PROYECTO FINAL: PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA
IMPLEMENTACIÓN Y PARALELIZACIÓN DE UNA RED NEURONAL (MLP)

JAIME ANDRÉS CARDONA DÍAZ
DANER ALEJANDRO SALAZAR COLORADO
ESTUDIANTES

REINEL TABARES SOTO
DOCENTE

UNIVERSIDAD DE CALDAS
PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA
INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
2025

Contenido

1. INTRODUCCIÓN

Objetivos del proyecto:

a) Implementar un algoritmo de Deep Learning (MLP) desde cero, entendiendo su base matemática.
b) Identificar los cuellos de botella computacionales (pista: multiplicación de matrices).
c) Aplicar técnicas de paralelismo en CPU con memoria compartida (OpenMP) y memoria distribuida (multiprocessing).
d) Aplicar técnicas de paralelismo masivo en GPU (CUDA o PyCUDA).
e) Medir y analizar métricas de rendimiento como el Speedup, el Overhead y la Ley de Amdahl.
f) Entender la problemática de la transferencia de datos (Host-Device) en cómputo GPGPU.

2. ARQUITECTURA

Describir la arquitectura elegida (¿Por qué escogieron N neuronas?).

Se utilizó la siguiente arquitectura para el proyecto:

- Capa de entrada: 784 neuronas fijas como requisito del proyecto.
- Capa oculta (N): se utilizaron 256 neuronas ya que con esta se consiguió la mayor eficiencia en cuanto a tiempo y accuracy.
- Capa de salida: 10 neuronas fijas como requisito del proyecto.

Inicialmente, la versión CPU del MLP utilizaba una capa oculta de 500 neuronas, mientras que la versión CUDA usaba 256.

Al unificar la arquitectura a 256 neuronas, el tiempo de ejecución en C, Python y OpenMP se redujo aproximadamente a la mitad.

Esto confirma que la arquitectura —y específicamente el tamaño de las capas— tiene un impacto directo sobre el costo computacional del entrenamiento, ya que determina el número total de multiplicaciones de matrices. Con la arquitectura unificada, las comparaciones de rendimiento entre métodos (secuencial, multihilo y GPU) se vuelven más justas y precisas.

3. METODOLOGÍA DE PRUEBAS: Descripción del hardware (CPU, # de núcleos, GPU) y cómo se midieron los tiempos.

3.1. Hardware utilizado para las pruebas:

Sistema de Pruebas:

- Computador: Laptop de alto rendimiento
- Sistema Operativo: Windows 11 con emulación de Windows 10 para compatibilidad
- Procesador: Intel Core i7 de 14ª generación (Raptor Lake Refresh)
- Cantidad de núcleos:
    - 20 núcleos totales
        - 8 P-cores (alto rendimiento) → cada uno con HyperThreading (16 hilos)
        - 12 E-cores (eficiencia) → sin HyperThreading (12 hilos)
    - 28 hilos lógicos en total
- Memoria RAM: 16 GB DDR5
- Almacenamiento: SSD NVMe PCIe 4.0

GPU de Aceleración:

- Modelo: NVIDIA GeForce RTX 5060 Laptop GPU
- Memoria: 8 GB GDDR6 (128-bit bus)
- Multiprocesadores (SMs): 26
- Núcleos CUDA: 3,328 (26 SMs × 128 núcleos/SM)
- Arquitectura: Ada Lovelace (última generación)
- Capacidad de cómputo: 12.0 (soporte completo CUDA 12.x)
- Memoria compartida por bloque: 48 KB
- Máximo hilos por bloque: 1,024

- Warp size: 32 hilos
- Memoria constante: 64 KB
- Registros por bloque: 65,536

Software y Configuración:

- Sistema Operativo Base: Windows 11 Pro
- Entorno de desarrollo: Python 3.10.11 (pycuda_env)
- Framework CUDA: PyCUDA 2023.1+
- CUDA Toolkit: 12.x (compatible con arquitectura Ada Lovelace)
- Driver NVIDIA: Game Ready Driver 551.xx o superior
- Librerías científicas: NumPy 1.24+, Matplotlib 3.7+, pandas 2.0+
- IDE: Microsoft Visual Studio 2022 Professional

