Cómputo Paralelo y Distribuido

Proyecto II

Grupo: 8CC2

Semestre: Agosto - Diciembre 2025

Profesor: De Lira Miramontes José Saul

Integrantes:

Martín Eduardo Chacón Orduño - 351840

Allan Hall Solorio - 358909

Jorge Alejandro Beltran Rosales - 348635

Fecha de entrega: 3/11/2025

1. Calcular el producto de dos matrices de dimensiones 10,000 x 10,000, que requiere operaciones masivas en paralelo. Solución con Python y CUDA: Usar CuPy para mapear las operaciones matriciales directamente a kernels CUDA, reduciendo el tiempo de cómputo en GPU.

2. Implementar una función que encuentre el valor máximo en un array de números flotantes de gran tamaño (>100 millones de elementos) usando GPU.
a. Entrada: Array 1D de números reales (float32)
b. Salida: El valor máximo del array
c. Comparar rendimiento: CPU vs GPU
3. Calcular la suma de todos los elementos en un vector muy grande es un problema común que requiere una estrategia de reducción paralela para ser eficiente en una GPU. Implementa un kernel de reducción para sumar todos los elementos de un vector con millones de elementos.
a. Entrada: Vector 1D con números reales (float32)
b. Salida: Número (float32) que representa la suma de los elementos del vector
4. Implementar el algoritmo Map-Reduce utilizando Python CUDA

Problema 1#

CUDA es una tecnología desarrollada por NVIDIA que permite ejecutar programas de manera paralela utilizando la tarjeta gráfica (GPU). En este modelo, la CPU funciona como el controlador principal (host), mientras que la GPU se encarga de realizar los cálculos masivos (device). En nuestro proyecto, primero generamos las matrices en la CPU, luego las copiamos a la memoria de la GPU para realizar la multiplicación, y finalmente devolvemos el resultado a la CPU para medir el tiempo.

La operación en la GPU se ejecuta mediante kernels, que son funciones paralelas. Cada kernel se ejecuta a través de muchos hilos (threads), los cuales realizan operaciones pequeñas y simultáneas. Estos hilos se agrupan en bloques (blocks), y todos los bloques forman una rejilla (grid). Esta estructura permite que miles de hilos

trabajen en diferentes partes de la matriz al mismo tiempo, calculando en paralelo los elementos del resultado.

En cuanto a la memoria, la GPU tiene una jerarquía: los registros son los más rápidos y están dentro de cada hilo, luego está la memoria compartida, que es usada entre los hilos del mismo bloque y permite reutilizar datos sin necesidad de ir a la memoria global, haciendo el cálculo más eficiente. Finalmente está la memoria global, que es más grande pero más lenta y se usa para almacenar las matrices completas en la GPU. El acceso eficiente a la memoria es fundamental para obtener un buen rendimiento.

Un aspecto importante es la transferencia de datos entre CPU y GPU. Antes de multiplicar las matrices, debemos enviarlas desde la memoria RAM hacia la memoria de la GPU. Esta transferencia tiene un costo, ya que se realiza a través del bus PCIe y puede ser más lenta que el cálculo en sí. Por esta razón, si se realizan muchas operaciones dentro de la GPU sin regresar los datos a la CPU, el beneficio de usar CUDA es mucho mayor.

Finalmente, para acelerar la multiplicación de matrices, se usa una técnica llamada tiling, donde la GPU divide las matrices en sub-bloques para aprovechar la memoria compartida y reducir el número de accesos a memoria global. En nuestro código con CuPy, no tenemos que programar esto manualmente, ya que la biblioteca llama internamente a cuBLAS, una librería altamente optimizada de NVIDIA que aplica todas estas estrategias automáticamente.

