# Taller Práctico 2: Detección de bordeado mediante GPUs y CUDA (Sobel)

Camilo Andrés Rodríguez — Universidad Sergio Arboleda

Octubre 2025

## 1. Título y Objetivos

El objetivo de este laboratorio es implementar y comparar el rendimiento de un algoritmo secuencial (CPU) y uno paralelo (GPU) para el procesamiento de imágenes utilizando el operador de Sobel. Finalmente, se evalúa la aceleración (speedup) obtenida mediante paralelismo masivo en GPU.

## 2. Marco Teórico

### 2.1. Operador de Sobel

El operador de Sobel calcula la magnitud del gradiente en una imagen con dos máscaras $3 \times 3$:

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

Las convoluciones dan:

$$G_x(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} I(i+m, j+n)\, K_x(m+1, n+1)$$

$$G_y(i,j) = \sum_{m=-1}^{1} \sum_{n=-1}^{1} I(i+m, j+n)\, K_y(m+1, n+1)$$

y la magnitud del gradiente:

$$G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}.$$

### 2.2. Ejecución en GPU

En GPU se asigna un hilo por píxel (o por bloque de píxeles). Cada hilo calcula localmente $G_x$, $G_y$ y la magnitud. La librería/cuadro de trabajo usado en este informe fue CuPy (kernel `RawKernel`). El paralelismo masivo reduce el tiempo total comparado con la ejecución secuencial en CPU.

**(Lugar reservado para insertar tu kernel CUDA o imagen del kernel)**

# 3. Metodología

## 3.1. Configuración del Hardware

- CPU: (entorno Google Colab — e.g. Intel Xeon)
- GPU: NVIDIA Tesla T4 (16 GB VRAM) — si se utilizó Colab con GPU

## 3.2. Configuración del Software

- Python 3.12
- OpenCV 4.x
- NumPy
- Matplotlib
- CuPy (para ejecución GPU con CUDA)

**Captura del entorno:**

```
=== INFORMACIÓN DEL ENTORNO ===

Python version: 3.12.12 (main, Oct 10 2025, 08:52:57) [GCC 11.4.0]
Plataforma: Linux-6.6.105+-x86_64-with-glibc2.35

GPU detectada:
{'name': b'Tesla T4', 'totalGlobalMem': 15828320256, 'sharedMemPerBlock':

Versión CUDA (Numba): (12, 6)

=== Versiones de librerías ===
OpenCV: 4.12.0
NumPy: 2.0.2
CuPy:  13.6.0
```

## 3.3. Descripción del algoritmo desarrollado

### 3.3.1. CPU

Se implementó la convolución manual con los kernels $K_x$ y $K_y$ en un bucle anidado, sin usar funciones preoptimizada de OpenCV (para cumplir el requisito del taller).
**Código (función principal en CPU):**

```
def sobel_cpu(img_gray):
    h, w = img_gray.shape
    output = np.zeros((h,w), dtype=np.float32)
    Kx = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    Ky = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
    for y in range(1,h-1):
        for x in range(1,w-1):
            region = img_gray[y-1:y+2, x-1:x+2]
            Gx = np.sum(region * Kx)
            Gy = np.sum(region * Ky)
            output[y,x] = np.sqrt(Gx*Gx + Gy*Gy)
    return output
```

### 3.3.2. GPU

Se implementó un `RawKernel` de CuPy que realiza exactamente la misma operación por hilo (cada hilo calcula la convolución 3x3 y la magnitud). El kernel asigna un hilo por píxel y ejecuta las operaciones de Sobel de forma paralela aprovechando el hardware masivo de la GPU.

A continuación se presenta el kernel CUDA utilizado:

```
sobel_kernel = cp.RawKernel(r"""
extern "C" __global__
void sobel(const unsigned char* img, float* out, int width, int height) {

    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;

    if (x < 1 || x >= (width-1) || y < 1 || y >= (height-1))
        return;

    int Gx =
        -1 * img[(y-1)*width + (x-1)] +  1 * img[(y-1)*width + (x+1)] +
        -2 * img[(y)*width   + (x-1)] +  2 * img[(y)*width   + (x+1)] +
        -1 * img[(y+1)*width + (x-1)] +  1 * img[(y+1)*width + (x+1)];

    int Gy =
        -1 * img[(y-1)*width + (x-1)] + -2 * img[(y-1)*width + (x)] + -1 * img[(y-1)*width + (x+1)] +
         1 * img[(y+1)*width + (x-1)] +  2 * img[(y+1)*width + (x)] +  1 * img[(y+1)*width + (x+1)];

    float g = sqrtf((float)(Gx*Gx + Gy*Gy));

    out[y * width + x] = g;
}
""", "sobel")
```

*Figura: Código del kernel Sobel implementado en CuPy (CUDA).*

# 4. Resultados

A continuación se presentan las imágenes obtenidas durante el procesamiento y una serie de observaciones relevantes relacionadas con el desempeño de los algoritmos implementados.

