Instituto Politécnico Nacional

“Escuela Superior de Cómputo”

Integrantes:

- Ignacio Cortés Atzin Maxela

Unidad de aprendizaje: Cómputo paralelo

Profesor: Luis Alberto Ibáñez Zamora

Grupo: 6BM2

“Reporte de práctica 9”

Map Reduce en Cuda

índice

Resumen Práctico

En el procesamiento paralelo de imágenes con GPU, se pueden aplicar dos enfoques complementarios: **MapReduce global** y **convolución local**. El enfoque global tipo MapReduce se utiliza para operaciones que pueden aplicarse de forma independiente a cada píxel, como contar cuántos tienen una intensidad mayor a cierto umbral. En este caso, se implementó un ReductionKernel de PyCUDA para contar cuántos píxeles de una imagen en escala de grises tienen una intensidad superior a 0.8. Este tipo de procesamiento es eficiente y escalable, ya que todos los hilos GPU operan sobre datos individuales sin necesidad de comunicación entre ellos.

Por otro lado, el enfoque local mediante un **filtro promediador 3x3** requiere que cada hilo acceda no solo a su píxel asignado, sino también a los vecinos que lo rodean. Esto se logra implementando un kernel CUDA que recorre el vecindario de cada píxel (excepto en los bordes) para calcular el promedio de los 9 valores circundantes. Este tipo de procesamiento es fundamental en operaciones de suavizado, detección de bordes y convoluciones más complejas. A diferencia del enfoque global, este método implica acceso compartido a la memoria y coordinación espacial, por lo que su implementación en GPU debe manejarse cuidadosamente para optimizar el rendimiento.

Introducción

El crecimiento exponencial en el volumen de datos ha hecho indispensable el uso de arquitecturas de cómputo paralelo para acelerar tareas computacionalmente intensivas. En este contexto, las unidades de procesamiento gráfico (GPU) han cobrado relevancia debido a su capacidad para ejecutar miles de hilos en paralelo. PyCUDA permite acceder a la potencia de CUDA desde Python, facilitando la programación de kernels paralelos con una sintaxis de alto nivel. Esta práctica se enfoca en aplicar dos enfoques de paralelización sobre imágenes: uno global, utilizando el paradigma MapReduce para conteo de intensidades, y otro local, mediante un filtro promediador 3x3 que accede a vecinos. Ambos enfoques permiten explorar distintas estrategias de acceso a memoria y distribución del trabajo entre hilos de GPU, contrastando su implementación y resultados frente a su contraparte en CPU.

Metodología

Para abordar esta práctica, se utilizaron herramientas como **Python, PyCUDA, NumPy** y **matplotlib**, ejecutándose sobre un entorno que soporte GPU (como Google Colab). Se desarrollaron dos experimentos principales:

1. **MapReduce sobre imagen normalizada:**
   Se cargó una imagen en escala de grises y se normalizó al rango [0, 1]. Luego, se aplanó el arreglo bidimensional para ser procesado por GPU. Mediante ReductionKernel de PyCUDA, se aplicó un mapeo paralelo donde cada hilo evaluó si su píxel correspondiente tenía intensidad mayor a 0.8. Los resultados (1 o 0) se redujeron mediante suma, obteniendo el conteo total. Finalmente, este resultado se comparó con su equivalente en CPU utilizando np.sum() para validar consistencia.
2. **Filtro promediador 3x3 con acceso local:**
   Se diseñó un kernel CUDA personalizado que accede a cada píxel interno de la imagen y su vecindario 3x3. Cada hilo calculó el promedio de los 9 valores y escribió el resultado en la imagen de salida. Se utilizaron bloques de 16×16 hilos y memoria global para acceder a la imagen original. Posteriormente, se recuperó la imagen suavizada desde la GPU y se visualizó para observar el efecto del filtro.