3.2. Métodos de Medición de Tiempos:

- C Secuencial:

  Función de Medición: clock_t (de <time.h>)

  #include <time.h>

  clock_t start_total = clock();
  // ... código a medir ...
  clock_t end_total = clock();
  double time_spent = (double)(end_total - start_total) / CLOCKS_PER_SEC;

  Características:

  - Usa clock() que mide tiempo de CPU consumido
  - Convierte a segundos dividiendo entre CLOCKS_PER_SEC
  - Se mide tanto el tiempo total de entrenamiento como el tiempo por época
  - Precisión: Depende del sistema, típicamente milisegundos

Limitaciones:

- clock() mide tiempo de CPU, no tiempo de reloj pared (wall-clock time)
- En sistemas multiprocesador, puede contar múltiples núcleos

- Python secuencial:

  Función de Medición: time.time()

  ```
  import time

  start_time = time.time()
  # ... código a medir ...
  end_time = time.time()
  total_time = end_time - start_time
  ```

  Características:

  - Usa time.time() que devuelve segundos desde epoch (1 de enero de 1970)
  - Mide tiempo de reloj pared (wall-clock time)
  - Se mide el tiempo total de entrenamiento (10 épocas)
  - Precisión: Depende del sistema operativo, típicamente milisegundos

  Ventajas:
  - Es una función de alto nivel que es portable
  - Mide el tiempo real transcurrido

- Python Paralelo con Multiprocessing:

  Función de Medición: time.time()

  ```
  import time

  start_time = time.time()
  ```

```
# ... código MLP paralelo ...
end_time = time.time()
total_time = end_time - start_time
```

Características:

- Usa time.time() igual que la versión secuencial de Python
- Mide tiempo de reloj pared incluyendo:
  - Creación de procesos workers
  - Comunicación entre procesos
  - Computación paralela
  - Sincronización
- Número de Workers: 4 (configurable en N_WORKERS)
- Se mide el tiempo total de entrenamiento completo

- C Paralelo con OpenMP:

  Función de Medición: omp_get_wtime() (de <omp.h>)

```
#include <omp.h>

double start_total = omp_get_wtime();
// ... código paralelizado con OpenMP ...
double end_total = omp_get_wtime();
double time_spent = end_total - start_total;  // en segundos
```

  Características:

  - Usa omp_get_wtime() que mide tiempo de reloj pared de forma portátil
  - Esta función es específicamente diseñada para aplicaciones paralelas
  - Se mide tanto el tiempo total como el tiempo por época
  - Número de Threads OpenMP: Determinado por omp_get_max_threads() (variable de entorno OMP_NUM_THREADS)
  - Precisión: Alta (típicamente microsegundos)

  Ventajas sobre clock():

- omp_get_wtime() mide tiempo real transcurrido, no tiempo de CPU
- Es la función recomendada para medir performance de código OpenMP
- Permite comparaciones justas entre versiones secuencial y paralela

- PyCUDA (Paralelo):

  Funciones de Medición:

  - time.perf_counter() para tiempo total
  - pycuda.driver.Event para eventos GPU

```
import time
import pycuda.driver as drv

# Tiempo total de entrenamiento
epoch_start = time.perf_counter()
# ... código de entrenamiento ...
epoch_end = time.perf_counter()
epoch_time = epoch_end - epoch_start

# Tiempos específicos de GPU
start = drv.Event()
end = drv.Event()

start.record()
# ... kernel CUDA o transferencia de datos ...
end.record()
end.synchronize()
t_kernel = start.time_till(end)  # en milisegundos
```

  Características:

  - time.perf_counter(): Mide tiempo de reloj pared de alta resolución
  - pycuda.driver.Event: Mide tiempos específicos en GPU con sincronización
  - Se mide:
    - Tiempo total de cada época
    - Tiempos de transferencia Host-to-Device (CPU ↔ GPU)

- Tiempos de kernels CUDA (computación en GPU)
- Tiempos de transferencia Device-to-Host (GPU ↔ CPU)