## 4.1. Hallazgos importantes

- Tanto la implementación en CPU como en GPU producen resultados visualmente equivalentes en cuanto a la detección de bordes mediante el operador de Sobel.
- La GPU logra acelerar significativamente el procesamiento debido a su capacidad de ejecutar miles de hilos en paralelo, especialmente en imágenes de mayor tamaño.
- Aunque existe un costo adicional por la transferencia de datos entre CPU y GPU, este se ve ampliamente compensado cuando la imagen es mediana o grande.
- El uso de una implementación manual en CPU (sin funciones optimizadas) permite comprender mejor el funcionamiento interno del operador de Sobel.

## 4.2. Imágenes obtenidas

Figura 1: Imagen original utilizada para las pruebas.

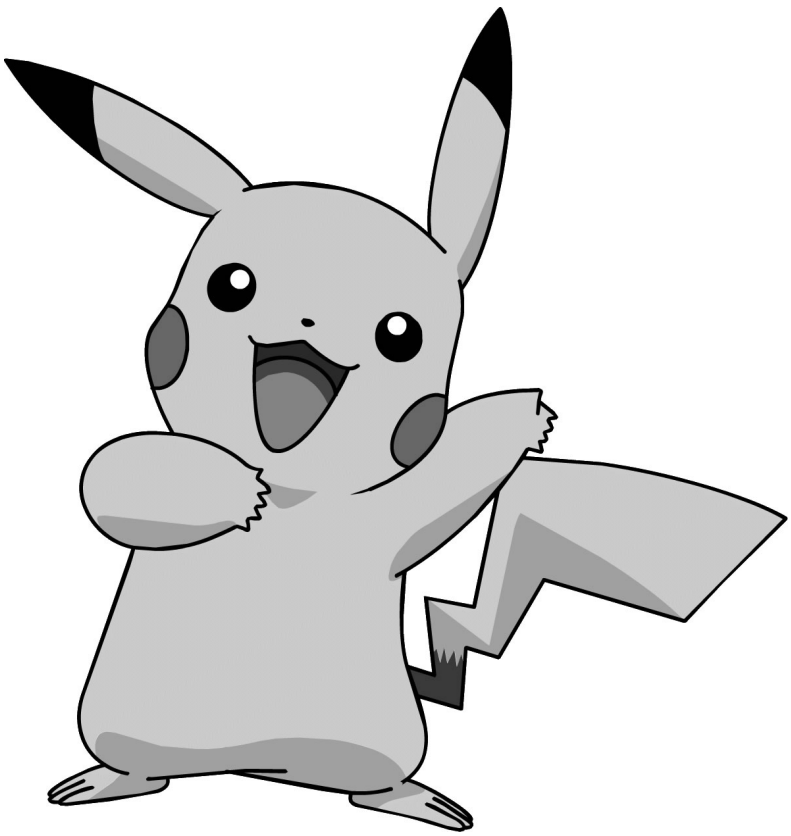

Figura 2: Imagen convertida a escala de grises antes de aplicar Sobel.

