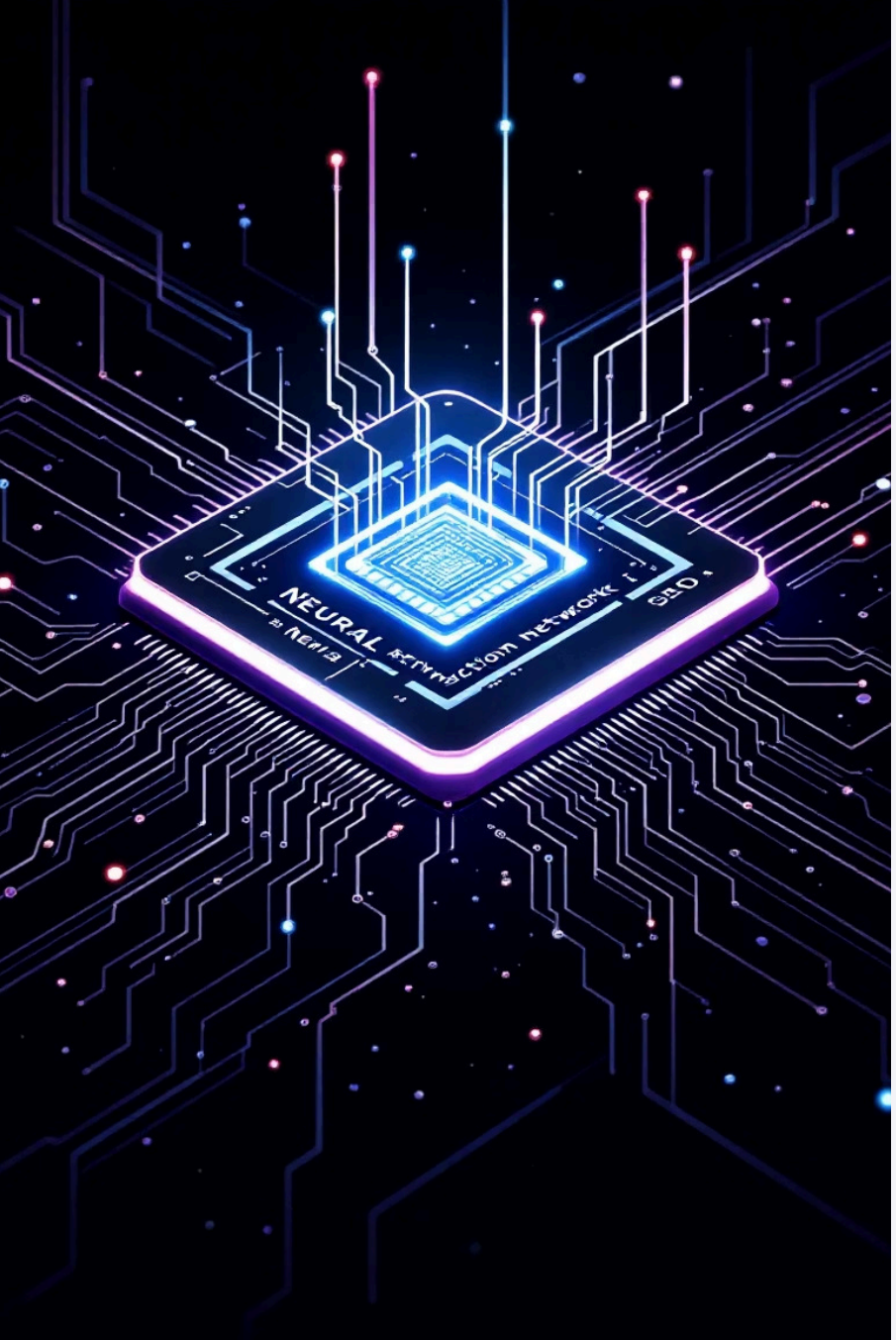


Análisis de Rendimiento y Paralelización de un Perceptrón Multicapa: Comparativa CPU vs. GPU

Presentado por: Juan Pablo C - Nicolás Jiménez O - Diciembre 2025

Introducción: Optimizando el Entrenamiento de Redes Neuronales

Implementación de MLP

Desarrollo de un Perceptrón Multicapa desde cero para la clasificación del dataset MNIST.

