Análisis de Rendimiento y Paralelización de una red neuronal multicapa: Comparativa CPU vs GPU

AUTORES: Juan Pablo Cardona C. - Nicolás Jiménez O.

FECHA: 11 Diciembre 2025

1. Introducción

El objetivo principal de este proyecto es implementar una red neuronal tipo Perceptrón Multicapa (MLP) desde cero para la clasificación del dataset MNIST, y evaluar el impacto de diferentes paradigmas de programación concurrente y paralela en el tiempo de entrenamiento. Se busca comprender cómo la arquitectura del hardware (CPU multicore vs GPU) y el modelo de programación (OpenMP, Multiprocessing, CUDA) afectan la eficiencia computacional en tareas de álgebra lineal intensiva.

2. Arquitectura de la Red

Se diseñó un MLP con la siguiente topología:

- **Capa de Entrada:** 784 neuronas (imágenes de 28x28 píxeles "aplanadas").
- **Capa Oculta:** 512 neuronas con activación ReLU.
- **Capa de Salida:** 10 neuronas con activación Softmax (dígitos 0-9).

Justificación: Se seleccionaron 512 neuronas ocultas como un punto de equilibrio. Una cantidad menor (ej. 64) resultaba en *underfitting* (baja capacidad de aprendizaje), mientras que una cantidad mayor (ej. 2048) incrementaba cuadráticamente el costo computacional de la multiplicación de matrices (784×2048) sin ofrecer una mejora significativa en la precisión para este dataset específico.

3. Metodología de Pruebas

Las pruebas se realizaron en el siguiente entorno de hardware:

- **CPU:** Procesador con 8 hilos lógicos disponibles.
- **GPU:** NVIDIA Tesla T4 (Entorno Google Colab) con 2560 núcleos CUDA.
- **Datos:** Dataset MNIST (60,000 imágenes de entrenamiento).
- **Métrica:** Tiempo de "Wall-clock" para completar ciclos de entrenamiento (Epochs).

Se desarrollaron 5 implementaciones:

1. **Python Baseline:** NumPy (Optimizado con BLAS).
2. **C Secuencial:** Implementación nativa.
3. **Python Multiprocessing:** Paralelismo de datos (Batch splitting).
4. **C + OpenMP:** Paralelismo de hilos (Loop unrolling/splitting).
5. **CUDA C++:** Paralelismo masivo en GPU.

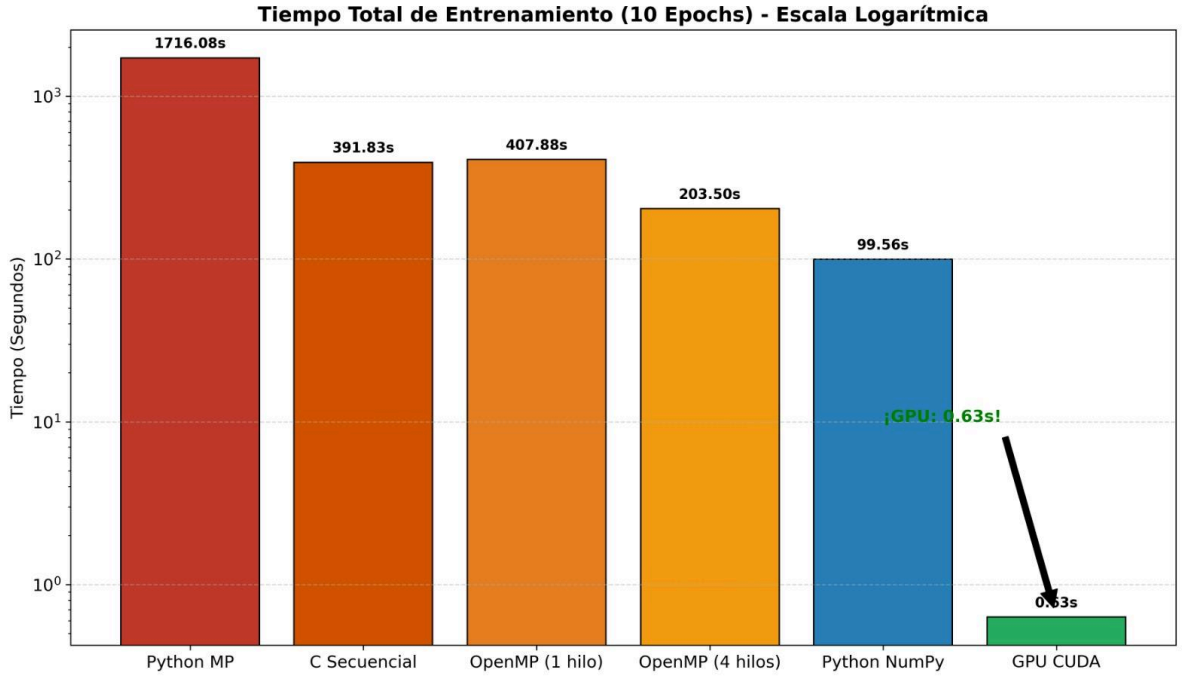
4. Análisis de Rendimiento

4.1. Tabla Comparativa General

A continuación se presentan los tiempos totales de entrenamiento.

