-3: Funciones llamadas en las distintas etapas del entrenamiento de un modelo de ML

Etapa	Métodos		
	Globales	A nivel de *batch*	A nivel de *epoch*
Entrenamiento	*on_train_begin()* *on_train_end()*	*on_train_batch_begin()* *on_train_batch_end()*	
Evaluación	*on_test_begin()* *on_test_end()*	*on_test_batch_begin()* *on_test_batch_end()*	*on_epoch_begin()* *on_epoch_end()*
Inferencia	*on_predict_begin()* *on_predict_end()*	*on_predict_batch_begin()* *on_predict_batch_end()*	

```
1   class MeasureOnTrainPhase(MyCallbacks):
2     def __init__(self, lib_path, events_file, output_file=None):
3       super(MeasureOnTrainPhase, self).__init__(
4                     events_file=events_file, lib_path=lib_path,
5                     output_file=output_file)
6     def on_train_begin(self, logs=None):
7       self.mp.start_measure()
8     def on_train_end(self, logs=None):
9       self.mp.stop_measure()
10      self.mp.print_measure(self.output_file)
```

Figura 3-3: Callback personalizado que llama a DL-Prof durante la fase de entrenamiento.

Se ha seguido el modelo de herencia y se ha decidido crear una subclase de *keras.callbacks.Callback* llamada *MyCallbacks*, donde recibe por parámetro los valores necesarios para que *DL-Prof* pueda funcionar: *path* donde se encuentra la librería en *C*, fichero con eventos a medir y, opcionalmente, fichero donde se han de guardar las medidas. Esta subclase es la encargada de realizar las preparaciones para poder medir; de tal manera, se deja al usuario la libertad de crear una subclase que herede de *MyCallbacks* y modifique únicamente las funciones donde quiere medir (las presentes en la Tabla 3-3). Un ejemplo se muestra en la figura siguiente (Figura 3-3).

3.4.Problemas encontrados

Durante el diseño del *framework* han surgido ciertos contratiempos que han complicado el desarrollo del código. En ocasiones, dichos problemas han requerido modificaciones relevantes en la funcionalidad de la herramienta, necesitando llevar a cabo un proceso de adaptación del framework para asegurar un funcionamiento correcto. Las siguientes subsecciones describen en detalle el origen de los dos contratiempos más relevante, así como las soluciones adoptadas.

3.4.1. Recolector de basura (*garbage collector*)

El *Garbage Collector* (GC), presente en algunos lenguajes de programación como *Java*, *C#* y *Python* entre otros, evita que el usuario tenga la tediosa tarea de gestionar la memoria manualmente. Si consideramos que nuestra meta es la de abstraer todo lo posible al usuario de cómo está desarrollada por dentro la herramienta, el uso de este mecanismo es algo positivo. Supongamos el código de ejemplo mostrado en la Figura 3-4, escrito en *Python*, donde se llevan a cabo tareas previas al proceso de medida.

En la declaración de la función (línea 1) se observa el paso por parámetro de un array de enteros (*cpus*), que se corresponde con los identificadores de los *cores* del sistema que se pretenden medir. Tras una serie de comprobaciones necesarias, se crea una variable temporal (línea 5, *cpus_for_c*) donde se realiza una transformación para que el array pasado por parámetro se transforme en uno que pueda ser interpretado por *C*. El problema surge en la línea siguiente, cuando se pretende llamar a la función en *C*, utilizando como argumento dicha variable temporal. Durante el tiempo que transcurre entre su creación y su uso a través de la llamada a la función *C*, el GC de *Python* identifica dicho array como

posiciones de memoria que no se van a usar y libera su contenido. Tras esta operación, el array que se pasa como argumento deja de tener contenido alguno, produciendo un fallo de acceso inválido.

```
1   def prepare_measure(self, events_file, cpus=None):
2     # ... Check other params and cpus is not None ...
3
4     # These two lines will produce an error:
5     cpus_for_c = (c_int * len(cpus))(*cpus)
6     self.p_lib.my_prepare_measure(..., cpus_for_c)
7
8     # But these will work:
9     self.cpus_for_c = (c_int * len(cpus))(*cpus)
10    self.p_lib.my_prepare_measure(..., self.cpus_for_c)
```

Figura 3-4: Trozo de la función en Python que prepara el sistema para realizar las medidas.

Identificar al GC como culpable de este comportamiento no ha sido un proceso sencillo, y ha requerido un proceso de depuración relevante con múltiples comprobaciones y pruebas. Una vez detectado, hemos sido conscientes de que nos encontrábamos ante un problema habitual del módulo *ctypes*. La solución adoptada, descrita en la Figura 3-4 (líneas 9 y 10), consiste en la utilización de un atributo de clase en vez de variable temporal para guardar el contenido a pasar a *C*. Con ello conseguimos que el GC no pueda eliminar el contenido pues no sabe si se usará dicho atributo más adelante.

3.4.2. Memoria del sistema

Otro problema al que hemos tenido que enfrentarnos surge al utilizar el espacio de memoria del *heap* a través de las funciones *malloc()* y *free()* de *C*. La reserva de memoria dinámica resulta problemática porque el paso de punteros entre ambos lenguajes acarrea la liberación automática, por parte de *Python* o del módulo *ctypes*, de dicho espacio de memoria (de manera similar al problema anterior). Con lo cual, si se quiere hacer una llamada a la función *free()* desde *Python* al terminar la ejecución del programa, surge un error de doble liberación de memoria (uno hecho por la función free y otro hecho por el GC de *Python*). A primera vista podría parecer que la solución es sencilla, delegando la liberación de memoria completamente al GC de *Python*, pero utilizando este mecanismo la ejecución termina con un error de tipo *segmentation fault*.

Tras un análisis exhaustivo, se ha detectado que tras el error de liberación de memoria no se encuentra el GC de *Python* (GC), sino que su origen está en la librería *ctypes*. Queda fuera de los objetivos del TFG analizar el funcionamiento interno de esta librería, por lo que en este caso se ha optado por retocar la herramienta desarrollada en *C*. Se han cambiado aquellas reservas de memoria dinámica (no todas resultan en fallo, sólo la memoria que se reserva desde *Python*) a estática; haciendo necesario en este caso definir, en la librería misma, algunos aspectos relacionados con la máquina en la que se va a ejecutar la herramienta (el número máximo de *cores* o eventos a medir).

3.5. *Git* público

Antes de comenzar con la descripción de la herramienta, es necesario señalar que todo el código que se describe en este documento se encuentra alojado en un repositorio público (tanto la herramienta desarrollada como las pruebas y experimentos que se han llevado a cabo) por si fuera de utilidad en el desarrollo de investigaciones en el área y para su consulta por parte del tribunal. Las menciones que se hagan al código harán referencia a dicho repositorio, para que la longitud de este documento no sea extensa y para que pueda ser verificado por el lector y toda persona que busque un instrumento de estas características.

El enlace para acceder a él es el siguiente:

https://github.com/jlpadillas/DL-Prof

4 Comprobación de la herramienta

La librería y la interfaz desarrolladas han sido sometidas a un proceso de validación exhaustivo a medida que se iban desarrollando. Para ello se han utilizado diversos *micro-benchmarks*; programas con una estructura muy sencilla que permiten estimar, antes de su ejecución, el número de cierto tipo de eventos (cada *benchmark* se diseña para medir eventos distintos). En todos los casos evaluados, la medición de eventos obtenida con *DL-Prof* ha sido la esperada y en esta memoria incluimos, únicamente, uno de los experimentos desarrollados, dado el interés que pueden tener los resultados obtenidos.

4.1.Preparación previa

La medida de los eventos, asociados a un proceso concreto, es una tarea compleja. El *kernel* puede planificar la ejecución de procesos, diferentes al de interés, en el *core* sobre el que se cuentan los eventos; haciendo difícil distinguir qué eventos de la cuenta final han sido causados por nuestro código. Las instrucciones con valores de tipo flotante son utilizadas en contadas ocasiones por el Sistema Operativo, lo que minimiza las interferencias en el proceso de validación de nuestro *micro-benchmark*. En cualquier caso, para éste y todos los *benchmarks* ejecutados, siempre se han utilizado herramientas (disponibles en el SO) para aislar sus procesos de aquellos sobre los que se realiza la medida. El *scheduling* de los procesos propios del *kernel* se limita a ciertos *cores* (gestionando el parámetro *CPUAffinity* de *systemd*) y la ejecución de los procesos a medir se mapea sobre un grupo de *cores* disjunto (a través del comando *taskset* disponible en la *Shell* de *Linux*).

Tabla 4-1: Características del servidor utilizado.