Evaluación del Rendimiento

Análisis del impacto de paradigmas de programación concurrente y paralela en el tiempo de entrenamiento.

Comparativa de Hardware

Exploración de la eficiencia computacional entre arquitecturas CPU multicore y GPU.

Este proyecto busca profundizar en cómo diferentes arquitecturas de hardware y modelos de programación (OpenMP, Multiprocessing, CUDA) afectan la eficiencia en tareas intensivas de álgebra lineal, cruciales en el ámbito del Deep Learning.

Arquitectura de la Red: Perceptrón Multicapa (MLP)

1

Capa de Entrada: 784 Neuronas

Imágenes de 28x28 píxeles aplanadas, sirviendo como la primera capa de nuestro modelo.

2

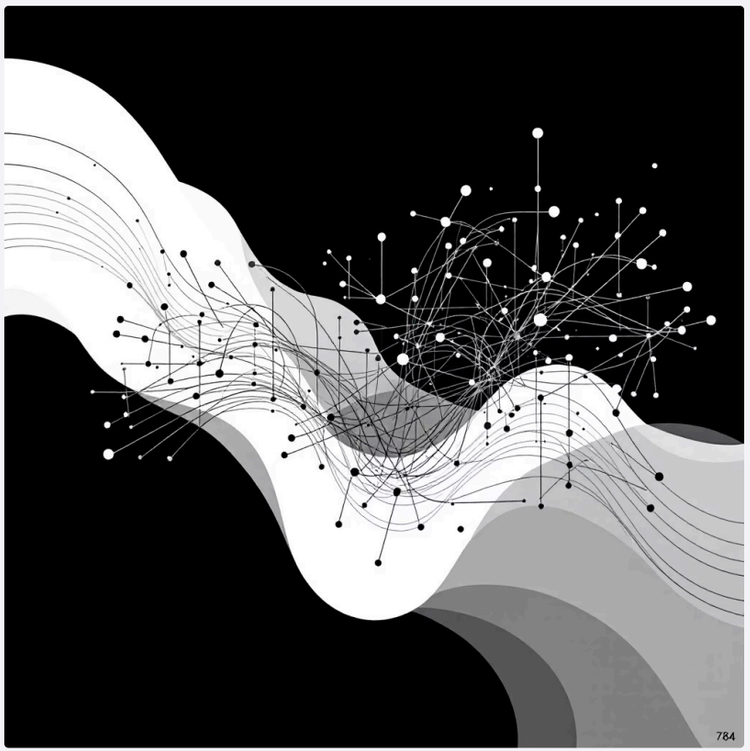
Capa Oculta: 512 Neuronas

Utilizando la función de activación ReLU para introducir no-linealidad en el modelo.

3

Capa de Salida: 10 Neuronas

Con activación Softmax, representando los dígitos del 0 al 9 para la clasificación.



La elección de 512 neuronas ocultas representa un equilibrio óptimo, evitando el "underfitting" con pocas neuronas y el costo computacional excesivo con demasiadas, sin mejoras sustanciales en precisión para MNIST.

Metodología de Pruebas: Entorno y Configuraciones

Hardware Utilizado

- **CPU:** Procesador con 8 hilos lógicos.
- **GPU:** NVIDIA Tesla T4 (Google Colab) con 2560 núcleos CUDA.

Dataset

- **MNIST:** 60,000 imágenes de entrenamiento para clasificación de dígitos.

Métrica de Evaluación

- **Tiempo de "Wall-clock":** Medición del tiempo total para completar ciclos de entrenamiento (Epochs).

Implementaciones Desarrolladas

01

Python Baseline (NumPy)

Optimizado con librerías BLAS para eficiencia.