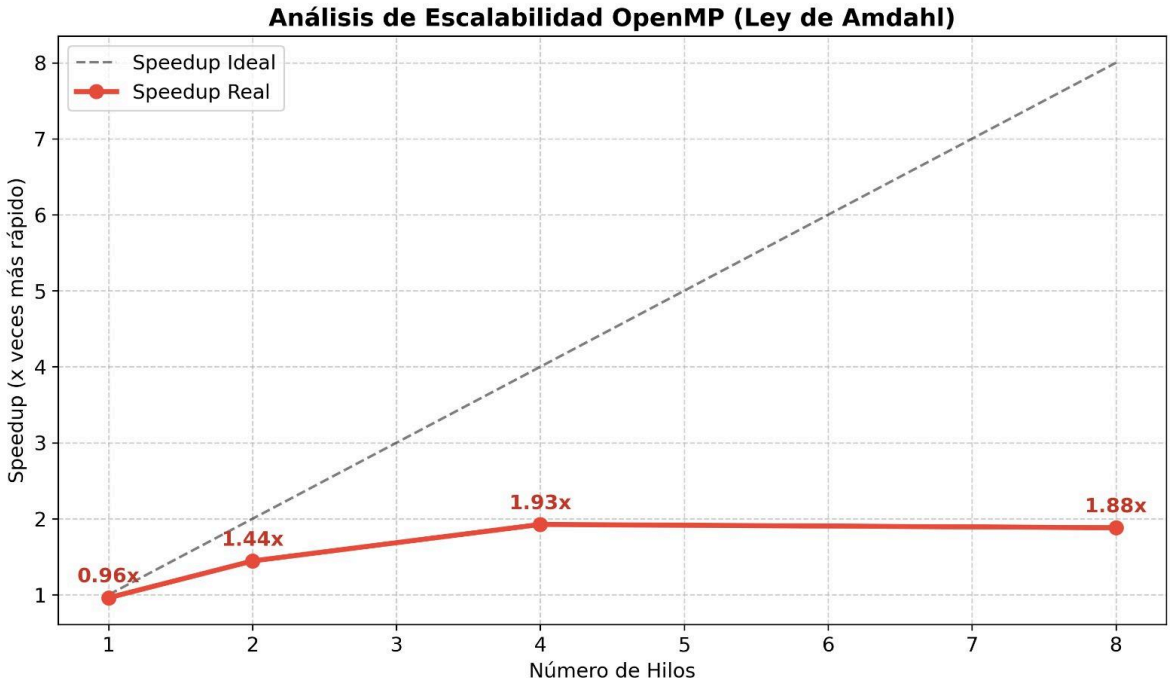
Implementación	Tiempo Total (s)	Speedup vs C Seq	Observaciones
C Secuencial	391.83 s	1.0x (Base)	Algoritmo base sin optimizar.
Python (NumPy)	99.56 s	3.9x	Uso eficiente de librerías de bajo nivel.
C + OpenMP (4 hilos)	203.50 s	1.92x	Mejor configuración en CPU.
Python Multiprocessing	> 1700 s (*)	< 0.2x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

Proyección a 10 epochs basada en ejecución parcial. (5 epochs)*



Se evaluó la escalabilidad en C utilizando OpenMP con 1, 2, 4 y 8 hilos.