Servidor	:	*Intel(R) Xeon(R) Silver 4216*
CPU(s)	:	*32*
Socket(s)	:	*2*
Core(s) por socket	:	*16*
Thread(s) por socket	:	*1*
L1d/L1i	:	*32KB*
L2	:	*1MB*
L3	:	*22MB*
RAM	:	*112 Gib*
Kernel	:	*Linux version 4.9.0-12-amd64*
GCC	:	*gcc (Debian 8.3.0-6) 8.3.0*
Python	:	*Python 3.7.3*

La Tabla 4-1 proporciona una descripción detallada del servidor que se ha utilizado para llevar a cabo todas las ejecuciones. En todos los casos, el Sistema Operativo se ejecuta sobre los *cores* físicos 0 y 1, y el *micro-benchmark*, en el resto. Se ha desactivado la opción de SMT (*Simultaneous Multithreading*) del procesador para poder medir más eventos de manera simultánea. Dada la variabilidad experimentada en los resultados en algunos casos, los *benchmarks* se ejecutan 10 veces, descartando el resultado mayor y menor y calculando la media de los resultados restantes.

El código de nuestro *benchmark* llevará a cabo una multiplicación de matrices cuyos elementos son de tipo "flotante", algoritmo para el que el número de operaciones es conocido, en función únicamente de las dimensiones de las matrices. Utilizaremos dos códigos con una sintaxis similar, escritos en lenguajes distintos, *C* y *Python*.

4.2. Multiplicación de matrices, *Python* vs *C*

Utilizamos esta prueba con dos objetivos: validar nuestra herramienta y analizar el rendimiento de dos lenguajes de programación de distinta naturaleza. Partiremos en ambos casos de una estructura de código similar (*main.c* y *main.py*), analizando los resultados de rendimiento y utilizando *DL-Prof* para justificar los mismos.

El programa presenta una estructura sencilla:

- Una primera fase de inicialización, en la que las matrices se rellenan con valores aleatorios de punto flotante.
- En la fase de ejecución, las matrices se recorren de la forma más favorable para aprovechar la localidad de los datos. Para ello, la mejor forma es recorrer la matriz resultante primero, luego la primera matriz (de donde se obtienen las filas) y la segunda matriz (de donde se obtienen las columnas). De tal manera el algoritmo utilizado sería kji, que, en términos de tasa de fallos, es lo mismo que ikj. Se podría haber obtenido aún más velocidad aplicando un algoritmo inconsciente de la cache (*Cache-oblivious*), pero no creemos que sea necesario tanto detalle en este pequeño *benchmark*.

4.2.1. Validación

En el caso de nuestro problema, sabemos el número de instrucciones de FP (*Floating Point*) estimadas en función únicamente del tamaño de las matrices a operar. Para calcular este valor recordemos que, en el producto de dos matrices, cada elemento (de la matriz resultante) requiere el mismo número de operaciones. En cada una de ellas, el número de multiplicaciones está directamente relacionado con el número de elementos en las filas de la primera matriz. Posteriormente, el resultado de dichos productos se tiene que sumar. Para facilitar este cálculo se hará uso de matrices cuadradas durante el experimento, quedando la ecuación de la siguiente manera (Donde la dimensión de la matriz resultante es n x n):

$$\mathbf{FP_{operaciones}}(\mathbf{M_{nn}}) = (n^2) \cdot \big(n + (n-1)\big) = \boldsymbol{n^2(2n-1)}$$

Probamos, tanto para *C* como para *Python*, si la ejecución y medida (en el caso de *Python* con nuestra herramienta) retorna el valor esperado. La Tabla 4-2 muestra los resultados obtenidos para los tamaños de matriz evaluados.

Los valores de la tabla nos muestran dos resultados que deben ser comentados. En primer lugar, las ejecuciones con ambos lenguajes arrojan el mismo número de instrucciones de punto flotante en todos los casos. Este resultado es el esperable, y una muestra de que *DL-Prof* está realizando la cuenta de eventos de manera correcta. En segundo lugar, podemos observar que existen diferencias entre las medidas que se han calculado y las estimadas para el algoritmo.

Tabla 4-2: Medidas estimadas y obtenidas en ambos lenguajes para un determinado tamaño de matriz.

Dimensión de la matriz	Estimación F.P.	Medida *C*	Medida *Python*
128	4.177.920	4.194.304	4.194.304
512	268.173.312	268.435.456	268.435.456
1.024	2.146.435.072	2.147.483.648	2.147.483.648
2.048	17.175.674.880	17.179.869.184	17.179.869.184

Se puede ver que a medida que aumenta la dimensión de la matriz, la diferencia también se incrementa: pasa de 16.384 para la matriz de 128x128 a unos 4.194.304 con la matriz de 2.048x2.048. Estos valores no son aleatorios y tienen una relación directa con la manera en la que se ha implementado el código. Para explicar mejor esto, en la Figura 4-1 se muestra el algoritmo utilizado para multiplicar las matrices. Como se ha comentado anteriormente, el orden en el que se han accedido a las matrices es mediante el algoritmo *ikj*, con lo que se obtiene menos búsquedas en las matrices. Sin embargo, el "problema" reside en la línea 5, donde se define una variable temporal llamada *sum* y se le da un valor igual a 0. En el bucle siguiente (líneas 6-9) se suma a dicha variable las operaciones propias de la multiplicación; y, es esa primera operación $(0.0 + \cdots)$ la culpable de que se añada una operación de punto flotante adicional. Como en la matriz se tienen n^2 elementos, la diferencia de eventos medidos es igual a dicho número. Esto es equivalente a ambos lenguajes pues, aunque en la Figura 4-1 se muestre el algoritmo en *C*, en *Python* es el mismo.

```
1   for (i = 0; i < rows_a; i++)
2   {
3     for (k = 0; k < cols_b; k++)
4     {
5       double sum = 0.0;
6       for (j = 0; j < cols_a; j++)
7       {
8         sum += M_a[i * cols_a + j] * M_b[j * cols_b + k];
9       }
10      M_c[i * cols_b + k] = sum;
11    }
12  }
```

Figura 4-1: Algoritmo ikj utilizado para multiplicar las matrices.

4.2.2. *Python* vs *C*

Aprovechamos el micro-benchmark de multiplicación y la versatilidad de nuestra herramienta para completar este sencillo experimento con información que puede resultar interesante al lector. *C* y *Python* son dos lenguajes de programación de naturaleza muy distinta. Tomando como punto de partida las versiones básicas de ambos programas, *DL-Prof* nos permite realizar medidas en el código en *Python* equivalentes a las realizadas en *C* a través de la librería PAPI. Comenzaremos evaluando dos métricas básicas de rendimiento como son el IPC y el tiempo de ejecución, midiendo los resultados obtenidos a medida que el tamaño de matriz aumenta. La Figura 4-2 muestra estos resultados, donde

en ambos casos el eje *x* representa la dimensión de la matriz (recordemos que es cuadrada) desde 32 hasta 4.096. El eje *y* de la figura izquierda representa el IPC, mientras que el de la derecha, el tiempo de ejecución normalizado a los resultados obtenidos con *C*.

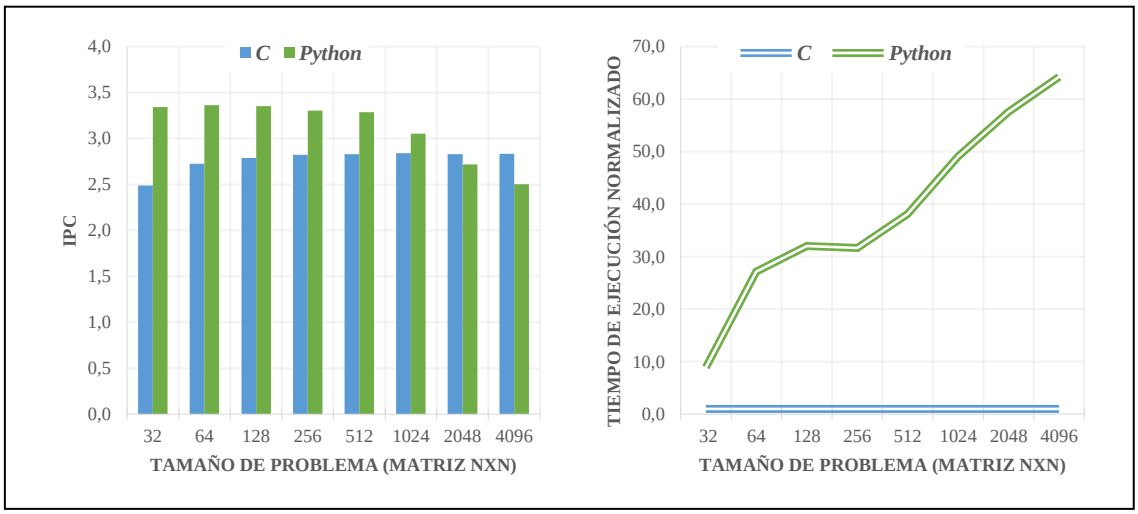

Figura 4-2: Métricas básicas de rendimiento para ambos lenguajes de programación.