Ambos enfoques fueron ejecutados sobre la misma imagen de entrada para poder comparar su rendimiento y propósito. El enfoque global permitió evaluar el poder de reducción masiva sobre grandes volúmenes de datos, mientras que el enfoque local evidenció la complejidad adicional que conlleva el acceso a memoria vecina en tareas como el suavizado de imágenes.

Marco teórico

MapReduce es un **paradigma de programación paralela** diseñado para procesar grandes volúmenes de datos de forma eficiente dividiendo el problema en dos fases:

1. Map (Mapeo)

Se aplica una función **independiente** a cada elemento de un conjunto de datos. Esto es **altamente paralelizable**, ya que no hay dependencia entre elementos.

entrada: [1, 2, 3, 4]

map: $x \mapsto x^2$

salida: [1, 4, 9, 16]

2. Reduce (Reducción)

Se **combinan los resultados del Map** en un único valor. Esta parte puede implicar una reducción jerárquica (en árbol binario), como:

entrada: [1, 4, 9, 16]

reduce: suma → 1 + 4 + 9 + 16 = 30

¿Cómo se implementa en CUDA?

En CUDA (y PyCUDA), la implementación de MapReduce se divide así:

1. Map: procesamiento paralelo

Cada **hilo de GPU (thread)** toma un elemento del arreglo y ejecuta la función de mapeo.

- Cada hilo ejecuta su propio cálculo independiente.
- Este paso está limitado solo por el número de hilos que puedas lanzar.

2. Reduce: sincronización y jerarquía

La reducción requiere:

- **Sincronización** entre hilos.
- **Accesos coordinados a memoria compartida (shared memory)**.
- Opcionalmente, una reducción en múltiples pasos:
    - **Intra-bloque**: reducción dentro de un bloque de hilos.
    - **Inter-bloque**: resultados parciales enviados a la CPU o a un nuevo kernel para finalización.

Implementación del código

Objetivos:

- Comprender la estructura del paradigma MapReduce en GPU.
- Aplicar funciones condicionales dentro de un kernel CUDA.
- Comparar rendimiento y exactitud entre CPU y GPU.

Competencia a desarrollar:

1. Genera un arreglo de al menos **1 millón de números aleatorios** entre 0 y 1.
2. Utiliza ReductionKernel de PyCUDA para aplicar:
    - **Map:** Evaluar si cada número es mayor al umbral (por ejemplo, 0.8) y asignar 1.0 si cumple, 0.0 si no.
    - **Reduce:** Sumar todos los 1.0 para obtener el **conteo total** de elementos que cumplen la condición.
3. Muestra el número total de elementos que superan el umbral.
4. Compara el resultado obtenido por GPU con un cálculo equivalente en CPU (usando NumPy) para verificar consistencia.
5. Compara el resultado obtenido por GPU con un cálculo equivalente en CPU (usando NumPy) para verificar consistencia.

Ilustración 1 Se instala cuda en Google colab

```
[3]
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
import numpy as np
from pycuda.reduction import ReductionKernel
import time

# Parámetros
umbral = 0.8
n_datos = 1_000_000

# Datos aleatorios
np.random.seed(42)
host_data = np.random.rand(n_datos).astype(np.float32)

# GPU: MapReduce
gpu_data = gpuarray.to_gpu(host_data)

# Definimos el kernel MapReduce
reduce_kernel = ReductionKernel(np.float32,
    neutral="0",
    reduce_expr="a + b",
    map_expr=f"(x[i] > {umbral}) ? 1.0 : 0.0",
    arguments="float *x")

# Tiempo GPU
start_gpu = time.time()
conteo_gpu = reduce_kernel(gpu_data).get()
end_gpu = time.time()

# CPU
start_cpu = time.time()
conteo_cpu = np.sum(host_data > umbral)
end_cpu = time.time()

# Resultados
print(f"GPU: {int(conteo_gpu)} elementos > {umbral} (tiempo: {end_gpu - start_gpu:.6f}s)")
print(f"CPU: {int(conteo_cpu)} elementos > {umbral} (tiempo: {end_cpu - start_cpu:.6f}s)")
```

```
GPU: 200252 elementos > 0.8 (tiempo: 0.003074s)
CPU: 200252 elementos > 0.8 (tiempo: 0.001179s)
```