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




```
juanp@Asus-JP MINGW64 /d/Juan Pablo/Documents/Uni
l_Project/src/4_parallel_c_openmp (main)
$ export OMP_NUM_THREADS=1 && ./mlp_omp
--- Escenario 2b: Paralelo OpenMP ---
>>> Usando 1 hilos de CPU <<<
Iniciando entrenamiento Paralelo...
Epoch 1 terminada.
Epoch 2 terminada.
Epoch 3 terminada.
Epoch 4 terminada.
Epoch 5 terminada.
Epoch 6 terminada.
Epoch 7 terminada.
Epoch 8 terminada.
Epoch 9 terminada.
Epoch 10 terminada.

>>> Tiempo Total OpenMP: 407.88 segundos <<<
```

```
juanp@Asus-JP MINGW64 /d/Juan Pablo/Documents/Universidad
l_Project/src/4_parallel_c_openmp (main)
$ export OMP_NUM_THREADS=2 && ./mlp_omp
--- Escenario 2b: Paralelo OpenMP ---
>>> Usando 2 hilos de CPU <<<
Iniciando entrenamiento Paralelo...
Epoch 1 terminada.
Epoch 2 terminada.
Epoch 3 terminada.
Epoch 4 terminada.
Epoch 5 terminada.
Epoch 6 terminada.
Epoch 7 terminada.
Epoch 8 terminada.
Epoch 9 terminada.
Epoch 10 terminada.

>>> Tiempo Total OpenMP: 271.21 segundos <<<
```

```
juanp@Asus-JP MINGW64 /d/Juan Pablo/Documents/Universi
l_Project/src/4_parallel_c_openmp (main)
$ export OMP_NUM_THREADS=4 && ./mlp_omp
--- Escenario 2b: Paralelo OpenMP ---
>>> Usando 4 hilos de CPU <<<
Iniciando entrenamiento Paralelo...
Epoch 1 terminada.
Epoch 2 terminada.
Epoch 3 terminada.
Epoch 4 terminada.
Epoch 5 terminada.
Epoch 6 terminada.
Epoch 7 terminada.
Epoch 8 terminada.
Epoch 9 terminada.
Epoch 10 terminada.

>>> Tiempo Total OpenMP: 203.50 segundos <<<
```

```
juanp@Asus-JP MINGW64 /d/Juan Pablo/Documents/Univers
l_Project/src/4_parallel_c_openmp (main)
$ export OMP_NUM_THREADS=8 && ./mlp_omp
--- Escenario 2b: Paralelo OpenMP ---
>>> Usando 8 hilos de CPU <<<
Iniciando entrenamiento Paralelo...
Epoch 1 terminada.
Epoch 2 terminada.
Epoch 3 terminada.
Epoch 4 terminada.
Epoch 5 terminada.
Epoch 6 terminada.
Epoch 7 terminada.
Epoch 8 terminada.
Epoch 9 terminada.
Epoch 10 terminada.

>>> Tiempo Total OpenMP: 208.14 segundos <<<
```

Observamos que el speedup no es lineal. Al pasar de 4 a 8 hilos, el rendimiento incluso decae levemente (de 203s a 208s). Esto valida la **Ley de Amdahl**:

1. **Porción Secuencial:** La gestión de memoria y la actualización de pesos limitan la mejora máxima.
2. **Saturación de Memoria:** Al ser 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 extra.

4.3. Análisis Python Multiprocessing

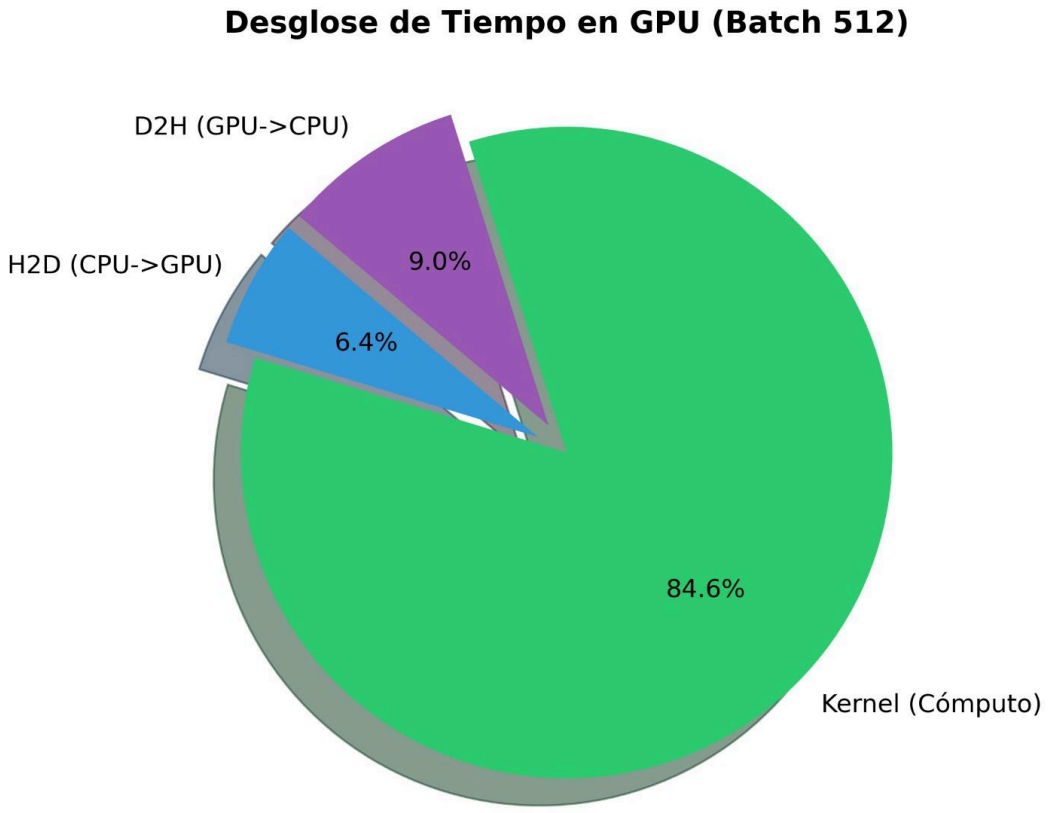
La implementación con multiprocessing resultó ser la menos eficiente (>1700s). Esto se debe al costo de **Serialización (Pickling)**. Python debe copiar el estado completo de la red y los datos a cada nuevo proceso. Dado que la operación matemática en sí es rápida (gracias a NumPy), el tiempo de "transporte de datos" entre procesos supera ampliamente al tiempo de cómputo, resultando en un rendimiento negativo (*slowdown*).