Como podemos observar, si nos fijáramos únicamente en el IPC como métrica de rendimiento, creeríamos que ambos programas tienen un rendimiento similar. Lo que se puede observar a la vista de los resultados en la gráfica de la izquierda es que, con C, el rendimiento se mantiene estable a partir de un tamaño de problema de 64 y que de ahí en adelante no se ve afectado por el tamaño de la matriz. En *Python*, sin embargo, ocurre algo distinto; y es que, a medida que el tamaño de la matriz aumenta, el rendimiento va decreciendo.

El IPC solamente es una métrica válida cuando se compara el mismo conjunto de instrucciones, por lo que no es apropiada en nuestro caso. De hecho, los resultados son completamente distintos si analizamos una métrica de rendimiento mucho más significativa, como es el tiempo de ejecución. La gráfica de la derecha en la Figura 4-2 muestra estos resultados. En este caso el tiempo de ejecución del problema, con *Python*, aumenta a medida que la matriz a multiplicar aumenta su dimensión: pasa de un *Slowdown* de 10, para una matriz de 32x32, hasta 64 para una matriz de 4.096x4.096.

La herramienta diseñada nos permite obtener información que explique estos resultados. Para ello, analizaremos en ambos lenguajes el número total de instrucciones ejecutadas para llevar a cabo la multiplicación, así como la mezcla de estas (FP, aritméticas, load/store y saltos). Los resultados de dicho análisis se muestran en el gráfico de la Figura 4-3. De nuevo, el eje *x* muestra los distintos tamaños de problema, mientras que en el eje *y* representamos el número de instrucciones de cada tipo por cada instrucción de punto flotante (este número es igual en ambos lenguajes, como vimos en la Tabla 4-2). Como podemos observar, parece claro que gran parte del *overhead* observado en el tiempo de ejecución es debido al elevado número de instrucciones que se ejecutan para el código en *Python*. La diferencia llega a ser de dos órdenes de magnitud en todos los casos; superando las 200 instrucciones aritméticas por cada instrucción de punto flotante, mientras

que en *C* se mantiene sobre las 4,5 instrucciones del mismo tipo. Otras métricas importantes, como lo son los *load/store* de datos se ven totalmente incrementados en *Python*, llegando a ser 40 veces más que en el otro lenguaje. Por último, las instrucciones de salto en *C* no superan la unidad, pero sí llegan a las 150 instrucciones de salto por cada instrucción de FP.

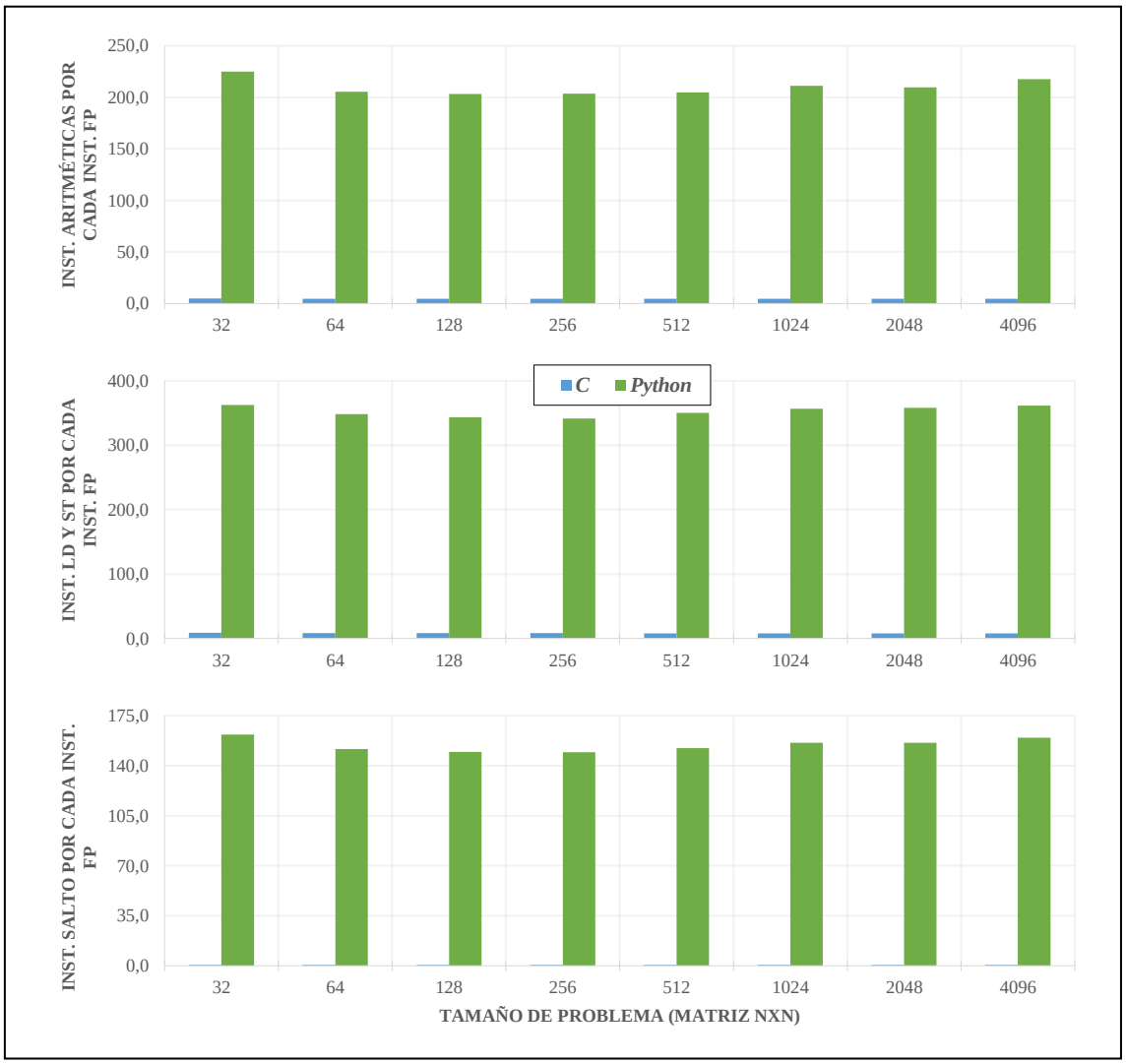

Figura 4-3: Instrucciones generadas por cada instrucción de punto flotante en C y Python. Arriba instrucciones aritméticas, en medio instrucciones de Load y Store, y, abajo instrucciones de salto.

Todos estos resultados son los esperados, debido a las diferentes características de ambos lenguajes. Uno es un lenguaje de programación compilado y otro un lenguaje interpretado. Esto quiere decir que *C* necesita que el código fuente se compile (con el compilador *gcc*, por ejemplo) y se genere un código máquina para que pueda ejecutarse. Si fuese necesario modificar el código, haría falta volver a compilar el código fuente y ejecutarlo nuevamente. Por otra parte, *Python* esconde el proceso de "traducción" de lenguaje de alto nivel a lenguaje máquina, operación que se lleva a cabo de manera implícita al invocar el siguiente comando en la consola: $python\ main.py$; donde $main.py$ es el nombre del fichero que se desea ejecutar. El proceso de traducción, que se lleva a cabo de manera transparente al usuario, se muestra en la Figura 4-4: donde el código fuente se compila

primero a un código de bytes (*byte-code*) y luego es interpretado por el PVM, generando el código binario que ejecutará el ordenador. *Python* toma un tiempo de CPU significativo para la interpretación, lo que explica en gran parte los resultados obtenidos.

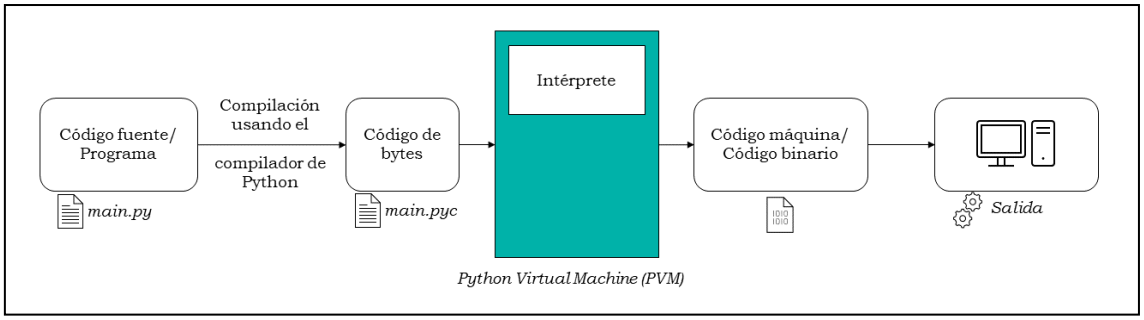

Figura 4-4: Proceso de ejecución de un programa en Python.

Queda todavía pendiente por explicar por qué el *overhead* de *Python* crece con el tamaño de problema. A la vista de los resultados en la gráfica izquierda de la Figura 4-2, la jerarquía de memoria podría estar detrás de este comportamiento porque a medida que el tamaño de la matriz crece, el rendimiento empeora.

