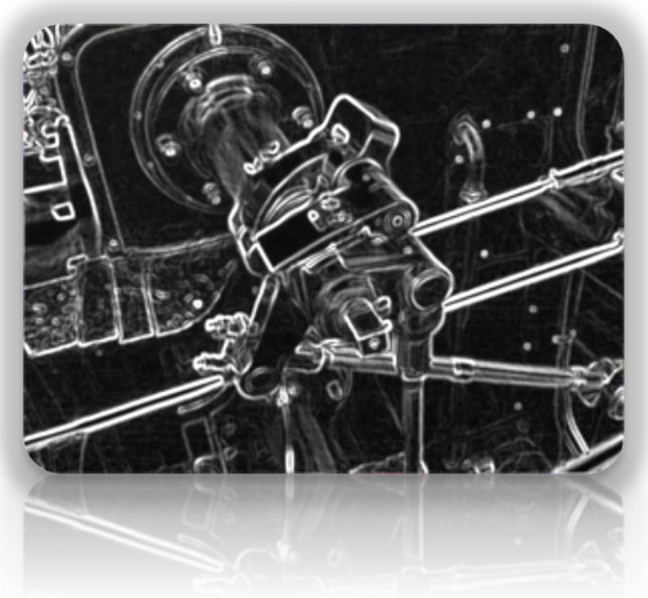
UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

CUDA Filtro Sobel



Enrique Rodríguez Villafranca

Asignatura: Computadores Avanzados
Titulación: Grado en Ingeniería Informática
Fecha: 22/12/2020

1. Compilación y ejecución

1) Generar archivos cmake y make: **cmake .**
2) Compilar: **make**
3) Ejecución del programa: **./Sobel_Filter** nombre_imagen.formato
   Importante: asegurarse de que la imagen que vamos a tratar se encuentra en la carpeta imput_images y que se trata de un archivo soportado por OpenCV (Ej: jpg, png).
4) Ejecución del programa con un filtro Gaussiano de kernel N: **./Sobel_Filter** nombre_imagen.formato N

2. Filtro Sobel: Qué es

Se trata de un mecanismo de detección de bordes, es decir, consiste encontrar las regiones en una imagen donde tenemos un cambio brusco de intensidad o un cambio brusco de color, un valor alto indica un cambio pronunciado y un valor bajo indica un cambio superficial.

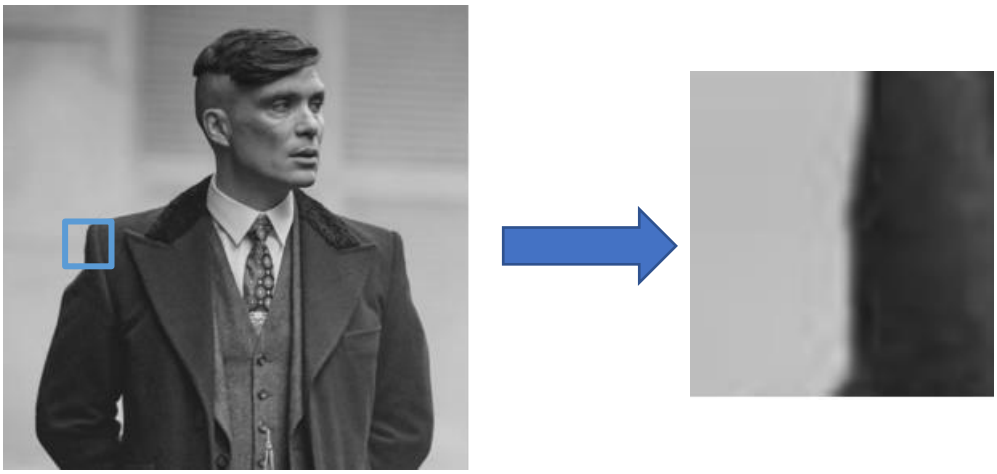
El operador Sobel distingue las direcciones vertical y horizontal, Y y X. Para ello, usamos una matriz de núcleo de 3 por 3, una para la dirección X a la que denominaremos **Gx** y otra para la dirección Y, denominada **Gy**.

3. Filtro Sobel: Cómo funciona

Como hemos definido anteriormente, el filtro Sobel se trata de un mecanismo de detección de bordes. Este tipo de mecanismos se basan en métodos matemáticos que tienen como objetivo identificar puntos en una imagen digital en los que el brillo de la imagen cambia drásticamente o, dicho de otro modo, tiene discontinuidades.