En resumen, CUDA acelera la multiplicación de matrices porque permite ejecutar miles de operaciones al mismo tiempo, utiliza una jerarquía de memoria optimizada para el rendimiento y aprovecha al máximo la arquitectura paralela de la GPU.

```
import time
import numpy as np
import cupy as cp


# Configuración del tamaño de la matriz
```

```
N = 10000

print(f"Multiplicando matrices de {N}x{N}...\n")

# Calculo en cPU
# Generar dos matrices con valores aleatorios en CPU
A_cpu = np.random.rand(N, N).astype(np.float32)
B_cpu = np.random.rand(N, N).astype(np.float32)

# Medir el tiempo antes y después de la multiplicación en CPU
start_cpu = time.time()
C_cpu = A_cpu @ B_cpu  # Operación matricial usando BLAS en CPU
end_cpu = time.time()
cpu_time = end_cpu - start_cpu

print(f"CPU terminada en: {cpu_time:.3f} segundos")



# Calculo en GPU
# Las matrices generadas en NumPy (host) se copian a la memoria de la
GPU (device)
A_gpu = cp.asarray(A_cpu)
B_gpu = cp.asarray(B_cpu)

# Asegurar que la GPU ya terminó las transferencias antes de medir el
tiempo
cp.cuda.Stream.null.synchronize()

# Medimr el tiempo de la multiplicación en GPU
start_gpu = time.time()
C_gpu = A_gpu @ B_gpu  # Multiplicación matricial ejecutada en CUDA
cp.cuda.Stream.null.synchronize()   # Esperar a que la GPU termine el
cálculo
end_gpu = time.time()

gpu_time = end_gpu - start_gpu
print(f"GPU terminada en: {gpu_time:.3f} segundos")

# Calcular cuánto más rápido es la GPU comparada con la CPU
speedup = cpu_time / gpu_time
print(f"\nAceleración GPU vs CPU: {speedup:.2f}x más rápido")

print("\nPrograma terminado")
```

Problema 2#

Implementar una función en Python que encuentre el **valor máximo de un arreglo de números flotantes muy grande** (>100 millones de elementos), comparando el rendimiento entre ejecución en **CPU (NumPy)** y **GPU (CuPy con CUDA)**.

Transferencia de datos CPU -> GPU

1. **Host → Device:** Los datos (arrays) se envían desde la RAM (CPU) a la memoria de la GPU.
2. **Host → Device:** Los datos (arrays) se envían desde la RAM (CPU) a la memoria de la GPU.
3. **Ejecución del Kernel:** La GPU ejecuta las operaciones en paralelo.

Estas transferencias pueden ser **costosas en tiempo**, por eso es más eficiente procesar grandes volúmenes de datos directamente en GPU.

Implementación en Google Colab

Versión CPU (numpy)

```
# CPU Versión secuencial (base de comparación)
import numpy as np
import time

# Crear array grande en CPU (float32)
N = 100_000_000
arr_cpu = np.random.rand(N).astype(np.float32)

# Medir tiempo CPU
t0 = time.perf_counter()
max_cpu = np.max(arr_cpu)
t1 = time.perf_counter()

print(f"Máximo (CPU): {max_cpu}")
print(f"Tiempo CPU: {t1 - t0:.4f} segundos")
```

- np.random.rand() genera el arreglo

- np.max() realiza una **reducción secuencial**, recorriendo el array elemento por elemento.
- Solo usa **un hilo** de CPU (proceso secuencial).

Versión GPU (CuPy)

```
import cupy as cp
import time

# Copiar datos a GPU
t0 = time.perf_counter()
arr_gpu = cp.array(arr_cpu)  # Transferencia host->device
cp.cuda.Stream.null.synchronize()
t1 = time.perf_counter()

# Calcular máximo en GPU
t2 = time.perf_counter()
max_gpu = cp.max(arr_gpu)
cp.cuda.Stream.null.synchronize()  # Esperar que termine
t3 = time.perf_counter()

# Transferir resultado a CPU
max_gpu_host = max_gpu.get()
t4 = time.perf_counter()

print(f"Máximo (GPU): {max_gpu_host}")
print(f"Tiempo transferencia Host→Device: {t1 - t0:.4f} s")
print(f"Tiempo kernel GPU: {t3 - t2:.4f} s")
print(f"Tiempo Device→Host: {t4 - t3:.4f} s")
print(f"Tiempo total GPU: {t4 - t0:.4f} s")
```

- cp.array() Transfiere los datos del CPU a la GPU.
- cp.max() Ejecuta un **kernel de reducción paralela**, donde miles de hilos comparan grupos de elementos simultáneamente.
- max_gpu.get() Devuelve el resultado al CPU
- cpu.cuda.Stream.null.synchronize asegura que la GPU termine antes de medir el tiempo

Conclusión

- La **GPU** aprovecha el **paralelismo masivo** de CUDA para realizar operaciones sobre millones de datos simultáneamente
- Sin embargo, las **transferencias de memoria** entre CPU y GPU pueden reducir la ganancia si los datos son pequeños.
- Cuanto más grande es el arreglo, **más beneficiosa es la ejecución en GPU**.
- CuPy facilita el uso de CUDA **sin necesidad de escribir kernels manuales** (lo hace internamente).