Nuestra herramienta permitiría seguir explorando para confirmar nuestra hipótesis, con un análisis detallado de los contadores relacionados con el rendimiento de los niveles de cache del procesador. El objetivo de este TFG es mostrar la versatilidad de nuestra herramienta, por lo que hemos decido invertir el tiempo de trabajo en ampliar el número de experimentos.

4.2.3. Optimizaciones

El punto de partida ha sido muy básico porque se quería tener control sobre lo que se está ejecutando en todo momento; sin embargo, no es la manera habitual de trabajar con este tipo de algoritmos. Por ello, vamos a probar diferentes técnicas de optimización que ayudan al rendimiento, intentando comprender de nuevo (gracias a *DL-Prof*) cuales son las fuentes de mejora cuando se utilizan este tipo de herramientas.

Durante el proceso de compilación de *C*, se puede indicar al compilador que aplique un cierto nivel de optimización. El tercer nivel es el máximo, y activa una serie de funciones que permiten acelerar la ejecución del programa a cambio de un posible incremento del espacio que ocupará el binario. En concreto, se aplicará dicha optimización sobre los ficheros *matrix.c* y *main.c* con el fin de que la generación de las matrices y la multiplicación pueda realizarse en menos tiempo. En *Python*, es habitual el uso de librerías especializadas para la realización de operaciones algebraicas. Una de las más conocidas es *NumPy*, que da soporte para crear matrices y vectores multidimensionales grandes, junto con una gran colección de funciones matemáticas para operar con ellas. Lo que esta librería hace por debajo es operar todo lo posible con *arrays* o matrices en vez de escalares, confiando en BLAS y LAPACK para ejecutar las operaciones más comunes de algebra lineal (suma y multiplicación de vectores, combinación lineal, multiplicación de matrices, etc.). Estas dos librerías (BLAS y LAPACK) cuentan con rutinas de bajo nivel, en C y FORTRAN (haciendo uso de *bindings*, como el realizado en este TFG), que permiten optimizar las

operaciones para obtener una mejora en cuanto al tiempo de ejecución. Para ello hace uso de dos técnicas de mejora de rendimiento básicas, paralelización y vectorización (registros vectoriales e instrucciones SIMD).

Vamos a comparar los resultados obtenidos en la sección previa con las dos nuevas optimizaciones al multiplicar las matrices. La Figura 4-5 analiza el rendimiento para tres de los cuatro casos analizados (*C*, *C-O3* y *NumPy*), y podemos observar que, con las optimizaciones, el tiempo de ejecución se reduce notablemente.

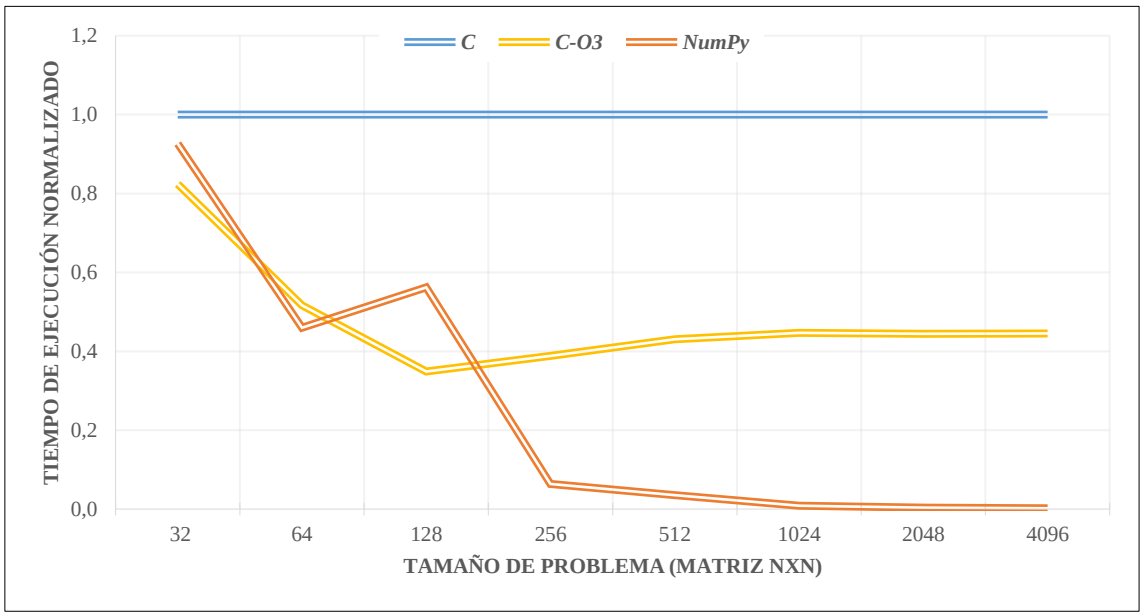

Figura 4-5: Tiempo de ejecución normalizado que se tarda en multiplicar matrices en C, C con optimizaciones de nivel 3 y con NumPy.

Ya desde el primer valor medido, se observa una ligera mejora con ambas optimizaciones, la cual se va haciendo más notoria cuando mayor es el tamaño de las matrices a multiplicar. Si nos fijamos en la optimización *C-O3*, podemos ver que se llega a un punto (a partir de matrices de dimensión 128x128) en el que no es posible reducir la diferencia de tiempo con su versión original, y se mantiene constate. Esto tampoco es despreciable pues se consigue que el programa termine en menos de mitad del tiempo que ha tardado con *C*. Por otra parte, mirando los resultados de la optimización que aplica *NumPy*, no es de extrañar que sea tan popular puesto que llega a alcanzar un *Speedup* de 500 para una multiplicación de matrices de tamaño 4.096.

En la Figura 4-5 puede que un dato haya llamado la atención al lector: la cresta que aparece al multiplicar matrices cuadradas de tamaño 128. De nuevo, gracias a nuestra herramienta, podemos intentar identificar a qué se debe este comportamiento extraño y el comportamiento general de los resultados obtenidos con las optimizaciones. Repetimos el análisis de instrucciones previo, y en los nuevos resultados (Figura 4-6) observamos lo que más o menos se preveía: que las optimizaciones ejecutan menos instrucciones por cada instrucción de punto flotante.

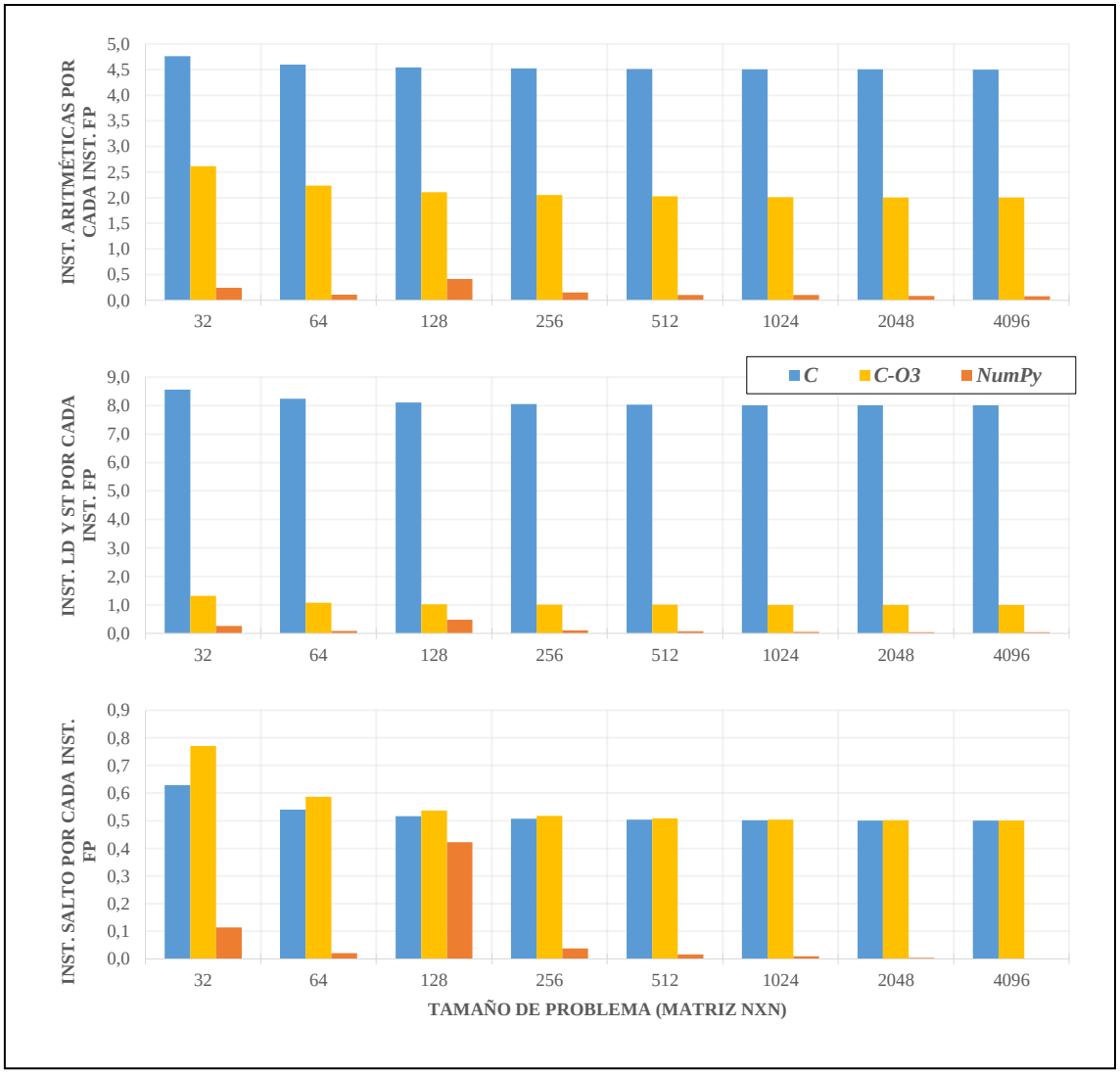

Figura 4-6: Instrucciones generadas por cada instrucción de punto flotante en C, C optimizado y NumPy. Arriba instrucciones aritméticas, en medio instrucciones de Load y Store, y, abajo instrucciones de salto.