Tomemos como ejemplo esta imagen:

Como 1º paso, necesitamos pasar nuestra imagen original como **escala de grises** para reducir el coste computacional del cálculo, y poder trabajar sobre un solo canal, en el que 0 representaría el negro (luz mínima) y 255 el blanco (luz máxima).



Como podemos apreciar en la *Ilustración 1* e *Ilustración 2*, tenemos representado un borde, que consta una región clara (valor_pixel = 100) y otra oscura (valor_pixel = 50).

Básicamente, lo que estamos tratando de hacer aquí con el Operador Sobel es tratar de averiguar la cantidad de diferencia colocando la matriz de degradado sobre cada píxel de nuestra imagen.

Haremos esto para ambas direcciones, multiplicando los valores de la imagen original por los valores de estas dos matrices, este proceso se conoce como **convolución**, y como resultado obtendremos 2 valores, uno para la dirección horizontal (**Gx**) y otra para la vertical (**Gy**).

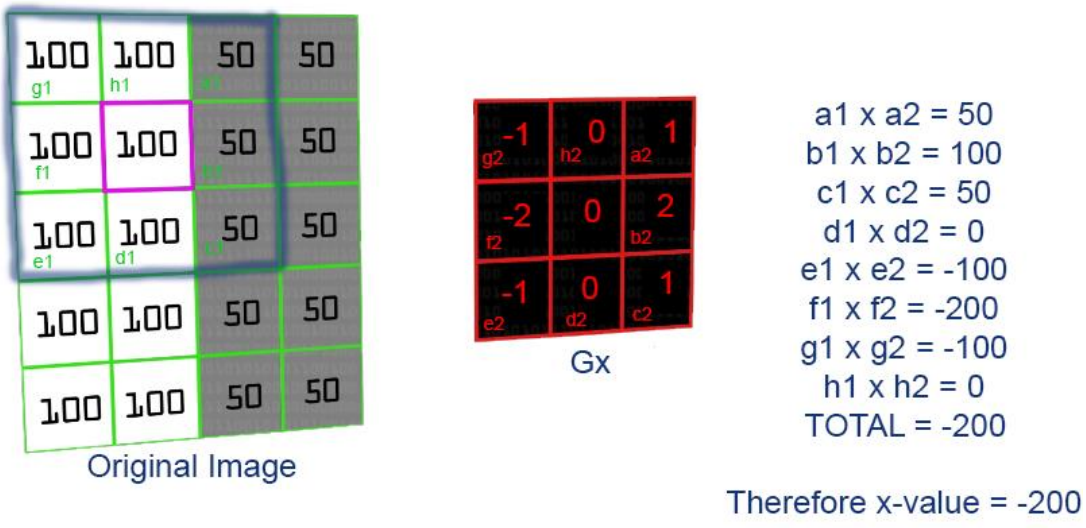


Ilustración 1 [fuente]