- Precisión: Milisegundos para kernels, microsegundos para perf_counter

4. ANÁLISIS DE RENDIMIENTO

4.1. Tabla Comparativa: Una tabla con el tiempo total de entrenamiento (ej. 10 epochs) para todas las implementaciones.

Implementación	Tecnología	Paralelismo	Tiempo Total (s)	Accuracy (%)	Velocidad Relativa*
Python Secuencial	Python/NumPy	Ninguno	28.79	91.87%	1.0x (base)
C Secuencial	C (AVX)	Ninguno	35.92	91.37%	0.80x
Python Paralelo	Python Multiprocessing	CPU (4 workers)	71.30	91.86%	0.40x
OpenMP	C (OpenMP)	CPU (4 threads)	9.74	91.39%	2.95x
CUDA/GPU	**PyCUDA**	**GPU (Masivo)**	**8.67**	**81.32%**	**3.32x**

4.2. Gráfica de Speedup (C+OpenMP): Una gráfica que muestre el speedup (Tiempo Secuencial / Tiempo Paralelo) de la versión C+OpenMP usando 1, 2, 4, 8, ... N hilos (hasta el máximo de su CPU).

Número de hilos	1	2	4	8	16
Tiempo (s)	27.8810	27.6210	20.2120	5.0780	9.7720

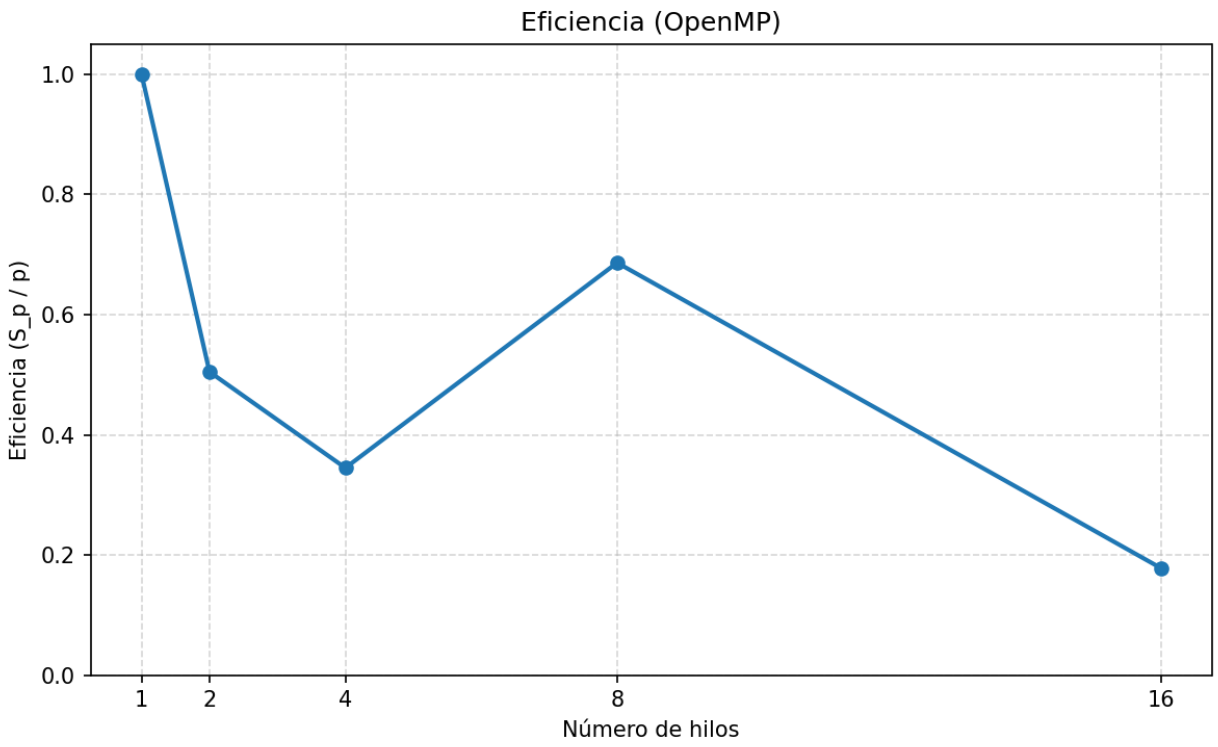

Figura 1. Eficiencia (Sp / p) de OpenMP en función del número de hilos