En el caso de las aritméticas, todas las medidas van reduciéndose ligeramente a medida que se va aumentando el tamaño de las matrices a multiplicar. Saltando a la vista que se consiguen ejecutar la mitad de las instrucciones con *C-O3* y apenas unas décimas con *NumPy*. Este mismo patrón se repite para las instrucciones de *loads/stores*. Pero para los saltos ocurre un caso extraño, o no. Porque las instrucciones de saltos son prácticamente las mismas, sino más (al principio), para *C-O3*. Y en todas las tres gráficas se puede seguir viendo la anomalía que se arrastraba desde la Figura 4-5: el pico de instrucciones en *NumPy* está presente para matrices de dimensión 128x128. Esto es algo a lo que nuestra herramienta puede dar respuesta, pero que queda fuera de los objetivos principales de este TFG y no se ha considerado apropiado dedicarle más tiempo. Concluimos que gracias a las librerías que tiene por debajo *Python*, el algoritmo va a ser óptimo pues éstas optimizan todo lo posible las operaciones algebraicas. Además, las técnicas de vectorización, y en cierta medida la de paralelización, ayudan a obtener un mejor rendimiento reduciendo el tiempo de ejecución y el número de instrucciones ejecutadas.

Cerramos esta sección garantizando el correcto funcionamiento de nuestra herramienta, y demostrando su versatilidad para la realización de un gran abanico de experimentos. El siguiente capítulo de esta memoria se centrará en el diseño de varios experimentos para el análisis de un tipo de aplicaciones concretas, relacionadas con el campo del *Deep Learning*.

5 Evaluación

Una vez completado el proceso de validación para asegurar el correcto funcionamiento de la herramienta, se puede pasar a realizar el grueso de experimentos que se tiene en mente y que pondrá a prueba la herramienta con aplicaciones de *Deep Learning*. Para realizar estos experimentos se han buscado modelos y ejemplos de aprendizaje automático en su fase de entrenamiento. El entrenamiento es una fase con un coste computacional muy elevado y es relevante medir y comprobar el comportamiento que tiene la CPU durante la misma. *TensorFlow* tiene una colección de modelos que usan sus funciones de alto nivel y que son representativos de los principales campos de aplicación del Deep Learning.

TensorFlow Model Garden [23] cuenta con modelos destinados a la clasificación de imágenes, detección y segmentación de objetos, procesamiento natural del lenguaje y modelos destinados a realizar recomendaciones. De entre todos ellos, se ha optado por tomar como punto de partida uno de los modelos más sencillos y conocidos: MNIST. Dicho modelo recibe su nombre por el uso de la *base de datos MNIST* (*Modified National Institute of Standards and Technology database*) [24]. Se trata de una base de datos de imágenes de números escritos a mano, con un total de 60.000 imágenes destinadas para la fase de entrenamiento y 10.000 para la fase de evaluación (o prueba). Todas las imágenes han sido pre-procesadas, normalizando su tamaño, centrando el dígito y unificando su tamaño a 28x28 píxeles.

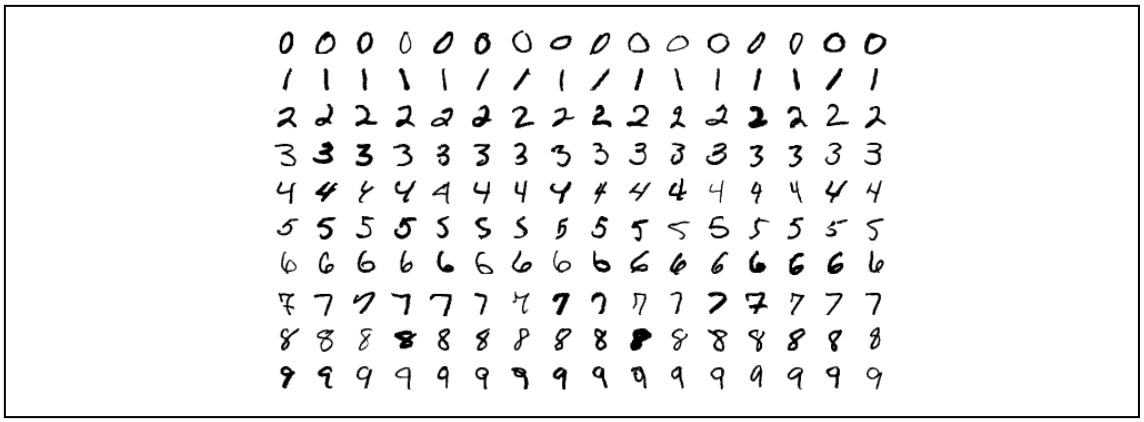

Figura 5-1: Algunas imágenes de números en la base de datos MNIST.

Además de las características del servidor ya comentadas en la Tabla 4-1 y los métodos usados para medir con *DL-Prof* (Sección 4.1); añadimos en este punto algunos parámetros adicionales de *TensorFlow* que son importantes y que pueden influir a la hora de hacer los experimentos. Para garantizar la compatibilidad de versiones entre los paquetes *Python* (*pip*) se ha trabajado en un entorno virtual (*virtualenv*) que cuenta con los paquetes requeridos por *TensorFlow* para poder ejecutar sus modelos de prueba. En concreto, se ha usado la versión 2.5.0 de *tensorflow*, con sus librerías matemáticas compiladas para hacer uso de las extensiones vectoriales del procesador (AVX512). El modelo de red es

del tipo *Convolutional Neural Network* (CNN), cuenta con nueve capas (Figura 5-2) y una configuración de hiperparámetros por defecto que se describen en la Tabla 5-1.

Tabla 5-1: Hiperparámetros usados para MNIST.

Hiperparámetro	***Valor***	***Definición***
Batch size	1.024	Define el número de muestras que se propagarán a través de la red entre cada actualización de pesos (*backpropagation*).
Epochs	5	También definido como "el número de pasadas sobre todo el conjunto de datos".
Learning rate	*tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=0.05, decay_steps=100000,decay_rate=0.96)*	Controla cuánto cambiar el modelo en respuesta al error estimado cada vez que se actualizan los pesos.

Lo que buscamos en este último capítulo es demostrar la gran cantidad de experimentos que se pueden llevar a cabo con *DL-Prof* y que nos ayudan a comprender cómo se comporta el código de las aplicaciones de Deep Learning cuando se ejecuta en una CPU. Para ello se ha diseñado tres experimentos sencillos que dan buena idea del potencial y versatilidad que tiene *DL-Prof*. En el primer experimento analizaremos los ámbitos de medida accesibles a través de los *Callbacks* de *Keras*. La siguiente sección evaluará la evolución temporal del rendimiento a lo largo de una época completa. Finalizaremos analizando el efecto sobre el rendimiento del valor de los hiperparámetros de configuración del modelo.

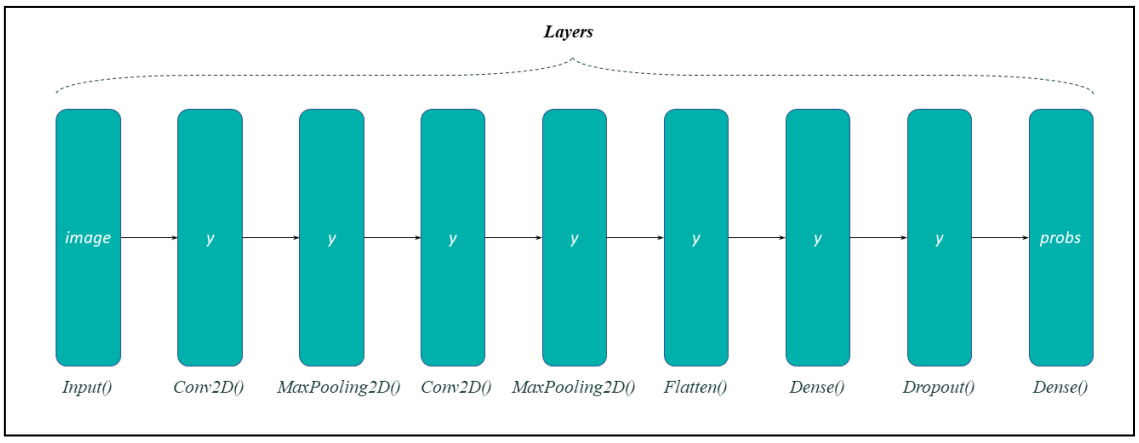

Figura 5-2: Representación de las capas presentes en el modelo MNIST, así como los tipos de cada una y la relación con las variables en el código de Model Garden.