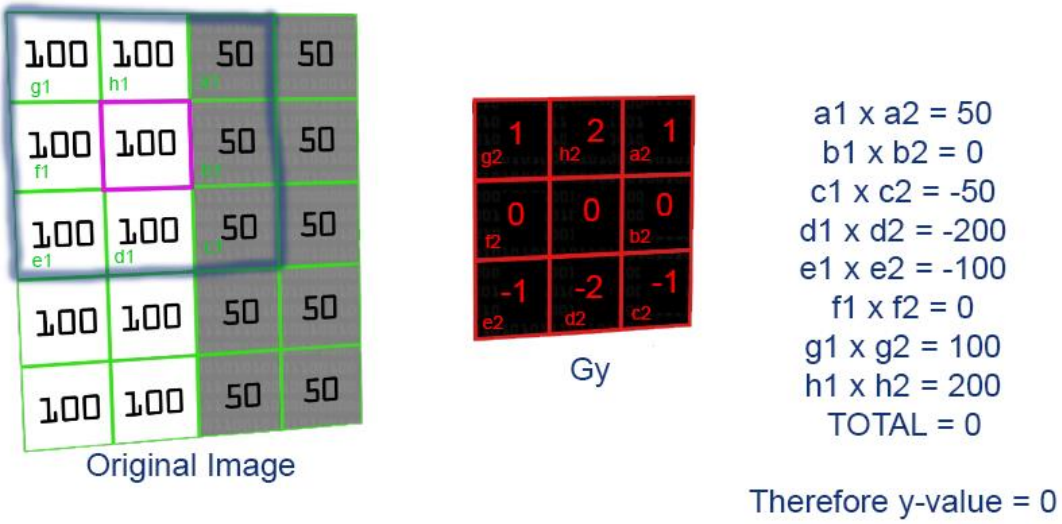
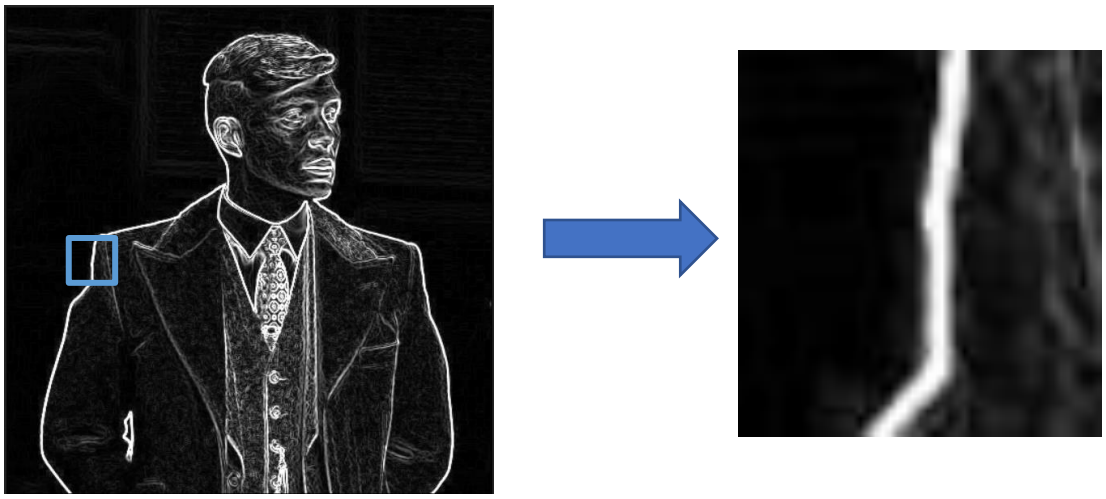


Ilustración 2 [fuente]

En cada punto de la imagen, los resultados de estas 2 aproximaciones de los gradientes horizontal y vertical pueden ser combinados para obtener la magnitud del gradiente, mediante la siguiente operación:

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Este es el resultado final:



Como podemos comprobar, hemos convertido los píxeles correspondientes al borde a color blanco, mientras que los que no formaban parte de un borde permanecen en negro.

Uso de filtros de reducción de ruidos:

Por último, para mejorar los resultados obtenidos, se ha incluido la opción de ejecución: /FilterImage [imagen.jpg] [Número_impar]

En donde Número_impar se corresponde con el tamaño del kernel Gaussiano a lanzar.

Esto se incluye dado que un recurso muy común para mejorar el filtro Sobel es el de utilizar otro mecanismo de filtrado de imagen como un filtro Gaussiano en este caso, que puede mejorar ampliamente los resultados en imágenes con mucho “ruido” o poca resolución.

Ejemplo:

Ilustración 3 Original

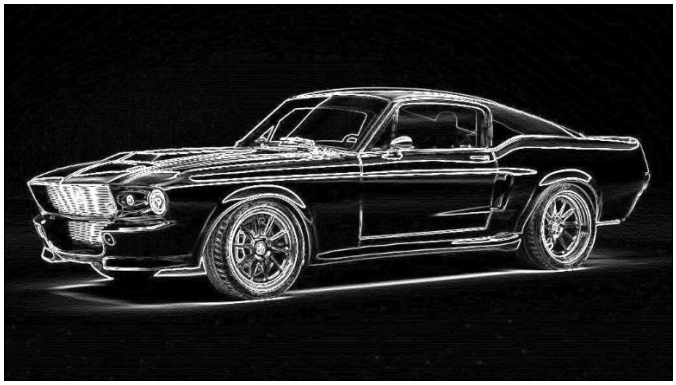


Ilustración 4 Sobel

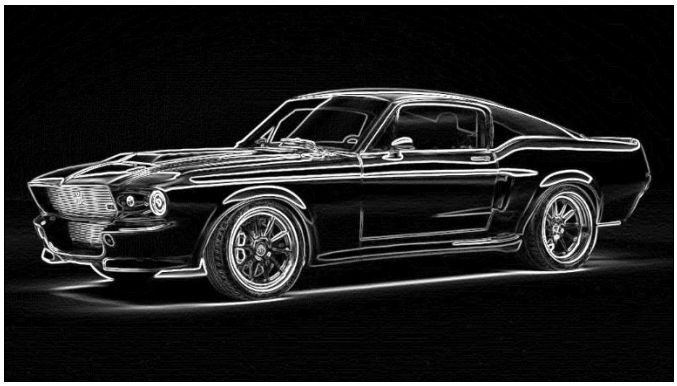


Ilustración 6 Sobel + Gauss (kernel 3x3)

Ilustración 5 Sobel + Gauss (kernel 5x5)

Como se puede apreciar, las imágenes a las que se ha aplicado Gauss producen un resultado más preciso y claro.