02

C Secuencial

Implementación nativa sin optimización, sirviendo como base comparativa.

03

Python Multiprocessing

Paralelismo de datos mediante división de lotes (Batch splitting).

04

C + OpenMP

Paralelismo de hilos con técnicas de desenrollado y división de bucles.

05

CUDA C++

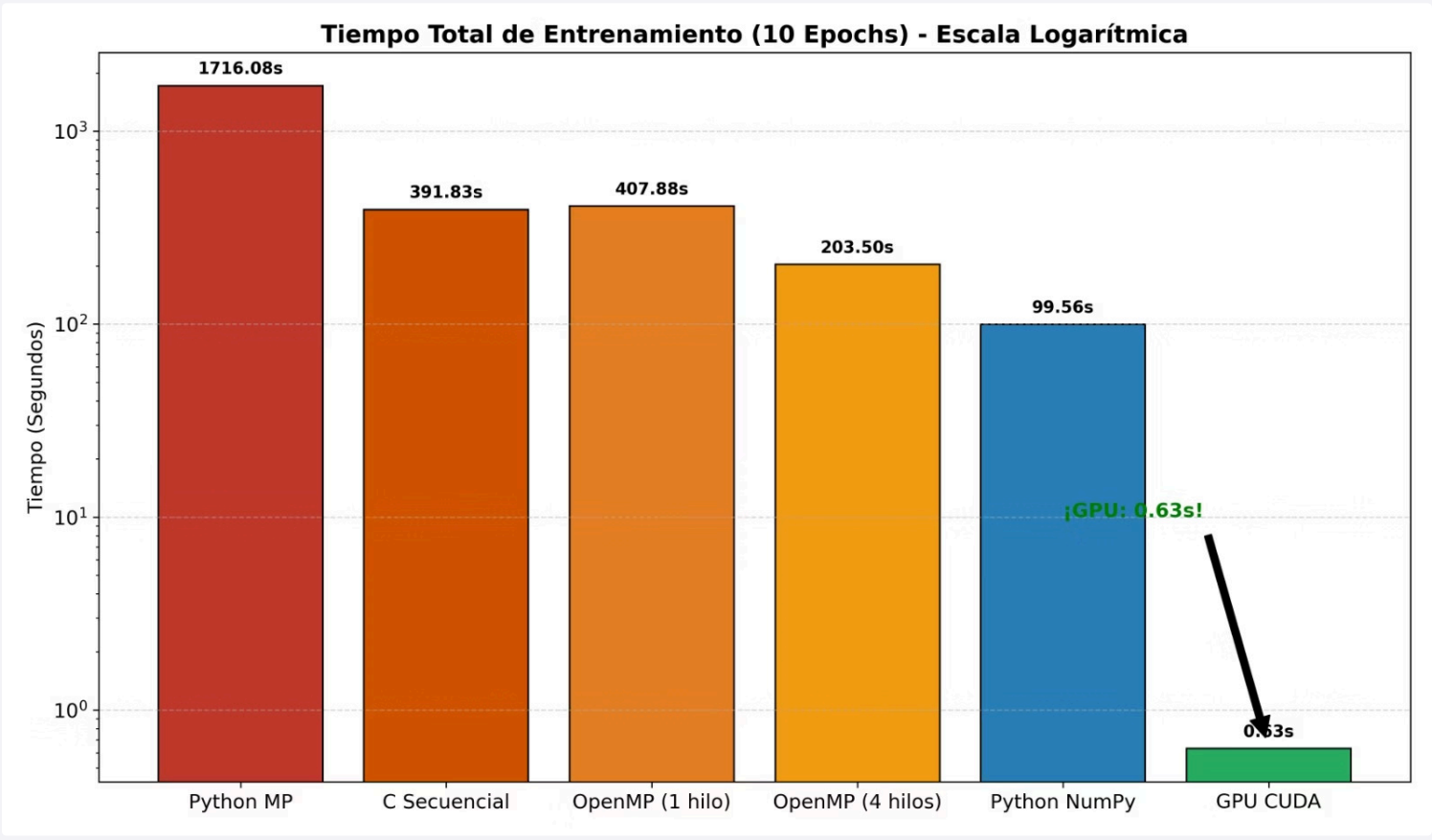
Paralelismo masivo en GPU para aceleración extrema.