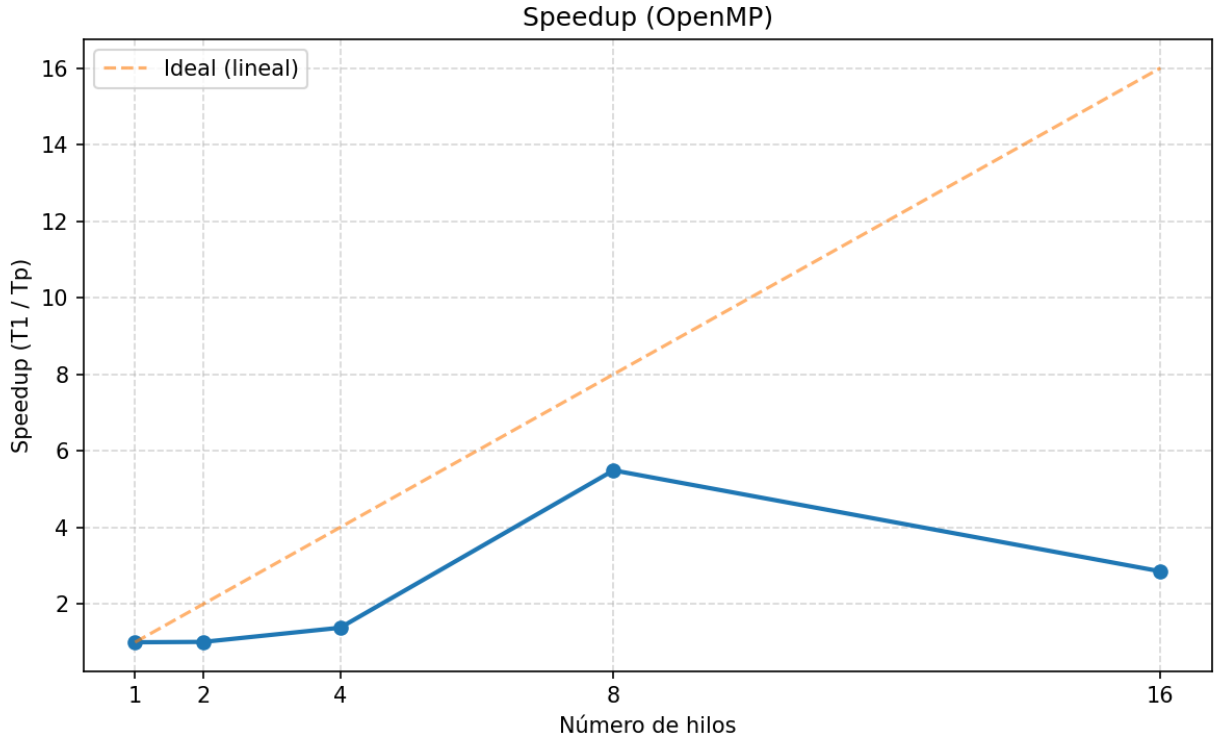

Figura 2. Speedup (T1 / Tp) de la versión C + OpenMP en función del número de hilos, comparado con el ideal lineal

4.3. Análisis Ley de Amdahl: Discutir la gráfica anterior. ¿Es el speedup lineal? ¿Por qué sí o por qué no? (Mencionar overhead y porción secuencial).

El speedup no es lineal, algunas causas podrían ser:

- Porción secuencial del código: inicialización, combinación de resultados, actualización de pesos.
- Overhead de OpenMP: crear y sincronizar threads tiene costo.
- Contención de memoria/cache: más hilos compitiendo por la misma memoria reduce la eficiencia.

4.4. Análisis Python multiprocessing: Comparar el speedup de multiprocessing vs. OpenMP. ¿Cúal es más eficiente y por qué? (Mencionar overhead de creación de procesos y serialización).

Resultados de Python multiprocessing:

Procesos	Tiempo (s)	Speedup
1	49.09	1.00
2	52.70	0.93
4	79.40	0.62
8	97.23	0.50
16	212.20	0.23

Multiprocessing no escala — incluso empeora conforme aumentan los procesos.