4. Filtro Sobel: Implementación en CUDA y OpenCV

Nuestro 1º paso es utilizar la función “imread(const String &filename, int flags=IMREAD_COLOR)” de OpenCV para cargar nuestra imagen en escala de grises (IMREAD_COLOR = IMREAD_GRAYSCALE).

```
Mat original_img = imread(image_path + argv[1],IMREAD_GRAYSCALE); //Lo pasamos en escala de grises para que solo exista un canal de color
if(original_img.empty()){
    printf("\033[1;31mError: Image not found\nPlease make sure it's on the \"input_images\" folder\033[0m \n");
    return 1;
}
```

Existe la opción de usar un filtro Gaussiano proporcionados por OpenCV para mejorar el resultado final de la imagen en caso de que exista demasiado ruido, en el caso contrario, se sigue con la ejecución normal del programa (sólo filtro Sobel mediante CUDA).

```
/** En caso de que pasemos un filtro Gaussiano para suavizar el ruido de la imagen y mejorar su resultado **/
Mat modified_img;
if (argc == 3){
    int gauss_size = atoi(argv[2]);

    if(gauss_size % 2 == 0){
        printf("\033[1;31mError: Gauss filter size must be an odd number (Ej: 3,5,7,etc)\033[0m \n"); //Impar, ya que el kernel debe ser simetrico
        return 1;
    }
    GaussianBlur(original_img,modified_img,Size(gauss_size,gauss_size),0);

/** Para la ejecucion normal del filtro Sobel **/
}else{
    modified_img = original_img;
}
```

A continuación, guardamos los datos que necesitamos de nuestra imagen (altura, anchura y tamaño) y reservamos el espacio en memoria necesario.

Posteriormente, copiamos los datos de la imagen a la memoria de la GPU, establecemos el tamaño y el número de bloques y ejecutamos el kernel de CUDA.

```
/** Copiamos la imagen a la memoria de la GPU **/
cudaMemcpy(src_img, modified_img.data, img_data_size, cudaMemcpyHostToDevice);

/** -> BLOQUE_HILOS = bloques de N x N hilos **/
dim3 threadsBlock(N_THREADS, N_THREADS, 1);

/** -> NUM_BLOQUES = calcular numero de bloques repartiendo el tamaño de la imagen entre los hilos de ejecución (ceil para redondear valores al alza) **/
dim3 numBlock(ceil(img_data_width/N_THREADS), ceil(img_data_height/N_THREADS), 1);

auto start_time = chrono::system_clock::now(); //Tiempo inicio

/** Ejecucion filtro Sobel mediante CUDA **/
filter_Sobel<<<numBlock, threadsBlock>>>(src_img, out_img, img_data_width, img_data_height); //Ejecucion del kernel CUDA
```

Tras la ejecución del kernel, copiamos los datos de la imagen de vuelta a la CPU y lo imprimimos mediante el uso de la función “imwrite” de OpenCV.

```
/** Copy data back to CPU from GPU **/
cudaMemcpy(modified_img.data, out_img, img_data_size, cudaMemcpyDeviceToHost);

/** Tamaño de nuestra iamgen **/
printf("\nProcessing %s: \033[1;34m%d\033[0m rows x \033[1;34m%d\033[0m columns\nTota

/** Imprimir imagen **/
imwrite( "output_image.png", modified_img );
```

Por último, liberamos el espacio anteriormente reservado mediante cudaFree().