5.1.Experimento 1: Ámbitos de medida

Tal y como se describió en la sección 3.3 la API de *Keras* proporciona una herramienta denominada Callbacks para realizar acciones definidas por el programador en diferentes puntos del proceso de entrenamiento. Hay tres niveles de detalle para estas acciones, entrenamiento completo, por época y por *batch* (ver Tabla 5-1). Desde la perspectiva del modelo, el entrenamiento completo es la suma de las épocas, y cada época es la suma de los *batches*. Nuestro objetivo es comprobar si esto también se cumple para los eventos hardware, analizando si dichas "sumas" se cumplen.

5.1.1. Entrenamiento completo frente a épocas

El punto de partida es medir algunos eventos importantes del sistema, como ya se hizo para la multiplicación de matrices, durante toda la fase de entrenamiento. Para esta medida se ha creado un *callback* personalizado que tiene por nombre *MeasureOnTrainPhase()*, y, que está presente en la herramienta *DL-Prof*.

La segunda medida planteada es sobre cada una de las cinco épocas del modelo, para ello se llama al *callback MeasureOnEachEpoch()*. Con ambos resultados sumamos las instrucciones obtenidas en cada época y comparamos los resultados con los del ámbito 1 para ver si hay alguna diferencia. El resultado de esta comparación se puede apreciar en la Figura 5-3, donde el eje x muestra los distintos tipos de instrucciones que se han medido y el eje y la cuenta de eventos para cada uno de ellos. Los eventos se han normalizado al valor del entrenamiento completo.

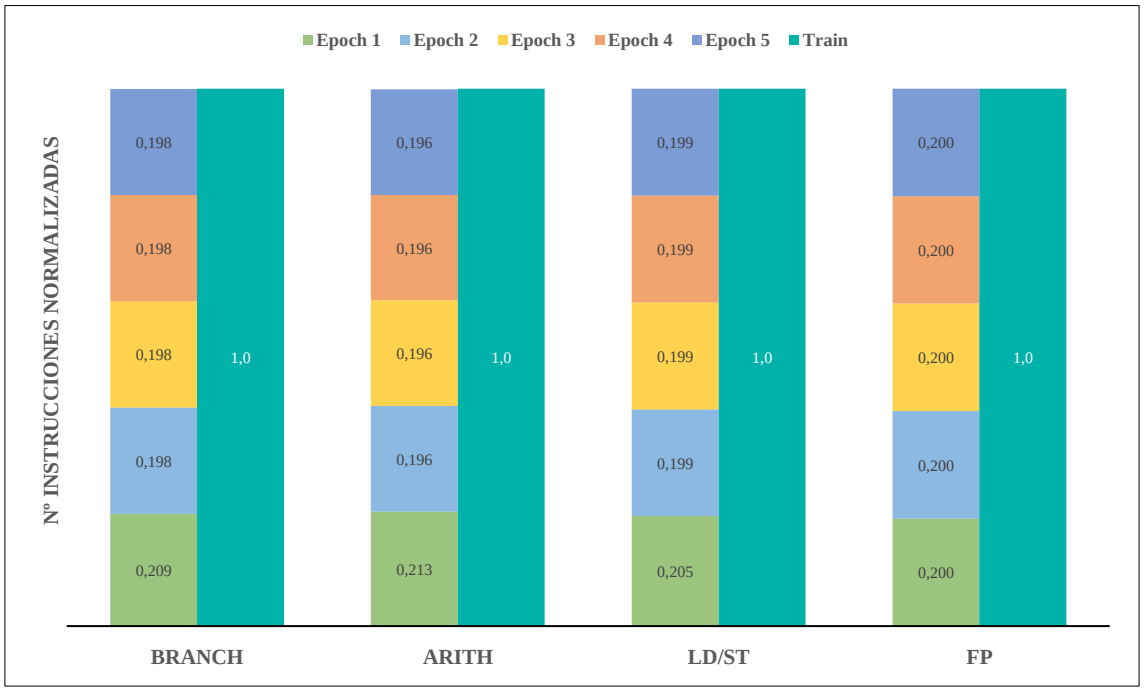

Figura 5-3: Eventos medidos durante toda la fase de entrenamiento (train) y durante cada una de las cinco epochs.

En este primer caso, los resultados coinciden con lo esperado, la suma de los eventos en cada época es prácticamente lo mismo a lo medido durante toda la fase de entrenamiento. Existen ligeras diferencias para las instrucciones aritméticas, pero no suponen ni un 0,11% del total. Lo que sí llama la atención es que durante la primera época (en color verde) se llevan a cabo más instrucciones que en las siguientes; a excepción las instrucciones de punto flotante, que se distribuyen homogéneamente durante las cinco partes. Es probable que, al tratarse de la primera época, algún proceso de inicialización sea el causante del incremento observado.

5.1.2. Época frente a *batches*

En este ámbito se profundiza aún más dentro de la fase de entrenamiento y se procede a medir los *batches* que están presentes en el primer *epoch* para compararlos con los

resultados obtenidos por el callback para esa época. En *DL-Prof*, el callback que permite medir por batches se llama *MeasureOnEachBath()*.

El resultado esperable en este caso sería el mismo del apartado anterior. La Figura 5-4 nos presenta esta comparación, y se puede observar que el resultado de la suma de batches no coincide con las métricas de la época. De los tipos de instrucción analizados el más llamativo es el resultado de las de punto flotante (FP). En principio, todas las operaciones de este tipo deberían corresponder al trabajo de los batches, por lo que la suma debería ser equivalente.

Esto resultaba sorprendente, por lo que se repitió la prueba midiendo todos los eventos de FP de los que dispone el servidor; con ello conseguimos estar seguros de que no pasamos por alto algún evento que deberíamos tener en cuenta. La medida se rehizo sobre toda la fase de entrenamiento, pero los únicos contadores que aportaron algún valor distinto de cero fueron los que ya se medían desde un principio. Con lo cual, la medida de FP es correcta en cada *batch* pero sin embargo, la suma de todas ellas no da lo que debería dar. Según lo visto en la figura del ámbito anterior, la primera época se comporta de una manera distinta a las siguientes; con lo cual, el problema puede residir únicamente en esa etapa y no en las siguientes. Se repiten las medidas sobre la época inmediatamente posterior y se representa en un nuevo gráfico; el cual no incluimos pues es muy parecido al de la Figura 5-4. Aquí, surge una hipótesis de lo que está sucediendo: que las medidas hechas dentro de un *epoch* incluyen un cierto *overhead* que no es capturado por la medida en los *batches*.

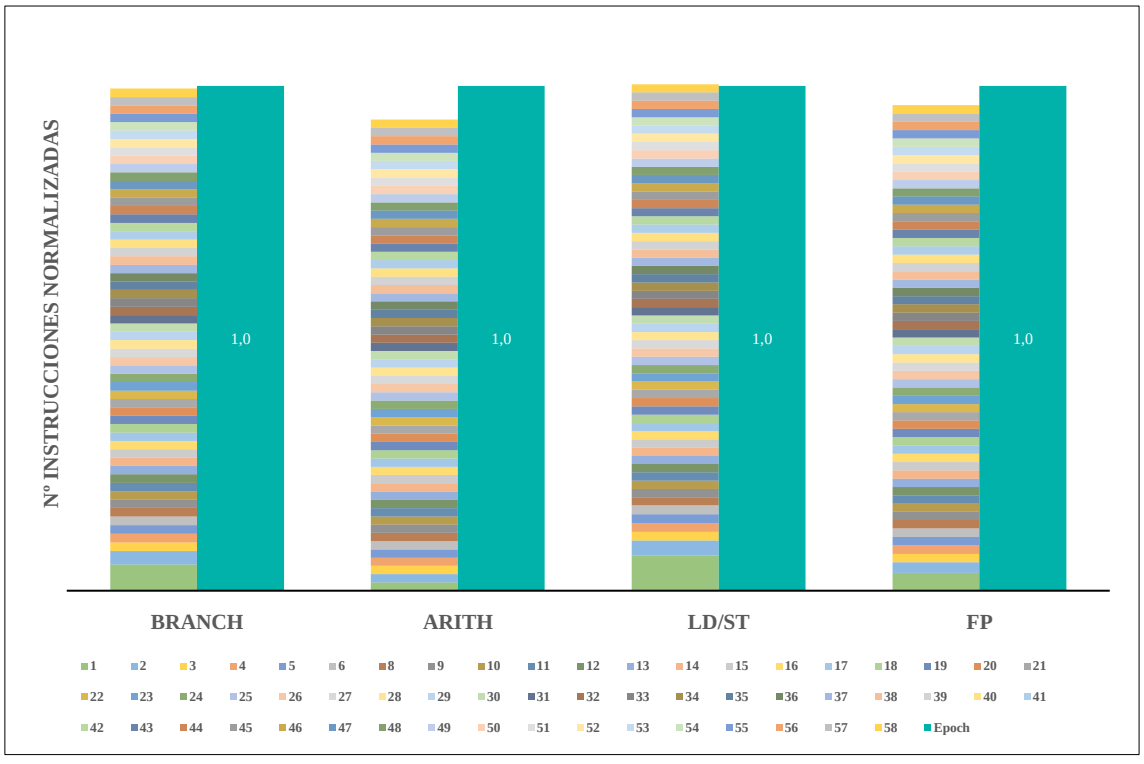

Figura 5-4: Eventos medidos durante el primer epoch y durante cada uno de sus 58 batches.