Resultados de OpenMP

Hilos	Tiempo (s)	Speedup
1	27.88	1.00
2	27.62	1.01
4	20.21	1.38
8	5.07	5.50
16	9.77	2.85

OpenMP sí escala, y obtiene speedup real y significativo.

¿Cuál es más eficiente y por qué?

OpenMP es mucho más eficiente que multiprocessing para este tipo de tarea porque:

a) Overhead de creación de procesos (muy alto en Python)

- Python multiprocessing crea procesos completos del sistema operativo, cada uno con:
    - Su propio intérprete de Python
    - Su propia memoria
    - Sus propios stacks

- Crear un proceso puede costar milisegundos o incluso decenas de ms.
- En el proyecto, se utilizaron 234 batches por época × 10 épocas × muchos procesos → miles de sincronizaciones, lo cual explota el overhead.

OpenMP, en cambio:

- crea hilos, no procesos.
- los hilos comparten memoria y son muchísimo más livianos.
- una vez creado el pool de hilos, no se destruyen por cada iteración.

b) Serialización y deserialización de datos (pickle)

En multiprocessing, cuando no hay shared memory:

- Cada proceso debe recibir datos serializados en formato pickle.
- Los tensores/arrays deben copiarse o mapearse entre procesos.
- Incluso con shared memory, las tareas deben serializar instrucciones y objetos Python.

La serialización es uno de los mayores cuellos de botella en multiprocessing.

En el caso del dataset MNIST es grande (~381 MB en memoria compartida), pero aun con shared memory:

- cada batch necesita comunicación de control
- sincronización pesada
- envío de índices, metadatos, gradientes, etc.

OpenMP no serializa nada, porque todos los hilos comparten el mismo espacio de memoria.

c) El GIL limita Python

Aunque se hace uso de procesos para evadir el GIL:

- El código Python sigue ejecutándose en CPython
- Mucho contexto depende del intérprete, lo cual agrega overhead por proceso

d) Comunicación y sincronización

Multiprocessing requiere:

- pipes
- locks
- colas
- semáforos

Todo esto pasa por el kernel, es lento.

OpenMP:

usa primitivas de sincronización a nivel de hardware

- optimizadas y ultra rápidas
- OpenMP trabaja directamente sobre C/C++, sin GIL.