Análisis de Rendimiento: Comparativa General de Tiempos

C Secuencial	391.83	1.0x (Base)	Algoritmo base sin optimizar.
Python (NumPy)	99.56	3.9x	Uso eficiente de librerías de bajo nivel.
C + OpenMP (4 hilos)	203.50	1.92x	Mejor configuración en CPU.
Python Multiprocessing	> 1700	< 0.2x	Degradación por overhead.
GPU CUDA	0.63	622.0x	Aceleración masiva.

La tabla muestra la vasta diferencia de rendimiento entre las distintas implementaciones, destacando la eficiencia de CUDA en GPU.

Fig 1. Comparativa logarítmica de tiempos de ejecución.



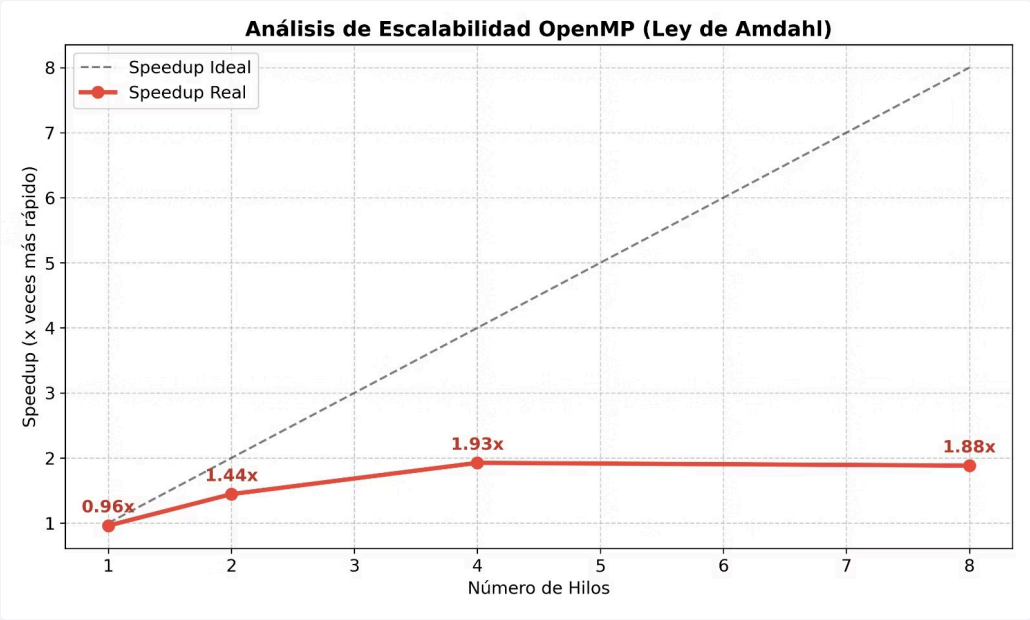
Ley de Amdahl: Límites de la Paralelización en CPU

Escalabilidad de OpenMP en C

- **1 Hilo:** 407.88 s
- **2 Hilos:** 271.21 s
- **4 Hilos:** 203.50 s
- **8 Hilos:** 208.14 s

Observamos que el "speedup" no es lineal; de hecho, al aumentar de 4 a 8 hilos, el rendimiento se degrada ligeramente.

> La Ley de Amdahl nos recuerda que la porción secuencial de un programa limita la máxima aceleración que se puede lograr con la paralelización.



Discusión: La gestión de memoria y la actualización de pesos introducen una porción secuencial ineludible. Al tratarse de una operación "Memory Bound", 8 hilos compiten por el ancho de banda de la RAM, creando un cuello de botella que anula la potencia de cálculo adicional.