Otro resultado interesante, igual que en el experimento anterior, es la diferencia que se observa entre el primer batch de la época y el resto. El primer batch ejecuta un número más alto de operaciones que el resto, lo que se puede explicar de nuevo por los procesos de inicialización que tienen lugar en este punto.

5.2. Experimento 2: Evolución temporal

El siguiente experimento con MNIST intentará identificar, para este tipo de aplicaciones, qué elementos de la microarquitectura pueden suponer un problema para el rendimiento. Para ello, utilizaremos una metodología conocida como *TopDown* [25], apropiada para procesadores de propósito general. Esta metodología hace uso de los contadores hardware del procesador, y analiza los componentes de la arquitectura de manera jerárquica para identificar los cuellos de botella. Esta jerarquía tiene varias capas y múltiples componentes, pero en este análisis nos limitaremos a los elementos de primer nivel, que se describen a continuación.

El objetivo es clasificar la actividad del pipeline del procesador, evaluando qué factores lo impiden llegar a su máximo rendimiento. En el primer nivel, las pérdidas de rendimiento se agrupan en las siguientes tres clases:

- Frontend Bound: el *frontend* no es capaz de proporcionar instrucciones suficientes a las unidades funcionales. El ejemplo más común son los fallos en la cache de instrucciones.
- Backend Bound: Hay instrucciones disponibles para ejecutar, pero no avanzan por el pipeline. La falta de unidades funcionales disponibles para realizar una operación es un ejemplo.
- Bad Speculation: Se arranca la ejecución de instrucciones especulativas (predicción de saltos) que no llegan a finalizar su ejecución.

A través de los contadores adecuados, realizamos un análisis *TopDown* de las tres primeras épocas del entrenamiento, mostrando los resultados para cada *batch*. La Figura 5-5 muestra estos resultados; donde el eje vertical representa los campos descritos, además del campo RETIRING, que representa el porcentaje de rendimiento obtenido por la aplicación sobre el máximo teórico (el valor 1 implicaría que finalizan su ejecución 4 instrucciones en cada ciclo).

Como veíamos en los resultados anteriores, la primera época presenta diferencias con respecto a las siguientes. En este caso, la gráfica superior de la Figura 5-5 cuenta con más pérdidas de rendimiento en el *FE Bound* que las épocas siguientes; es decir, que la hipótesis que planteábamos en apartados anteriores tiene sentido. Y es que, el incremento de instrucciones que veíamos en la Figura 5-2, para la primera época, se debe a que tanto instrucciones como datos aún no se encuentran en la jerarquía de memoria; y el sistema tiene que esperarles para poder continuar.

Otro aspecto que llama la atención mirando las tres gráficas es que, en todas ellas, tres quintas partes del rendimiento, el programa los pierde en el *Backend Bound*. Es decir, que hay instrucciones, probablemente de punto flotante (pues las aplicaciones de DL hacen un gran uso de ellas), que forman un cuello de botella pues no existen suficientes unidades

funcionales que puedan darles salida; o bien porque hay problemas con la jerarquía de memoria.

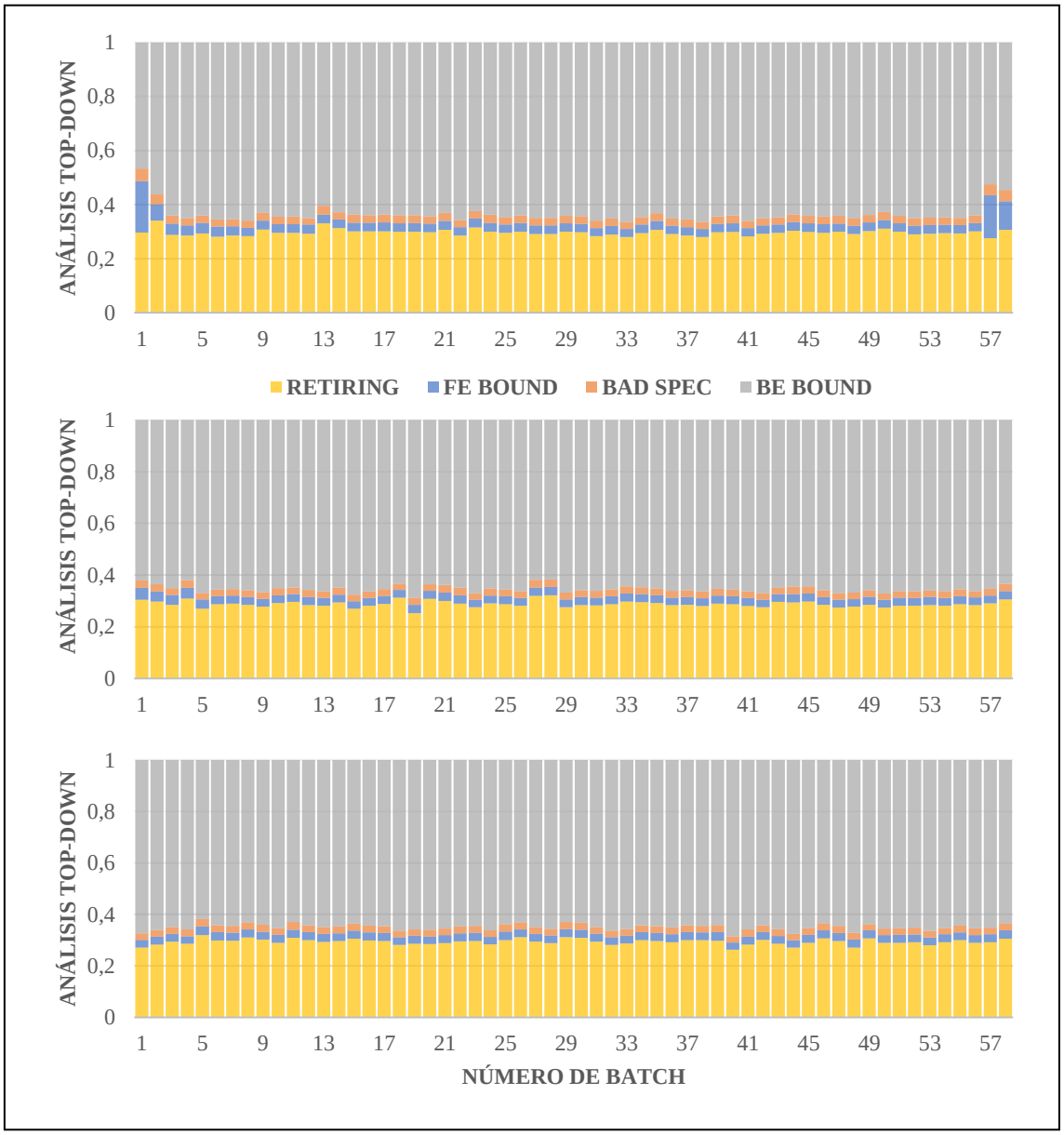

Figura 5-5: Análisis con la metodología top-down para las tres primeras épocas. Arriba la primera, en medio la segunda y abajo la tercera época.

5.3.Experimento 3: Hiperparámetros

Para terminar esta serie de experimentos, vamos a discutir sobre la importancia que tienen los hiperparámetros ya no sólo para obtener un correcto entrenamiento de los datos, sino también para ver cómo afectan al rendimiento durante el mismo.

De esta manera, se ha decidido medir el rendimiento durante las dos primeras modificando el tamaño de *batch* de manera progresiva. En la Figura 5-6 representamos en primer lugar el rendimiento (IPC) obtenido en cada época para un tamaño de *batch* que crece desde los dos elementos hasta los 16K. Adicionalmente, la misma figura incluye

resultados sobre cómo evoluciona la precisión del entrenamiento (pérdida y precisión) para el tamaño de *batch*.