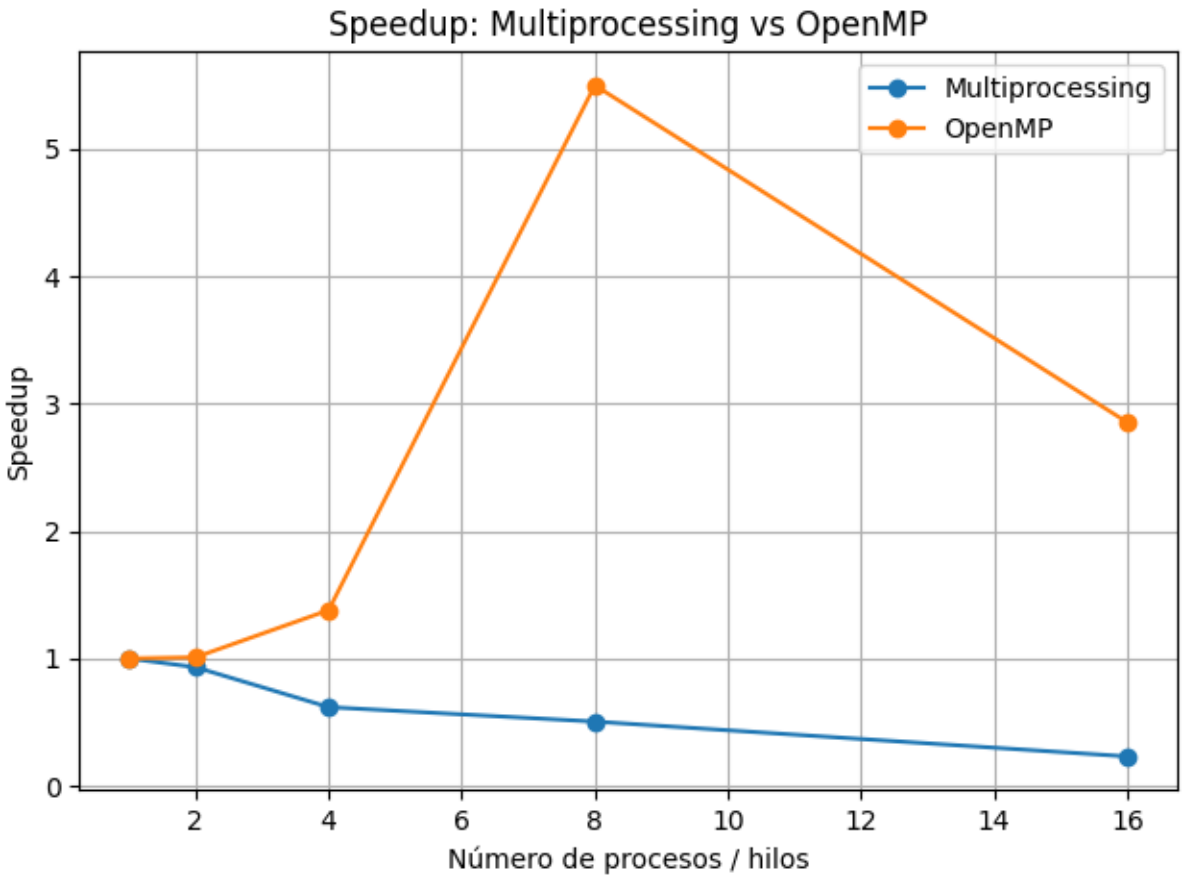

Figura 3. Speedup comparativo: Multiprocessing (procesos) vs. OpenMP (hilos)

4.5. Análisis CUDA/PyCUDA:

a) Presentar una gráfica de profiling que muestre el tiempo desglosado en: (1) Transferencia CPU→GPU, (2) Ejecución del Kernel, (3) Transferencia GPU→CPU.

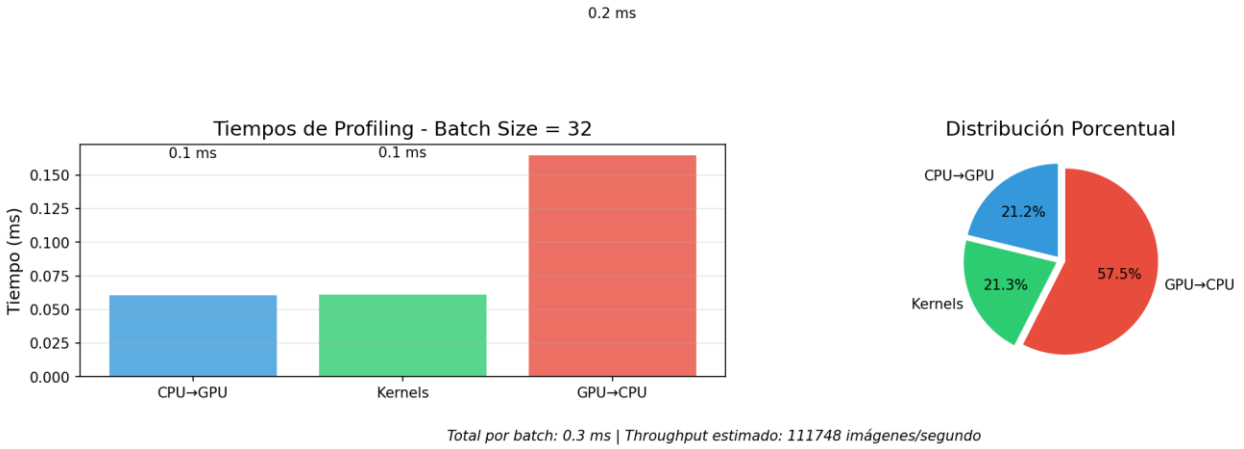

Figura 4. Tiempos de Profiling y Distribución Porcentual del tiempo por batch (CPU ↔ GPU y Kernels) para Batch Size = 32

Profiling detallado para batch size = 32

- Transferencias CPU→GPU: 21.2%
- Ejecución de kernels: 21.3%
- Transferencias GPU→CPU: 57.5%
- Throughput: 111,747 imágenes/segundo

b) Análisis de batch size: Ejecutar la versión de CUDA con un batch size pequeño (ej. 16) y uno grande (ej. 512). Explicar por qué el rendimiento cambia tan drásticamente.