Problema 3#

Calcular la suma de todos los elementos en un vector muy grande es un problema común que requiere una estrategia de reducción paralela para ser eficiente en una GPU. Implementa un kernel de reducción para sumar todos los elementos de un vector con millones de elementos.

a. Entrada: Vector 1D con números reales (float32)

b. Salida: Número (float32) que representa la suma de los elementos del vector

```
import numpy as np
import cupy as cp
import time
import os

# 1. Definir el tamaño del vector (más de 100 millones de elementos)
SIZE = 250_000_000
print(f"Vector size: {SIZE:,} elements")

# 2. Generar el vector de ENTRADA en la CPU (NumPy)
vector_cpu = np.random.rand(SIZE).astype(np.float32) ##tipo de dato

# Verificar la GPU y CUDA
try:
                                        print(f"\nGPU                Name:
{cp.cuda.runtime.getDeviceProperties(0)['name'].decode()}")
except:
    print("\n¡ADVERTENCIA! No se detectó una GPU de NVIDIA o CUDA.
¿Activaste la GPU en el menú 'Entorno de ejecución'?")
```

```
print("---")

# ============================================================
# A. Cómputo en CPU (Usando NumPy)
# ============================================================
print("Ejecutando en CPU (Secuencial)...")
start_cpu = time.perf_counter()
# np.sum() es la operación secuencial en la CPU
suma_cpu = np.sum(vector_cpu)
end_cpu = time.perf_counter()
time_cpu = end_cpu - start_cpu

print(f"Suma Total (CPU): {suma_cpu}")
print(f"Tiempo CPU (ms): {time_cpu * 1000:.4f}")

# ============================================================
# B. Cómputo en GPU (Usando CuPy / CUDA)
# ============================================================
print("\nEjecutando en GPU (Paralelo con CUDA)...")

# **Paso 1: Transferencia de datos CPU -> GPU**
#mover vector de la ram a la vram (gpu)
start_transfer_to_gpu = time.perf_counter()
vector_gpu = cp.asarray(vector_cpu)
end_transfer_to_gpu = time.perf_counter()

# **Paso 2: Ejecución del Kernel de Reducción**
# cp.sum() invoca un Kernel CUDA de reducción optimizado (tipo árbol
binario).
start_gpu_calc = time.perf_counter()
suma_gpu = cp.sum(vector_gpu)
# Sincronización: Espera a que la GPU termine todas las operaciones
cp.cuda.Stream.null.synchronize()
end_gpu_calc = time.perf_counter()
time_gpu_calc = end_gpu_calc - start_gpu_calc

# **Paso 3: Transferencia de datos GPU -> CPU**
#mover el resultado de vuelta a la CPU para imprimirlo.
suma_resultado = suma_gpu.get()
time_gpu_total = time_gpu_calc #Solo consideramos el tiempo de cómputo
para el Speedup puro

print(f"Suma Total (GPU): {suma_resultado}")
```

```
print(f"Tiempo Cómputo GPU (ms): {time_gpu_calc * 1000:.4f}")
print(f"Tiempo Transferencia CPU->GPU (ms): {(end_transfer_to_gpu -
start_transfer_to_gpu) * 1000:.4f}")



# ============================================================
# C. Comparación de Rendimiento (Salida)
# ============================================================
print("\n" + "="*50)
print("            resultados")
print("="*50)
print(f"Tiempo CPU (Secuencial): {time_cpu * 1000:.2f} ms")
print(f"Tiempo GPU (Paralelo):   {time_gpu_total * 1000:.2f} ms")

# El speedup (aceleración) es la razón de los tiempos de ejecución.
speedup = time_cpu / time_gpu_total

print(f"ACELERACIÓN (Speedup): {speedup:.2f}x")
print("================================================")
```

Salida

```
Vector Size: 250,000,000 elements
GPU Device: Tesla T4
---
CPU Execution (numpy.sum)...
Result (CPU): 125001656.0
GPU Execution (cupy.sum)...
Result (GPU): 125001224.0

================================================
             RENDIMIENTO (CPU vs GPU)
================================================
Time CPU (ms):         140.4460
Time GPU (ms):         84.5940
Data Transfer H->D (ms): 191.1357
SPEEDUP (Acceleration): 1.66x
================================================
```

Explicación

El proceso de cómputo paralelo inicia y termina con la transferencia de datos:

- El vector de entrada (vector_cpu) se copia desde la Memoria Principal (RAM) a la Memoria Global (VRAM) de la GPU. Esta es una operación de latencia relativamente alta.
- Una vez que el kernel de reducción calcula la suma, el resultado final se copia de vuelta a la Memoria Principal para ser mostrado por Python

La suma de reducción se realiza en la GPU mediante un Kernel CUDA, donde:

- El “Grid” es la colección total de bloques que se lanzan para procesar el vector completo.
- Los “Bloques (Block)” son grupos de hilos que trabajan de manera coordinada. En la reducción, los bloques son unidades de trabajo que realizan sumas parciales.
- Cada hilo ejecuta el mismo código del kernel. En la reducción, los hilos se emparejan para sumar dos elementos (A[i] + A[i+stride])

La eficiencia se logra usando la Memoria Compartida (Shared Memory):

- Los datos del vector se cargan de la lenta Memoria Global a la ultrarrápida Memoria Compartida dentro del bloque.
- Los hilos dentro de un bloque usan la memoria compartida para realizar una suma en forma de árbol binario. En cada iteración, el número de elementos a sumar se reduce a la mitad.
- Se utiliza la función **__syncthreads()** para asegurar que todos los hilos en el mismo bloque hayan terminado de leer y escribir datos en la Memoria Compartida antes de comenzar el siguiente paso de la reducción.

Problema 4#

Implementar el algoritmo Map-Reduce utilizando Python CUDA

```
# ======================================================
# Map-Reduce numérico usando GPU (CuPy) vs CPU (NumPy)
# ======================================================
# Objetivo: agrupar (map) y reducir (sum y count) por clave (buckets).
# Ejecutar en Google Colab con GPU activada.
# ======================================================

import time
import numpy as np

# Intentaremos importar cupy; si no está instalado, descomenta la línea de instalación arriba.
try:
    import cupy as cp
except Exception as e:
      print("CuPy no está instalado o no se pudo importar. Instala cupy-cuda11x o cupy-cuda12x según tu CUDA.")
    raise e

# -------------- Parámetros y datos --------------
N = 50_000_000    # número de elementos
K = 1024          # número de claves/buckets (ej. 1024)
rng = np.random.default_rng(12345)

# Generar datos de entrada en CPU (float32)
# Usamos una distribución para que keys sean variadas
values_cpu = (rng.random(N).astype(np.float32) * 1e6)   # valores en rango [0,1e6)
# Definimos la función de mapa: key = int(value) % K  (ejemplo simple)
# Pero no calculamos keys aún (lo haremos en CPU y GPU según versión)

print(f"Datos generados: N={N}, K={K}")

# -------------- CPU (NumPy) - Map + Reduce --------------
def cpu_map_reduce(values, K):
    t0 = time.perf_counter()
    # Map: calcular claves en CPU
    keys = (values.astype(np.int64) % K)
    # Reduce: contamos ocurrencias por key y sumamos por key
```

```
    t_map = time.perf_counter()
    counts = np.bincount(keys, minlength=K)            # conteo por key
     sums = np.bincount(keys, weights=values, minlength=K)   # suma por
key
    t1 = time.perf_counter()
    return {
        "counts": counts,
        "sums": sums,
        "t_map": t_map - t0,
        "t_reduce": t1 - t_map,
        "t_total": t1 - t0
    }

print("Ejecutando Map-Reduce en CPU (NumPy)... (esto puede tardar)")
cpu_res = cpu_map_reduce(values_cpu, K)
print(f"CPU    total    time:    {cpu_res['t_total']:.4f}    s,    map:
{cpu_res['t_map']:.4f}, reduce: {cpu_res['t_reduce']:.4f}")

# -------------- GPU (CuPy) - Map + Reduce --------------
#  Idea:  copiar  datos  a  GPU,  map  keys  en  GPU,  usar  cupy.bincount
(altamente optimizado) con weights
def gpu_map_reduce_cupy(values_cpu, K):
    # Transferir host -> device
    t0 = time.perf_counter()
    arr_gpu = cp.array(values_cpu)   # copia Host->Device
    cp.cuda.Stream.null.synchronize()
    t_after_copy = time.perf_counter()

    # Map (GPU): calcular keys
    # keys = arr_gpu.astype(int) % K  -> cast y mod en GPU
    keys_gpu = (arr_gpu.astype(cp.int64) % K)
    cp.cuda.Stream.null.synchronize()
    t_map_done = time.perf_counter()

    # Reduce (GPU): bincount para counts y sums (weights)
    counts_gpu = cp.bincount(keys_gpu, minlength=K)
    sums_gpu = cp.bincount(keys_gpu, weights=arr_gpu, minlength=K)
    cp.cuda.Stream.null.synchronize()
    t_reduce_done = time.perf_counter()

    # Traer resultados a host
    counts = counts_gpu.get()
    sums = sums_gpu.get()
```

```
    t_after_get = time.perf_counter()

    return {
        "counts": counts,
        "sums": sums,
        "t_copy_H2D": t_after_copy - t0,
        "t_map": t_map_done - t_after_copy,
        "t_reduce": t_reduce_done - t_map_done,
        "t_copy_D2H": t_after_get - t_reduce_done,
        "t_total": t_after_get - t0
    }

print("Ejecutando Map-Reduce en GPU (CuPy)... (warm-up y ejecución, esto puede tardar)")
# Warm-up (compilaciones / JIT internos)
_ = cp.array(np.array([1], dtype=np.float32))
cp.cuda.Stream.null.synchronize()

gpu_res = gpu_map_reduce_cupy(values_cpu, K)
print(f"GPU times (s): H2D={gpu_res['t_copy_H2D']:.4f}, map={gpu_res['t_map']:.4f}, reduce={gpu_res['t_reduce']:.4f}, D2H={gpu_res['t_copy_D2H']:.4f}, total={gpu_res['t_total']:.4f}")

# -------------- Verificación de igualdad (sanity check) --------------
print("Verificando que CPU y GPU obtuvieron el mismo resultado para algunos buckets...")
# Comparamos sums y counts para los primeros 10 buckets
for i in range(10):
        print(f"Bucket {i}: CPU count={cpu_res['counts'][i]}, GPU count={gpu_res['counts'][i]}, CPU sum={cpu_res['sums'][i]:.4f}, GPU sum={gpu_res['sums'][i]:.4f}")

# Chequeo numérico global (tolerancia pequeña)
counts_equal = np.array_equal(cpu_res['counts'], gpu_res['counts'])
sums_close = np.allclose(cpu_res['sums'], gpu_res['sums'], rtol=1e-6, atol=1e-3)
print(f"\nCounts equal: {counts_equal}, sums close: {sums_close}")

# -------------- Reporte final --------------
cpu_time = cpu_res['t_total']
gpu_time = gpu_res['t_total']
print(f"\n==== RESUMEN ====")
print(f"CPU total: {cpu_time:.4f} s")
```

```
print(f"GPU total: {gpu_time:.4f} s (incluye transferencias)")
print(f"Speedup (CPU/GPU): {cpu_time / gpu_time:.2f}x")
```