Funcionamiento del Kernel de Sobel mediante CUDA:

```
__global__ void filter_Sobel(unsigned char* src_img,unsigned char* out_img, unsigned int width, unsigned int height) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int idy = threadIdx.y + blockIdx.y * blockDim.y;
    float Gx, Gy; //Kernel para las direcciones x e y
    float Gx_0_0,Gx_0_1,Gx_0_2,Gx_1_0,Gx_1_1,Gx_1_2,Gx_2_0,Gx_2_1,Gx_2_2,
          Gy_0_0,Gy_0_1,Gy_0_2,Gy_1_0,Gy_1_1,Gy_1_2,Gy_2_0,Gy_2_1,Gy_2_2;
    float G;

    /* Comprobar los limites de la imagen */
    if(idx > 0 && idy > 0 && idx < width-1 && idy < height-1) {

        /*********************************   Multiplicamos los valores de cada gradiente por la posición correspondiente (idx,idy) de la imagen original
            -1 0 +1             +1 +2 +1    Iteramos por los pixeles de la imagen mediante el uso de idx, idy y el ancho de la imagen
        Gx => -2 0 +2        Gy =>  0  0  0
            -1 0 +1             -1 -2 -1
        *********************************/

        Gx_0_0 = src_img[(idy-1)*width + (idx-1)];                    Gy_0_0 = src_img[(idy-1)*width + (idx-1)];
        Gx_0_1 = src_img[(idy-1)*width + (idx)];                      Gy_0_1 = src_img[(idy-1)*width + (idx)];
        Gx_0_2 = src_img[(idy-1)*width + (idx+1)];                    Gy_0_2 = src_img[(idy-1)*width + (idx+1)];
        Gx_1_0 = src_img[(idy)*width + (idx-1)];                      Gy_1_0 = src_img[(idy)*width + (idx-1)];
        Gx_1_1 = src_img[(idy)*width + (idx)];                        Gy_1_1 = src_img[(idy)*width + (idx)];
        Gx_1_2 = src_img[(idy)*width + (idx+1)];                      Gy_1_2 = src_img[(idy)*width + (idx+1)];
        Gx_2_0 = src_img[(idy+1)*width + (idx-1)];                    Gy_2_0 = src_img[(idy+1)*width + (idx-1)];
        Gx_2_1 = src_img[(idy+1)*width + (idx)];                      Gy_2_1 = src_img[(idy+1)*width + (idx)];
        Gx_2_2 = src_img[(idy+1)*width + (idx+1)];                    Gy_2_2 = src_img[(idy+1)*width + (idx+1)];

        Gx = (-1 * Gx_0_0) + (0 * Gx_0_1) + (1 * Gx_0_2) +
             (-2 * Gx_1_0) + (0 * Gx_1_1) + (2 * Gx_1_2) +
             (-1 * Gx_2_0) + (0 * Gx_2_1) + (1 * Gx_2_2);

        Gy = (1 * Gy_0_0) + (2 * Gy_0_1) + (1 * Gy_0_2) +
             (0 * Gy_1_0) + (0 * Gy_1_1) + (0 * Gy_1_2) +
             (-1 * Gy_2_0) + (-2 * Gy_2_1) + (-1 * Gy_2_2);

    /* El gradiente resultante (G) es la raiz cuadrada de (Gx^2 + Gy^2) */
    G = sqrt(pow(Gx,2) + pow(Gy,2));
    /* Modificamos el pixel que estamos comprobando */
    if(G > 255){
        out_img[idy*width + idx] = 255;     //En caso de que sobrepasemos el valor maximo posible para este pixel (255), ponemos este ultimo como su valor actual
    }else{
        out_img[idy*width + idx] = G;       //Cualquier otro caso -> valor correspondiente
    }
```

Construimos cada celda del gradiente correspondiente a las direcciones X e Y utilizando el ancho de la imagen y los índices (idx, idy). Para ello, usamos el índice de hebra, que va avanzando, dependiendo del bloque (y la dimensión) en que se encuentre. Es decir, primero se recorre usando todas las hebras del bloque 0, a continuación, todas las del bloque 1, etc.

Finalmente, multiplicamos cada una por el valor que le corresponde (-2, -1, 0, +1 o+2). Después realizamos el sumatorio para obtener los valores de Gx y Gy y realizamos la raíz de la suma cuadrática para obtener el valor final de este píxel.

5. Filtro Sobel: Comparación tiempo de ejecución

Usando una CPU: **Ryzen 7 4800H** de 16 hilos y GPU: **GeForce RTX 2060**, procesando la siguiente imagen de 5864 x 3904 pixeles:

Tiempo de ejecución con **CPU** = 79.2 milisegundos
Tiempo de ejecución con **OpenMP** = 19.9 milisegundos

Tiempo de ejecución con **CUDA** = 1.6 milisegundos

Speedup de **GPU** con respecto a la **CPU** = x48.1

6. Bibliografía

Funcionamiento de Sobel:

- ➔ Fotos, fuente de las imágenes usadas para explicar el funcionamiento
- ➔ Vídeo explicativo, usado como fuente para comprender el funcionamiento de forma sencilla

Código usado como referencia:

- ➔ Tiempo de ejecución comparativo e información de GPU, así como la estructura básica
- ➔ OpenCV, tratamiento de imágenes con OpenCV