PROFILING Detallado - Batch Size = 16 vs PROFILING Detallado - Batch Size = 512

Concepto	Batch Size 16	Batch Size 512
Transferencias CPU → GPU	0.1 ms (21.0%)	0.5 ms (46.0%)
Ejecución de kernels	0.1 ms (21.7%)	0.1 ms (11.9%)
Transferencias GPU → CPU	0.2 ms (57.3%)	0.4 ms (42.0%)
Total por batch	0.3 ms	1.0 ms
Throughput estimado	60,307 imágenes/s	498,532 imágenes/s
Tiempo por epoch	8.25 s	0.68 s
Accuracy test final	0.8023	0.3388
Tiempo total	82.51 s	6.85 s

¿Por qué cambia tan drásticamente?

EFECTO EN RENDIMIENTO (THROUGHPUT):

Batch = 16 → 60307 img/s

Batch = 512 → 498532 img/s (8x MÁS RÁPIDO)

RAZONES:

1. Overhead de transferencias: batch pequeño = más transferencias CPU↔GPU.
2. Paralelización: GPU funciona mejor con mucho trabajo (batch grande)
3. Kernel launch overhead: Lanzar kernels tiene costo fijo

EFECTO EN ACCURACY:

Batch = 16 → 80.23% (bueno)

Batch = 512 → 33.88% (pésimo)

RAZONES:

1. Batch grande = gradientes menos ruidosos, pero convergencia más lenta
2. Batch pequeño = actualizaciones más frecuentes (mejor optimización)
3. Batch = 512 puede quedar atrapado en mínimos locales

