Daniel Ayala Guzmán
Samuel Chaves
Cristian Bello

Taller Práctico 2: Detección de Bordeado mediante GPUs y CUDA (SOBEL ALGORITHM)

---

1. Objetivo del Laboratorio

Implementar y comparar la eficiencia de un algoritmo de detección de bordes **Sobel** en **CPU** y **GPU (CUDA)** usando Google Colab.
El propósito es medir el *speedup* obtenido con el procesamiento paralelo en GPU, demostrando la ventaja del cómputo masivo de datos.

---

2. Marco Teórico

La **detección de bordes** busca identificar los puntos donde cambia bruscamente la intensidad de una imagen, marcando los límites de los objetos.

2.1. Operador Sobel

El operador Sobel aplica dos filtros de convolución 3×3 sobre la imagen en escala de grises para aproximar la derivada en las direcciones **horizontal (Gx)** y **vertical (Gy)**:

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

El resultado de cada convolución se combina para obtener la magnitud del gradiente:

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

2.2. Sobel en GPU

En CUDA, cada *hilo* procesa un píxel, lo que permite realizar millones de operaciones simultáneamente.
El método aprovecha el paralelismo masivo de la GPU, donde cada bloque de hilos calcula subconjuntos de la imagen.

3. Metodología

3.1. Configuración del Hardware (Colab)

Componente	Descripción
Entorno	Google Colab con acelerador GPU
GPU	NVIDIA Tesla T4 (2560 núcleos CUDA, 16 GB VRAM)
CPU	Intel Xeon virtual 2 núcleos
RAM	12 GB
Sistema operativo	Ubuntu (entorno Colab)

3.2. Configuración del Software

Componente	Versión
Python	3.10
CuPy	13.x (CUDA 12.x)
OpenCV	4.x (headless)
Matplotlib	3.x

3.3. Procedimiento

1. Cargar imagen RGB y convertir a escala de grises.
2. Implementar Sobel **CPU** usando bucles y máscaras 3×3.
3. Implementar Sobel **GPU** con `cupy.RawKernel` en Colab.
4. Calcular el tiempo de ejecución de ambos métodos.
5. Generar imágenes de salida y comparar resultados visualmente.
6. Calcular *speedup* = *tiempo_CPU / tiempo_GPU*.

---

4. Desarrollo e Implementación

4.1. Sobel en CPU (secuencial)

Se realiza una doble iteración (x,y) sobre los píxeles interiores de la imagen, aplicando los kernels 3×3 manualmente.

```
import cv2, numpy as np, time
Kx = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
Ky = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])

def sobel_cpu(gray):
    h, w = gray.shape
    out = np.zeros_like(gray)
    for y in range(1, h-1):
        for x in range(1, w-1):
            gx = np.sum(Kx * gray[y-1:y+2, x-1:x+2])
            gy = np.sum(Ky * gray[y-1:y+2, x-1:x+2])
            val = np.sqrt(gx**2 + gy**2)
            out[y,x] = 255 if val > 255 else val
    return out.astype(np.uint8)
```

4.2. Sobel en GPU (paralelo CUDA)

Implementado mediante `cupy.RawKernel` donde cada hilo CUDA calcula el valor de un píxel.

```
import cupy as cp
kernel = r'''
extern "C" __global__
void sobel_u8(const unsigned char* gray, unsigned char* out, int w, int h){
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if(x==0||y==0||x>=w-1||y>=h-1)return;
    int xm1=x-1,xp1=x+1,ym1=y-1,yp1=y+1;
    int p00=gray[ym1*w+xm1],p01=gray[ym1*w+x],p02=gray[ym1*w+xp1];
    int p10=gray[y*w+xm1],p12=gray[y*w+xp1];
    int p20=gray[yp1*w+xm1],p21=gray[yp1*w+x],p22=gray[yp1*w+xp1];
    int gx=(-1*p00)+(0*p01)+(1*p02)+(-2*p10)+(2*p12)+(-1*p20)+(1*p22);
    int gy=(1*p00)+(2*p01)+(1*p02)+(-1*p20)+(-2*p21)+(-1*p22);
    float mag=sqrtf(gx*gx+gy*gy);
    out[y*w+x]=(mag>255)?255:(unsigned char)mag;
}
'''
```

El kernel se lanza con bloques de 16×16 hilos, distribuidos dinámicamente según el tamaño de la imagen.

---

5. Resultados

Tipo	Tiempo (s)	Speedup
CPU (Colab Xeon)	0.132	1.0×
GPU (Tesla T4)	0.021	**≈6.3×**

Observaciones:

- Las imágenes `borde_cpu.png` y `borde_gpu.png` muestran los mismos contornos.
- La GPU mantiene una precisión equivalente a la CPU.
- El rendimiento mejora más con imágenes grandes (>720p).

---

6. Análisis de Rendimiento

- El Sobel CPU procesa píxel por píxel de forma secuencial.
- La GPU ejecuta la misma operación en miles de hilos simultáneos, obteniendo una ganancia significativa.
- En Colab, el tiempo de transferencia entre memoria CPU↔GPU es mínimo porque la imagen se mantiene en VRAM.
- La **eficiencia aumenta** con la resolución de la imagen y el número de bloques CUDA lanzados.

Speedup ≈ 6× confirma la ventaja del cómputo paralelo.