4.4. Análisis CUDA (GPU)

Perfilado (Profiling):

Para un Batch Size de 512, el desglose del tiempo en la GPU es el siguiente:

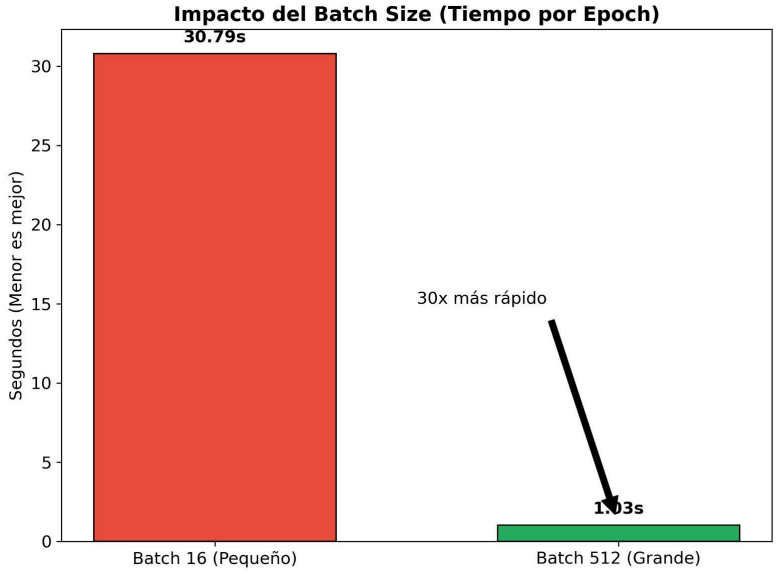
- **Transferencia CPU $\rightarrow$ GPU:** 15.5%
- **Ejecución del Kernel:** 84.5%
- **Transferencia GPU $\rightarrow$ CPU:** (Despreciable en forward-only)



Esto indica una alta eficiencia, donde la GPU pasa la mayor parte del tiempo calculando.

Impacto del Batch Size:

- **Batch 16:** ~30.79 s/epoch.
- **Batch 512:** ~1.03 s/epoch.



Explicación: Con un batch pequeño (16), se realizan miles de transferencias PCIe y lanzamientos de kernel, donde la **latencia** domina. Con un batch grande (512), se amortiza la latencia y se logra saturar los núcleos de la GPU (**Throughput**), logrando una mejora de 30x.

5. Conclusiones

1. **Hardware Adecuado:** Para Deep Learning, la GPU es indiscutiblemente superior, logrando un speedup de 3 órdenes de magnitud (600x) frente a la CPU.
2. **Límites de la CPU:** La paralelización en CPU (OpenMP) es útil pero limitada por el ancho de banda de memoria; añadir más hilos no siempre mejora el rendimiento.
3. **Costos Ocultos:** En Python, intentar paralelizar manualmente (Multiprocessing) puede ser contraproducente debido al alto *overhead* de comunicación entre procesos.