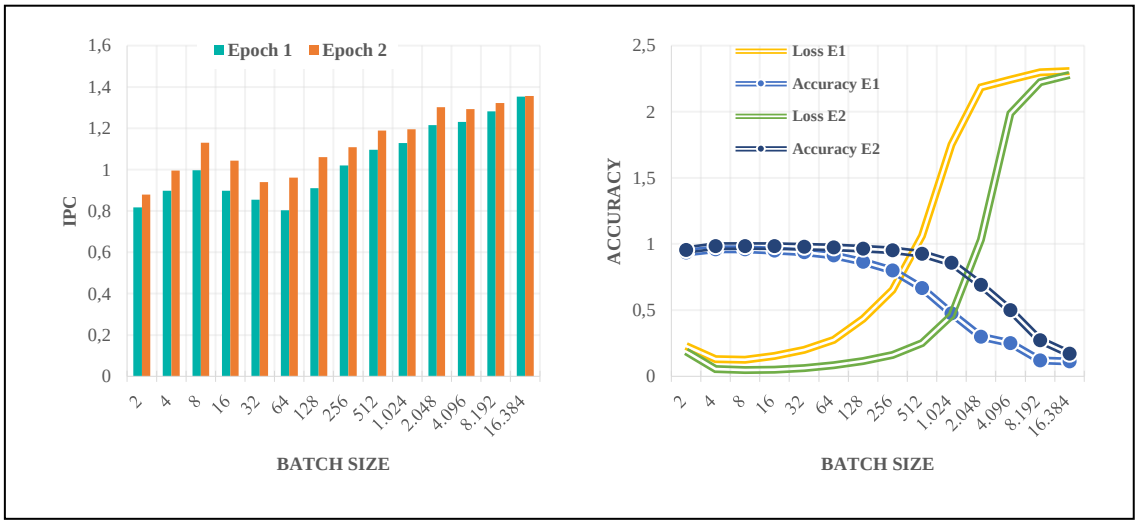

Figura 5-6: A la izquierda, IPC obtenido para las dos primeras épocas con distintos tamaños de batch. A la derecha, variación de las pérdidas y aciertos para las dos primeras épocas con distintos tamaños de batch.

Se puede observar una ligera diferencia del IPC entre ambas épocas, que probablemente se pueda explicar por las peculiaridades que hemos observado previamente en la primera época, así como la pérdida de rendimiento asociada a aspectos como el "calentamiento de la jerarquía de cache (fallos de calentamiento). Es decir, en la segunda época, algunos datos (como los pesos) e instrucciones ya están cargados en la cache; con lo cual el tiempo promedio de acceso a memoria disminuye y se obtiene un IPC mayor.

Otra característica interesante de la gráfica de rendimiento (IPC) es que se pueden apreciar dos patrones de comportamientos: uno para los tamaños de *batch* pequeños (desde el 2 al 32) y otro para los tamaños más grandes (32 hasta 16.384). La primera de ellas tiene una subida y bajada (formando una montaña), mientras que la segunda parece tener una pendiente creciente. Nuestra hipótesis es que este comportamiento está relacionado, otra vez, con la jerarquía de memoria; pero para conocer las causas últimas de este comportamiento habría que hacer más pruebas incluyendo contadores asociados a la jerarquía de memoria.

6 Conclusiones

En este trabajo hemos desarrollado una herramienta, *DL-Prof*, que permite recopilar información de la CPU cuando se ejecuta un determinado programa, y dentro de él, una determinada sección o región de interés. Esta información está relacionada con los eventos micro-arquitecturales; y, a partir de ellos poder obtener un análisis del rendimiento del software en cuestión. Se ha empezado midiendo las regiones de interés de programas escritos en *C* y en *Python*; y dada la relevancia que están teniendo las aplicaciones de aprendizaje automático, se pasa a medir el comportamiento del sistema cuando se ejecutan éstas. Para ello, hemos validado la herramienta mediante el desarrollo y ejecución de una serie de micro-benchmarks. Con los cuales se calculan teóricamente los eventos que se deberían ejecutar y se comparan con las medidas obtenidas, dando en todos los casos el resultado esperado. A partir de aquí, se plantean una serie de experimentos sencillos para demostrar la versatilidad que tiene la herramienta.

Esta serie de experimentos es sólo una pequeña muestra de la cantidad de ensayos que se pueden obtener, y de las aplicaciones que se pueden medir. Como trabajo futuro, se plantea el completar los experimentos que se han dejado a medias y que implican el profundizar aún más sobre el hardware para responder a algunas preguntas que han quedado al aire; esto es posible pues las métricas del sistema nos permiten seguir indagando. También se propone el realizar estos experimentos y aumentarlos para otras aplicaciones de DL presentes en *Model Garden*.

Con *DL-Prof* se ha ido descendiendo desde la fase de entrenamiento hasta llegar a la granularidad de *batch*. Además, se plantea el poder seguir ahondando en más detalle y poder incorporar a la herramienta la posibilidad de medir por capas (*layers*). Comprobar el efecto de utilizar una determinada función de pérdida o intentar encontrar el equilibrio con los hiperparámetros para obtener tanto un buen rendimiento (IPC y tiempo de ejecución principalmente) como una buena tasa de acierto.

7 Referencias

[1] W. Aspray, "The Intel 4004 Microprocessor: What Constituted Invention?", IEEE Annals of the History of Computing, Volume 19, Issue 3, Julio 1997.

[2] U. Verner, A. Mendelson, A. Schuster, "Extending Amdhal's Law for Multicores with Turbo Boost", IEEE Computer Architecture Letters, Volume 16, Issue 1, Enero 2017.

[3] T.M. Aamodt, W. WL. Fung, T. G. Rogers, M. Martonosi, "Genearl-Purpose Graphics Processor Architecture", Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2018.

[4] A. Krizhevsky, I. Sutskever, G.E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", International Conference on Neural Information Processing Systems, Diciembre 2012.

[5] A.M. Turing, "Computing Machinery and Intelligence", Mind, Volume LIX, Issue 236, Octubre 1950.

[6] .T. Brown, et. al. "Language Models are Few-Shot Learners", Advances in Neural Information Processing Systems 33, 2020.

[7] github copilot: https://copilot.github.com/, accedido por última vez el 5 de Julio de 2021.

[8] H. Chen, O. Engkvist, Y. Wang, M. Olivercrona, T. Blaschke, "The rise of Deep learning in drug Discovery", Drug Discovery Today, Volume23, Issue 6, Junio 2018.

[9] N.C. Thompson, K. Greenewald, K. Lee, G.F. Manso, "The Computational Limits of Deep Learning", arXiv:2007.05558v1.

[10] D. H. Ackley, G.E. Hinton, T. J. Sejnowski, "A learning algorithm for Boltzmann machines", Neurocomputing: foundations of research, enero 1988.

[11] Base de datos ImageNet: https://www.image-net.org/, accedido por última vez el 5 de julio de 2021.

[12] N.P. Jouppi et. al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", Annual International Symposium on Computer Architecture, Junio 2017.

[13] S. Mittal, P. Rajput, S. Subramoney, "A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations", IEEE Transactions on Neural Networks and Learning Systems, Abril 2021.

[14] Intel Deep Learning Boost: https://www.intel.ai/intel-deep-learning-boost/, accedido por última vez el 5 de julio de 2021.

[15] I. Aldridge, "High-Frequency Trading, A Practica Guide to Algorithmic Strategies and Trading Systems", Publicado por John Wiley & Sons, 2013.

[16] S.L. Graham, P.B. Kessler, M.K. McKusick, “gprof: a call graph execution profiler”, ACM SIGPLAN Notices, Volume 39, Issue 4, Abril 2009.

[17] M. Berndl, L. Hendren, “Dynamic profiling and trace cache generation”, International Symposium on Code generation and Optimization, Marzo 2003.

[18] J. Seward, N. Nethercote, J. Weidendorfer, “Valgrind 3.3, Advanced Debugging and Profiling for GNU/Linux applications”, Publicado por Network Theory Ltd. Marzo 2008.

[19] Intel VTune Release Notes and New Features. https://software.intel.com .

[20] NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems

[21] D. Terpstra, H. Jagode, H. You, J. Dongarra, “Collecting Performance Data with PAPI-C”, Tools for High Performance Computing, 2009.

[22] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, K. Smith, “Cython: the best of both worlds”, Computing in Science & Engineering, 2011.

[23] TensorFlow Model Garden. https://github.com/tensorflow/models.

[24] MNIST database. http://yann.lecun.com/exdb/mnist/

[25] A. C. de Melo, “The New Linux ‘perf’ Tools,” 17 International Linux System Technology Conference. Nuremberg, 2010, [Online]. Available: http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf

[26] A. Yasin, “A Top-Down method for performance analysis and counters architecture,” IEEE International Symposium on Performance Analysis of Systems and Software, 2014