Python Multiprocessing: Un Caso de "Slowdown"

Costo de la Serialización (Pickling)

La implementación con `multiprocessing` fue la menos eficiente, superando los 1700 segundos.

- Python debe copiar el estado completo de la red y los datos a cada nuevo proceso.
- El tiempo de "transporte de datos" entre procesos excede con creces el tiempo de cómputo.
- Esto resulta en un **rendimiento negativo** ("slowdown") en lugar de una aceleración.

El alto "overhead" de comunicación hace que la paralelización manual en Python sea contraproducente para tareas intensivas en datos y cómputo.

Análisis CUDA (GPU): Eficiencia y Escalabilidad

Perfilado del Tiempo en GPU

1 Transferencia CPU ↠ GPU

15.5% del tiempo total, el costo de mover datos a la GPU.

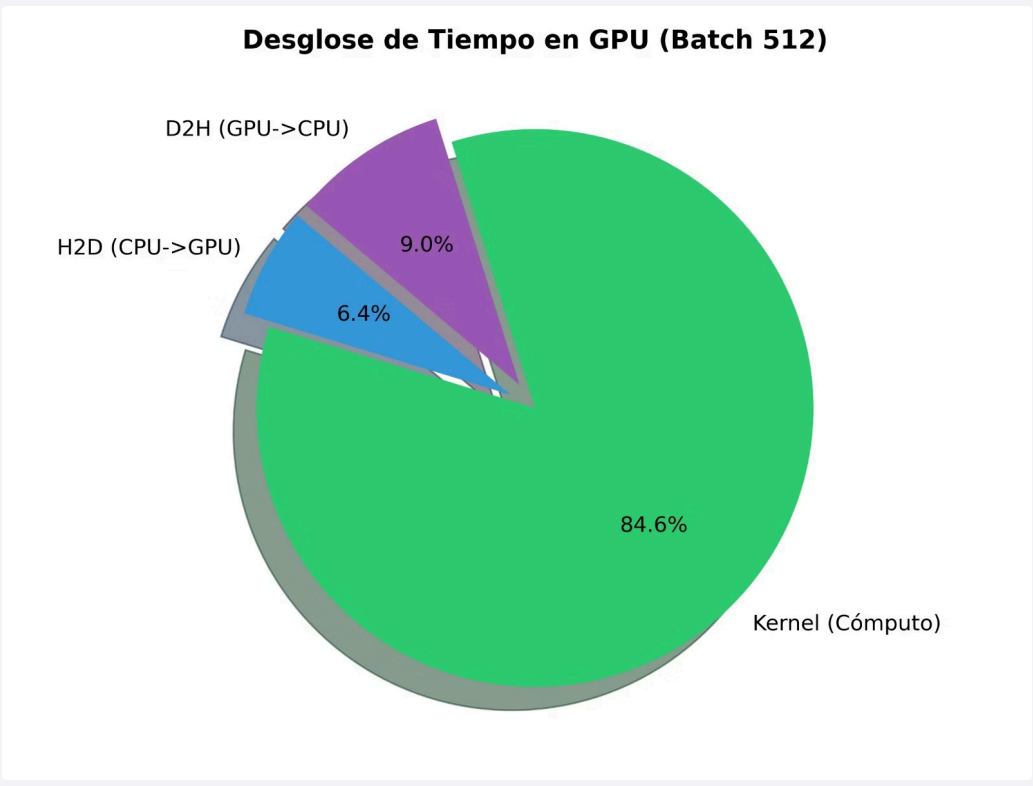
2 Ejecución del Kernel

84.5% del tiempo, indicando una alta eficiencia computacional.

3 Transferencia GPU ↠ CPU

Despreciable en escenarios de solo "forward-only".