Ilustración 2 Resultado del código

Parte A: MapReduce - Conteo de Intensidades Altas

1. Carga una imagen en escala de grises y normalízala al rango [0, 1].
2. Implementa un ReductionKernel de PyCUDA que:
    - Evalúe si cada píxel tiene intensidad > 0.8.
    - Devuelva el número total de píxeles que cumplen esa condición.
3. Muestra el resultado y compáralo con el equivalente en CPU (np.sum(img > 0.8)).

Este código realiza un conteo paralelo de píxeles brillantes en una imagen utilizando PyCUDA. Primero, se carga una imagen (flor.jpg), se convierte a escala de grises y se normaliza al rango [0, 1]. Luego, la imagen se aplana en un vector para facilitar su procesamiento en la GPU. A través de ReductionKernel, se define una operación tipo MapReduce donde cada hilo evalúa si el valor de intensidad de su píxel es mayor a 0.8 (Map) y asigna un 1 si cumple, o 0 en caso contrario. Estos valores binarios son luego sumados (Reduce) para obtener el conteo total de píxeles que superan el umbral. Finalmente, se compara el resultado obtenido por GPU con el cálculo equivalente realizado en CPU utilizando NumPy (np.sum) para validar la consistencia entre ambos métodos.

```python
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import pycuda.autoinit
import pycuda.driver as cuda
from pycuda.reduction import ReductionKernel

# Cargar imagen en escala de grises y normalizar
img = Image.open("flor.jpg").convert("L")
img_array = np.asarray(img).astype(np.float32) / 255.0
img_flat = img_array.flatten()

# Crear kernel de reducción para contar pixeles > 0.8
rk = ReductionKernel(np.int32, neutral="0",
    reduce_expr="a+b",
    map_expr="x[i] > 0.8 ? 1 : 0",
    arguments="const float *x")

# Ejecutar en GPU
import pycuda.gpuarray as gpuarray
gpu_img = gpuarray.to_gpu(img_flat)
result_gpu = rk(gpu_img).get()

# Comparar con CPU
result_cpu = np.sum(img_flat > 0.8)
print("Conteo GPU:", result_gpu)
print("Conteo CPU:", result_cpu)
```

```
Conteo GPU: 186142
Conteo CPU: 182486
```

Ilustración 3 Código2

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Cargar imagen y convertirla a escala de grises
image_path = "flor.jpg"
img = Image.open(image_path).convert("L")
img_array = np.asarray(img).astype(np.float32) / 255.0  # Normalizar

# Mostrar la imagen original normalizada
plt.imshow(img_array, cmap='gray')
plt.title("Imagen Normalizada en Escala de Grises")
plt.axis('off')
plt.show()

# Contar píxeles con intensidad > 0.8 usando CPU
high_intensity_count_cpu = np.sum(img_array > 0.8)
high_intensity_count_cpu
```


```
np.int64(182486)
```

Ilustración 4 Código 2 implementación de imagen

Este fragmento de código carga una imagen (flor.jpg), la convierte a escala de grises y la normaliza al rango [0, 1] para facilitar su análisis numérico. Luego, utiliza matplotlib para visualizar la imagen procesada en tonos de gris. Posteriormente, emplea NumPy para contar cuántos píxeles tienen una intensidad mayor a 0.8, lo cual permite identificar regiones brillantes en la imagen. Este conteo se realiza completamente en CPU como referencia para futuras comparaciones con enfoques de procesamiento paralelo en GPU.

Parte B: Filtro Promediador 3x3 con Acceso Local

1. Sobre la misma imagen, implementa un **kernel CUDA** que:
    - Para cada píxel no borde, acceda a su vecindario 3x3.
    - Calcule el promedio de esos 9 valores.
    - Asigne el resultado a la imagen de salida.
2. Visualiza la imagen suavizada.

