**Taller Práctico 2: Detección de bordeado mediante algoritmos paralelos (SOBEL ALGORITHM)**

Por: Jose Santiago Gonzalez y Oscar Mateo Arrubla

**OBJETIVO DEL LABORATORIO**

El objetivo de este laboratorio es implementar y comparar la eficiencia de un algoritmo secuencial en CPU y uno paralelo en GPU al procesar una imagen El resultado de este laboratorio será un informe que demuestre el speedup (aceleración) logrado con el paralelismo

**MARCO TEÓRICO**

La detección de bordes es una operación fundamental en el procesamiento de imágenes, ya que permite identificar regiones donde ocurre una variación brusca de intensidad. El operador Sobel es uno de los métodos más usados debido a su capacidad para resaltar cambios de luminosidad mediante la aproximación de derivadas parciales.

Este operador utiliza dos convoluciones:

- Kernel horizontal (Kx): detecta bordes verticales
- Kernel vertical (Ky): detecta bordes horizontales

Para cada píxel, se toma una ventana 3×3 alrededor y se realiza la convolución con ambos kernels. Los resultados obtenidos, Gx y Gy, representan la magnitud del cambio en cada dirección.

La magnitud total del borde se calcula aplicando la norma euclidiana:

$$\text{Magnitude}(G) = \sqrt{G_x^2 + G_y^2}$$

Cuando esta magnitud es alta, significa que hay un borde claro entre píxeles vecinos.

En cuanto al paralelismo, la convolución de Sobel es un proceso altamente paralelizable, ya que el cálculo de cada píxel es independiente. Sin embargo, existen sobrecostos asociados a la creación y sincronización de procesos que pueden afectar el rendimiento en imágenes pequeñas.

**METODOLOGÍA**

**Hardware:**

- CPU: 2 vCPU
- RAM 12 GB
- Entorno: Google Colab

**Software:**

- Lenguaje usado: Python 3.x
- Librerías usadas: itertools, math, multiprocessing, time.

**Algoritmos desarrollados:**

- **Secuencial:**

1. Se carga la imagen y se convierte a escala de grises.
2. Se recorre píxel por píxel (excluyendo bordes).
3. Para cada píxel, se obtiene una ventana 3×3.

4. Se aplica la convolución con los kernels Kx y Ky mediante sumatoria manual.
5. Se calcula la magnitud del gradiente y se almacena en la imagen de salida.
6. Se mide el tiempo total de ejecución.

```
print("\n" + "="*60)
print("EJECUTANDO ALGORITMO SECUENCIAL (1 CORE)")
print("="*60)

start_time = time.time()
sequential_result = sobel_sequential(gray_image)
sequential_time = time.time() - start_time

print(f"Tiempo de ejecución: {sequential_time:.4f} segundos")
```

- **Paralelo:**

1. Se determina el número de núcleos disponibles con mp.cpu_count().
2. Se divide la imagen en “chunks” (bloques horizontales), asignando a cada núcleo un grupo de filas.
3. Cada proceso calcula el operador Sobel en su bloque de forma independiente.
4. Cada proceso devuelve su sección procesada.
5. El programa principal ensambla las partes de la imagen para formar el resultado final.
6. Se mide el tiempo total del procesamiento paralelo.

```
print("\n" + "="*60)
print("EJECUTANDO ALGORITMO PARALELO (MULTICORE)")
print("="*60)

num_cores = mp.cpu_count()
print(f"Número de cores disponibles: {num_cores}")

start_time = time.time()
parallel_result = sobel_parallel(gray_image, num_cores)
parallel_time = time.time() - start_time

print(f"Tiempo de ejecución: {parallel_time:.4f} segundos")
```

**RESULTADOS**

Para esta práctica se procesó una imagen de dimensiones 370 × 320 píxeles mediante el operador Sobel utilizando dos enfoques: un algoritmo secuencial en CPU y un algoritmo paralelo empleando dos núcleos disponibles en Google Colab.
Los tiempos obtenidos fueron los siguientes:

Algoritmo Secuencial: 1.6948 s
Algoritmo Paralelo: 1.7732 s
Speedup: 0.96×
Eficiencia: 47.79% (con 2 cores)
Al analizar los resultados visuales, ambos métodos producen bordes prácticamente idénticos, lo que confirma que el paralelismo no afecta la calidad del procesamiento, sino únicamente su tiempo de ejecución. Las imágenes finales incluyen la versión original, la conversión a escala de grises y los resultados del filtro Sobel aplicado de ambas formas.


**ANÁLISIS DE RENDIMIENTO**

Los tiempos obtenidos muestran que, para esta imagen y este tamaño de entrada, el algoritmo paralelo resultó ligeramente más lento que el secuencial. Aunque en teoría dividir el trabajo entre múltiples núcleos debería reducir el tiempo total (idealmente Speedup = número de núcleos, es decir, 2×), en la práctica no fue así.

Esto se debe principalmente a:

**1. Sobrecarga del paralelismo:**

Crear procesos, dividir la imagen, enviar datos a cada proceso y luego combinar los resultados genera un costo adicional que, en problemas pequeños o medianos, puede superar la ganancia del paralelismo.

**2. Tamaño de la imagen relativamente reducido:**

Con solo 370×320 píxeles, el trabajo asignado a cada proceso no es lo suficientemente grande como para compensar la sobrecarga.

**3. Limitaciones de multiprocessing en Python:**

Python no es óptimo para tareas paralelas muy pequeñas. El modelo de procesos funciona mejor con cargas más pesadas.

Por eso, aunque el paralelismo completó la tarea correctamente, su eficiencia fue menor (47.79%) y no logró superar al enfoque secuencial.

Para los cálculos se utilizaron estas formulas:

$$Speedup = \frac{T_{secuencial}}{T_{Paralelo}}$$

$$Eficiencia = \frac{Speedup}{Numero\ de\ nucleos} X100$$

**CONCLUSIÓN**

El experimento permitió evidenciar que **el paralelismo no siempre es la solución más rápida**, especialmente cuando:

- El problema es pequeño,
- Los datos no son voluminosos,
- La sobrecarga de coordinación entre procesos es considerable.

En este caso, el algoritmo secuencial fue más eficiente debido a que la imagen procesada no generó suficiente carga de trabajo como para que el procesamiento paralelo compensara el costo de crear y sincronizar procesos.