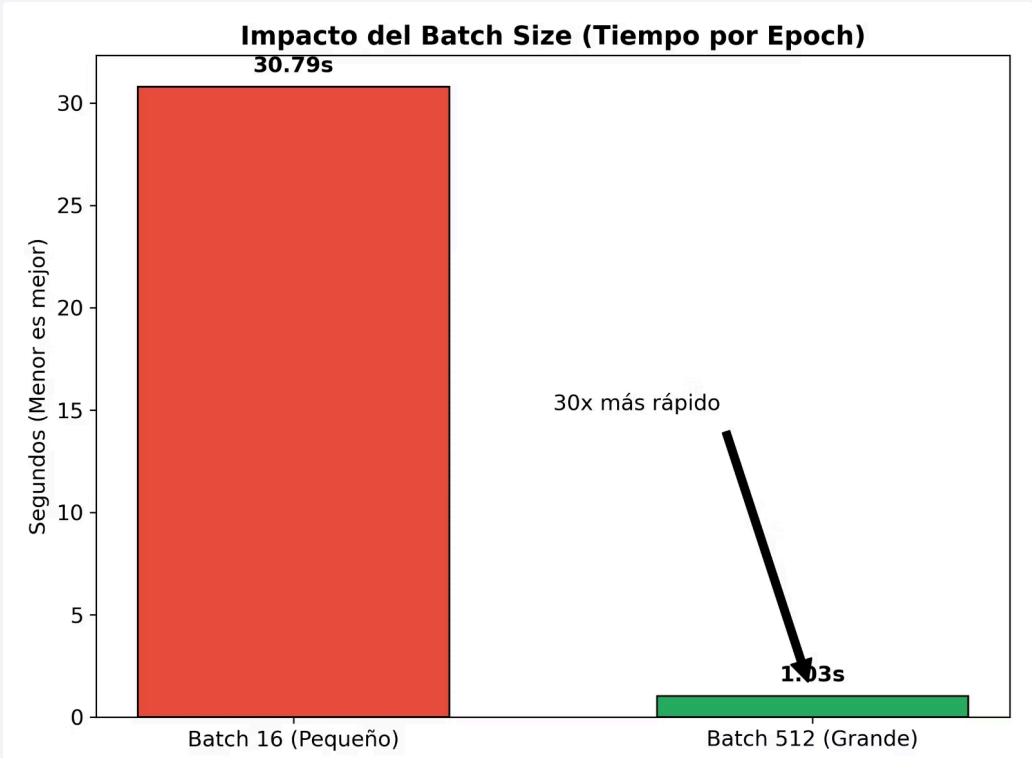
Fig 2. Desglose del tiempo en GPU para un Batch Size de 512.



Impacto del Batch Size

- **Batch 16:** ~30.79 s/epoch. Múltiples transferencias y lanzamientos de kernel dominados por la latencia.
- **Batch 512:** ~1.03 s/epoch. La latencia se amortiza, saturando los núcleos de la GPU y logrando un "throughput" óptimo.

La diferencia de 30x en rendimiento por el "batch size" subraya la importancia de la gestión de datos en la GPU.



Conclusiones Clave: Lecciones de Paralelización

→ El Dominio de la GPU en Deep Learning

Las GPUs son indiscutiblemente superiores, ofreciendo un "speedup" de hasta 3 órdenes de magnitud (600x) frente a las CPUs para cargas de trabajo intensivas en álgebra lineal.

→ Límites de la CPU y la Memoria

La paralelización en CPU (OpenMP) es útil pero limitada por el ancho de banda de memoria. Añadir más hilos no garantiza un rendimiento mejorado y puede incluso degradarlo.

→ Los Costos Ocultos de la Paralelización Ingenua

En Python, intentar paralelizar manualmente con `multiprocessing` puede ser contraproducente debido al alto "overhead" de comunicación entre procesos, llevando a un "slowdown" significativo.

→ Optimización del Batch Size en GPU

La elección adecuada del "batch size" es crucial para amortizar la latencia de transferencia de datos y saturar los núcleos de la GPU, maximizando el "throughput".

¡Gracias!