```
[ ]  import numpy as np
     import matplotlib.pyplot as plt
     from PIL import Image
     import pycuda.autoinit
     import pycuda.driver as cuda
     from pycuda.compiler import SourceModule

     # Cargar imagen (usa tu ruta o súbela a Colab)
     img = Image.open("flor.jpg").convert("L")
     img_array = np.asarray(img).astype(np.float32) / 255.0

     # Dimensiones
     height, width = img_array.shape
     img_flat = img_array.flatten()
     output_flat = np.zeros_like(img_flat)

     # Memoria en GPU
     img_gpu = cuda.mem_alloc(img_flat.nbytes)
     output_gpu = cuda.mem_alloc(output_flat.nbytes)
     cuda.memcpy_htod(img_gpu, img_flat)

     # Kernel CUDA para filtro 3x3
     kernel_code = """
     __global__ void average_filter(const float *img, float *out, int width, int height) {
         int x = blockIdx.x * blockDim.x + threadIdx.x;
         int y = blockIdx.y * blockDim.y + threadIdx.y;

         if (x >= 1 && x < width - 1 && y >= 1 && y < height - 1) {
             float sum = 0.0;
             for (int dy = -1; dy <= 1; dy++) {
                 for (int dx = -1; dx <= 1; dx++) {
                     int idx = (y + dy) * width + (x + dx);
                     sum += img[idx];
                 }
             }
             out[y * width + x] = sum / 9.0;
         } else {
             out[y * width + x] = img[y * width + x];  // mantener bordes
         }
     }
     """
```

Ilustración 5 Código 3

```
mod = SourceModule(kernel_code)
average_filter = mod.get_function("average_filter")

# Ejecutar kernel
block_size = (16, 16, 1)
grid_size = ((width + 15) // 16, (height + 15) // 16, 1)
average_filter(img_gpu, output_gpu, np.int32(width), np.int32(height), block=block_size, grid=grid_size)