Funcionamiento:

1. **Configuración:** Define el número de elementos (N = 50,000,000) y el número de "buckets" o claves (K = 1024). Genera un arreglo de números aleatorios (values_cpu) en la CPU.
2. **Función cpu_map_reduce (NumPy):**
    - Toma los values y K.
    - **Map:** Calcula la clave (keys) para cada valor usando la operación módulo (% K). Esto se hace en la CPU.
    - **Reduce:** Utiliza np.bincount para:
        - Contar cuántos valores caen en cada clave (counts).
        - Sumar los valores que caen en cada clave (sums) usando el argumento weights.
    - Mide y retorna los tiempos de cada paso y el total.
3. **Función gpu_map_reduce_cupy (CuPy):**
    - Toma los values_cpu (los datos originales en CPU) y K.
    - **Transferencia H2D (Host to Device):** Copia los datos de la CPU a la memoria de la GPU (cp.array(values_cpu)).
    - **Map (GPU):** Calcula las claves (keys_gpu) directamente en la GPU usando operaciones de CuPy.
    - **Reduce (GPU):** Utiliza cp.bincount en la GPU para calcular counts_gpu y sums_gpu.
    - **Transferencia D2H (Device to Host):** Copia los resultados (counts y sums) de vuelta a la memoria de la CPU (.get()).
    - Mide y retorna los tiempos de cada paso (incluyendo transferencias) y el total.
4. **Ejecución y Verificación:** Ejecuta ambas funciones, compara los resultados de los primeros buckets para verificar que sean iguales (o muy cercanos), y finalmente presenta un resumen de los tiempos y el "speedup" (aceleración) obtenido.