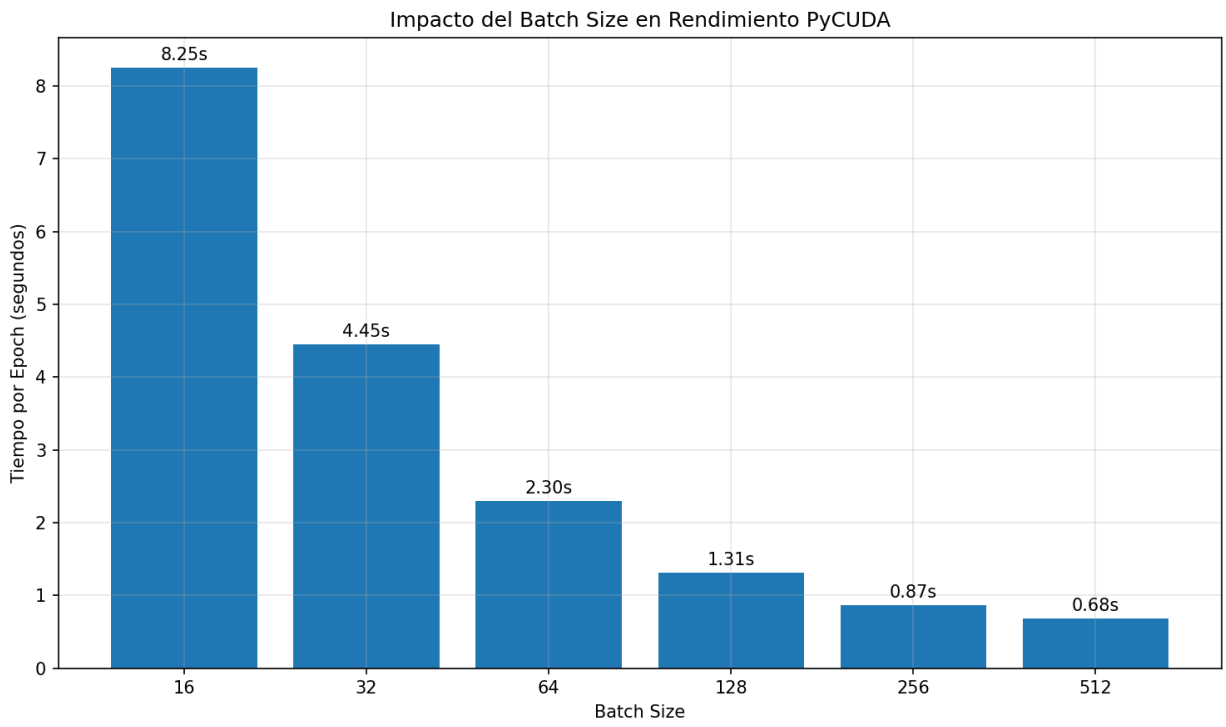

Figura 5. Impacto del Batch Size en el tiempo por época de rendimiento PyCUDA

5. CONCLUSIONES: Resumen de los hallazgos. ¿Cúando vale la pena usar cada paradigma?

Casos de uso diferentes paradigmas:

a) Python secuencial con numpy:

- Para tener una línea base de comparación
- Cuando se necesite validar que el MLP funcione
- Cuando el paralelismo tiene un overhead muy alto
- Cuando se cuentan con pocos nucleos en la maquina
- Cuando el trabajo es muy pequeño

b) C secuencial:

- Para tener la línea base más rápida sin paralelización
- Para medir speed up de OpenMp
- Para validar kernels paralelos
- Para entrenar datasets mediados

c) Python paralelo (multiprocessing):

- Cuando se quiere medir el overhead
- Para comparar distintos paradigmas
- Para entender el costo de dividir el batch + sincronizar gradientes

d) C paralelo (OpenMp):

- Cuando es una tarea altamente computacional (CPU-bound)
- Se tienen loops grandes e independientes
- Buscar el mejor rendimiento posible
- En un entorno HPC
- Cuando la latencia importa
- Cuando se requiere control fino del hardware

e) PyCuda Paralelo:

- Cuando la tarea es masivamente paralela
- Cuando se quiere usar GPU y mantener código Python
- Cuando se necesita control total de la GPU, a diferencia de TensorFlow o PyTorch

¿Qué aprendieron?

- Diseñar una red neuronal MLP desde 0, realizando las operaciones que componen el *forward propagation* y el *back propagation*.

- Aplicar el diseño en distintos métodos y lenguajes de programación, implementando en Python secuencial con NumPy, Python paralelo con *multiprocessing*, C secuencial, C paralelo con OpenMP, y finalmente con PyCuda.

- Entendimos la importancia de la arquitectura en la red neuronal: un diseño estilo “cuello de botella” con un número mayor de neuronas para la capa de entrada, un numero menor que las neuronas de entrada, pero mayor que las neuronas de salida para la capa oculta, y finalmente el numero menor de neuronas para la capa de salida. Una correcta elección de neuronas puede mejorar el desempeño

- La elección en el número del *batch size* tiene una gran importancia en el rendimiento de la red. Un numero alto de *batch* junto con un numero alto de hilos en OpenMP o el uso de la paralelización masiva en GPU, puede dar buenos tiempos, pero bajar el aprendizaje de la red. Es importante entender o seleccionar el número correcto de *batch size* y cantidad de hilos para mejorar el rendimiento. Por ejemplo, en PyCuda el mejor rendimiento fue con un *batch size* de 64.

- El uso de *multiprocessing* para la ejecución mostro la importancia de tomar en cuenta la comunicación entre procesos, entre más procesos se destinaban para la ejecución del entrenamiento, más costoso en tiempo se hacia el mismo, algo que parece una contradicción se vio claramente debido al excesivo *overhead* del *multiprocessing*.

- Los mejores resultados en la relación tiempo–accuracy se obtuvieron con OpenMP. En PyCUDA, el mejor desempeño se logró con un batch size de 64; sin embargo, al aumentar el batch size —por ejemplo, a 512— aunque el tiempo mejora ligeramente, la accuracy cae de forma significativa.

- En conjunto, el proyecto nos permitió integrar los conceptos vistos en clase y comparar en la práctica cómo diferentes enfoques de programación influyen en el desempeño de un mismo modelo.