# Recuperar resultado
cuda.memcpy_dtoh(output_flat, output_gpu)
output_img = output_flat.reshape((height, width))

# Mostrar imagen original y suavizada
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.imshow(img_array, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(output_img, cmap='gray')
plt.title("Suavizada con filtro 3x3")
plt.axis('off')
plt.show()
```

Ilustración 6 Resultado código 3

Este código aplica un filtro promediador 3x3 a una imagen en escala de grises utilizando PyCUDA para ejecutar el procesamiento de forma paralela en la GPU. Primero, se carga y normaliza la imagen (flor.jpg), luego se reserva memoria en la GPU para la imagen original y la de salida. A continuación, se define un kernel CUDA que recorre cada píxel (excepto los bordes) y calcula el promedio de su vecindario 3x3 para suavizar la imagen, mientras que los píxeles de borde se conservan sin cambios. El kernel se ejecuta usando bloques de 16×16 hilos, y el resultado se transfiere de vuelta a la CPU. Finalmente, se visualizan lado a lado la imagen original y la imagen suavizada para observar el efecto del filtrado. Este enfoque demuestra cómo acceder a datos vecinos (acceso local) en un entorno de cómputo paralelo con GPU.

Código

https://colab.research.google.com/drive/17Em34ujRHaVXdo_FXaWhms3Xzqj8JeK-?usp=sharing

Conclusiones

A lo largo de esta práctica se exploraron dos enfoques fundamentales de procesamiento paralelo en GPU aplicados al tratamiento de imágenes mediante PyCUDA: el enfoque global tipo MapReduce y el enfoque local mediante filtrado convolucional 3x3. Ambos métodos permitieron comprender cómo distribuir eficientemente el trabajo entre miles de hilos de la GPU y cómo estructurar kernels adecuados según el tipo de operación que se desea realizar sobre los datos.

El enfoque global, implementado a través de ReductionKernel, demostró ser altamente eficaz para realizar conteos rápidos sobre grandes volúmenes de datos, como identificar cuántos píxeles de una imagen tienen una intensidad mayor a un umbral determinado. Este tipo de operación no requiere acceso a datos vecinos, por lo que cada hilo puede operar de forma completamente independiente, aprovechando al máximo el paralelismo de la GPU. Además, la comparación con la operación equivalente en CPU evidenció la consistencia y precisión del resultado, destacando también mejoras de rendimiento en ciertos casos.

Por otro lado, el enfoque local con un filtro promediador 3x3 mostró cómo cada hilo debe acceder a su píxel y a los ocho vecinos circundantes para calcular un valor promedio, lo cual implica un patrón de acceso más complejo. Este tipo de procesamiento es ideal para tareas de suavizado o detección de características en imágenes, aunque su implementación en CUDA requiere mayor cuidado en la gestión de bordes y acceso a memoria.

Esta práctica permitió evidenciar las ventajas del cómputo paralelo en GPU y la utilidad de PyCUDA como herramienta para acelerar operaciones sobre imágenes. Se comprobó que el enfoque global es ideal para tareas de resumen o conteo, mientras que el enfoque local es esencial para tareas de procesamiento espacial, como filtros o convoluciones. Dominar ambos enfoques amplía significativamente la capacidad de diseñar soluciones eficientes para problemas de procesamiento de datos visuales en entornos de alto rendimiento.

Cuestionario

Cuestionario: ¿Qué está pasando en el código MapReduce con PyCUDA?

1. ¿Qué tipo de datos se están generando en el arreglo **host_data**? ¿Por qué se usa **astype(np.float32)**?

   Se están generando números aleatorios entre 0 y 1 con np.random.rand(), y se convierten explícitamente al tipo float32 porque es el formato de punto flotante compatible y más eficiente para operar en GPU. Además, PyCUDA espera este tipo de dato para evitar errores de compatibilidad al transferir datos y procesarlos en la GPU.

2. ¿Cuál es la función de **gpuarray.to_gpu(host_data)** en el flujo del programa?

   Esa función transfiere los datos desde la memoria de la CPU (host) a la memoria de la GPU (device), lo cual es necesario para que los kernels CUDA puedan procesarlos en paralelo. Sin esta transferencia, el kernel no tendría acceso a los datos.

3. ¿Qué representa la expresión **map_expr=f"(x[i] > {umbral}) ? 1.0 : 0.0"** dentro del **ReductionKernel**?

   Esa expresión es la parte "Map" del paradigma MapReduce. Evalúa si cada elemento del arreglo es mayor al umbral. Si lo es, devuelve 1.0; si no, 0.0. Así se transforma un arreglo de valores flotantes en un arreglo binario que indica qué elementos cumplen la condición.