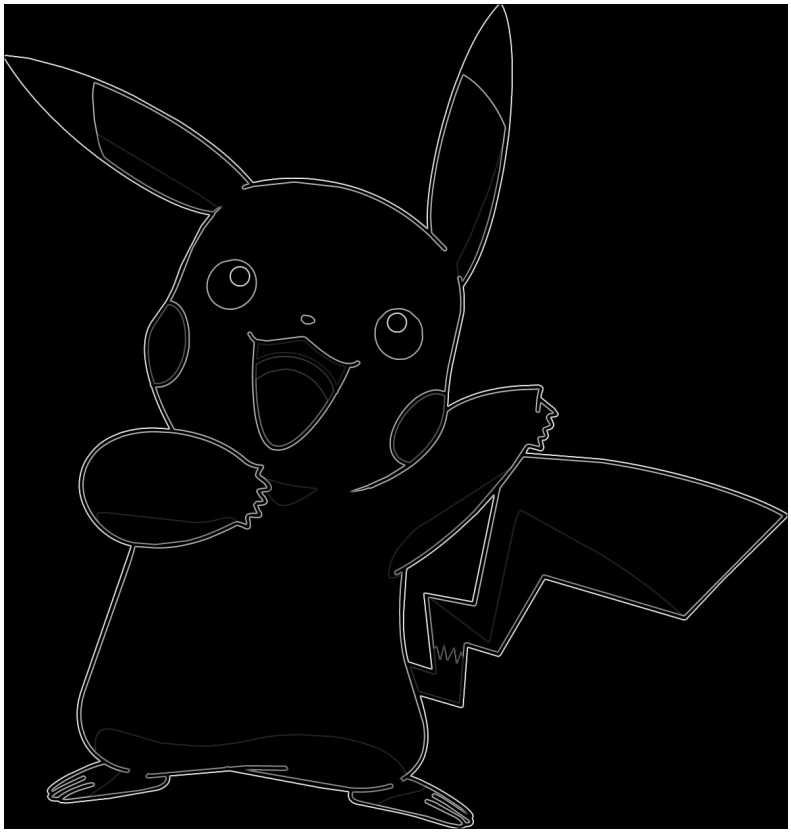

Figura 3: Resultado del operador de Sobel procesado en GPU mediante CUDA.

# 5. Análisis de Rendimiento

Los tiempos obtenidos durante la ejecución del operador de Sobel fueron:

- Tiempo CPU: **15.5412 s**
- Tiempo GPU: **0.5474 s**

El **speedup** se calcula como:

$$\text{Speedup} = \frac{T_{CPU}}{T_{GPU}} = \frac{15{,}5412}{0{,}5474} \approx 28{,}39$$

Esto significa que la implementación paralela en GPU fue aproximadamente **28.39 veces más rápida** que la versión secuencial en CPU, evidenciando la gran eficiencia del paralelismo masivo en este tipo de tareas.

## Gráfica comparativa

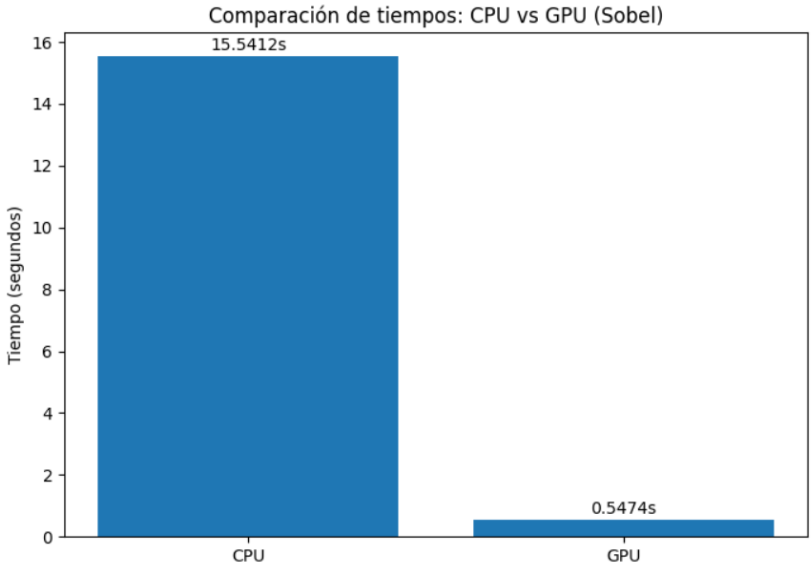


Figura 4: Comparación visual de tiempos de ejecución entre CPU y GPU.

# 6. Conclusiones

- El operador de Sobel permite detectar bordes de manera efectiva mediante el cálculo de gradientes locales en la imagen.
- La GPU acelera significativamente el proceso gracias a su capacidad para ejecutar miles de hilos en paralelo, lo que reduce de forma drástica el tiempo total de cómputo.
- El uso de paralelismo es especialmente beneficioso cuando el tamaño de la imagen aumenta, ya que el procesamiento por píxel es una tarea inherentemente paralelizable.

- El speedup de **28.39** obtenido demuestra la ventaja de trasladar algoritmos computacionalmente intensivos a la GPU.
- Este laboratorio permite comprender las diferencias fundamentales entre procesamiento secuencial y paralelo, así como la importancia de optimizar kernels para maximizar el rendimiento.