4. ¿Por qué el valor **neutral** se establece en **"0"**? ¿Qué pasaría si fuera distinto?

   El valor neutral es el punto de partida para la operación de reducción (en este caso, suma). Se establece en `"0"` porque 0 es el elemento neutro de la suma; no altera el resultado. Si se usara otro valor (por ejemplo, 1), el conteo estaría incorrectamente desplazado por ese valor inicial.

5. ¿Qué operación se define con **reduce_expr="a + b"** y qué efecto tiene sobre los valores generados por el **map_expr**?

Define la operación de reducción como una suma entre los resultados del mapeo. Es decir, suma todos los 1.0 y 0.0 generados por el `map_expr`, lo que da como resultado el número total de elementos que cumplen la condición (valores mayores al umbral).

6. ¿Cómo contribuye el paradigma MapReduce a acelerar esta tarea en comparación con un **for** tradicional en CPU?

MapReduce permite que todos los elementos se evalúen en paralelo en la GPU, usando miles de hilos. En contraste, un `for` en CPU ejecuta las evaluaciones secuencialmente. Esto representa una mejora significativa en velocidad para grandes volúmenes de datos (como 1 millón de elementos).

7. ¿Por qué se espera que el resultado del **reduce_kernel(gpu_data).get()** sea un número entero? ¿Qué significa ese valor?

Porque el resultado es la suma de muchos 1s y 0s (valores booleanos convertidos a flotantes), lo cual siempre dará un número entero que representa cuántos elementos cumplen la condición. Ese número es el conteo total de valores mayores al umbral.

8. ¿Qué sucede si cambiamos el valor del umbral de **0?8** a **0.2**? ¿Cómo afectará esto al resultado y a la carga de la GPU?

El resultado aumentará, porque más elementos del arreglo cumplirán con la condición (mayores a 0.2). Esto no cambia la cantidad de trabajo para cada hilo (la carga es constante por elemento), pero puede afectar la distribución de valores que deben ser sumados al final.

9. ¿Cuál sería el propósito de comparar el resultado de la GPU con **np.sum(host_data > 0.8)** en CPU?

Verificar que el resultado obtenido por la GPU es correcto y consistente con el resultado obtenido por CPU. Esta comparación sirve como validación de que la implementación en PyCUDA es funcional y no presenta errores lógicos o de sincronización.

10. ¿Cómo modificarías el kernel para contar cuántos valores están dentro de un rango, por ejemplo, entre **0?3** y **0.7**?

    map_expr="(x[i] >= 0.3 && x[i] <= 0.7) ? 1.0 : 0.0"

    Esto hará que el mapeo devuelva 1.0 solo si el valor está dentro del rango [0.3, 0.7], y 0.0 en caso contrario.

¿Por qué MapReduce no es adecuado para operaciones basadas en vecinos?
El paradigma MapReduce está diseñado para tareas donde cada elemento puede procesarse de forma independiente, sin necesidad de acceder a información de otros elementos. En operaciones basadas en vecinos, como filtros convolucionales, cada dato requiere conocer su entorno inmediato (por ejemplo, los valores de un vecindario 3x3). Esta dependencia espacial impide el procesamiento completamente independiente que caracteriza a MapReduce. Además, MapReduce no tiene un mecanismo integrado para acceder a múltiples posiciones relativas por hilo, lo que lo hace ineficiente o incluso inviable para operaciones que dependen de la posición en una matriz o de relaciones espaciales.

¿Qué diferencias observaste en la implementación y estructura de los kernels? En los kernels basados en MapReduce, como los generados por ReductionKernel, la estructura es simple: cada hilo aplica una operación a un solo dato (Map) y luego se combinan los resultados de forma jerárquica (Reduce), todo desde una función de alto nivel. En cambio, los kernels para operaciones locales requieren programación CUDA explícita, donde se debe calcular manualmente la posición (x, y) de cada hilo, acceder a múltiples elementos del arreglo (vecinos), y manejar bordes de forma especial. También se requiere mayor cuidado con el uso de memoria y sincronización, ya que varios hilos pueden necesitar acceso a regiones solapadas del arreglo.

¿Qué tipo de aplicaciones se benefician más de cada enfoque? MapReduce es ideal para aplicaciones de análisis de grandes volúmenes de datos donde cada elemento puede evaluarse por separado, como conteos, histogramas, filtrado de condiciones, estadísticas y agregaciones masivas. Por su parte, las operaciones basadas en vecinos son fundamentales en procesamiento de imágenes, simulaciones físicas, detección de patrones espaciales y redes neuronales convolucionales. Estas tareas requieren capturar relaciones locales entre los datos, por lo que no pueden resolverse con estrategias globales como MapReduce.