Universidad de Castilla - La Mancha

Computadores Avanzados

Escuela Superior de Informática

Operador Sobel en CUDA

Autor:
Álvaro Cerdá Pulla

Profesor:
María José Santofimia Romero

18 de Enero de 2021

Índice

1. Operador Sobel

El Operador Sobel o también llamado Operador Sobel-Feldman, que se utiliza en el campo de procesamiento de imágenes y visión por computador, más concretamente, para la detección de bordes. Técnicamente, es un operador de diferenciación discreta , que calcula una aproximación del gradiente de la función de intensidad de imagen. En cada punto de la imagen, el resultado del operador de Sobel-Feldman es el vector de gradiente correspondiente o la norma de este vector. Matemáticamente, el gradiente de una función de dos variables (en este caso, la función de intensidad de la imagen) para cada punto es un vector bidimensional cuyos componentes están dados por las primeras derivadas de las direcciones verticales y horizontales. Para cada punto de la imagen, el vector gradiente apunta en dirección del incremento máximo posible de la intensidad, y la magnitud del vector gradiente corresponde a la cantidad de cambio de la intensidad en esa dirección.

En resumen, el Operador Sobel, calcula el gradiente de la intensidad de una imagen en cada punto o píxel. Así, para cada píxel, este operador da la magnitud del mayor cambio posible, la dirección de este y el sentido desde oscuro a claro.

1.1. Gradiente

Hemos hablado del gradiente, pero, ¿qué es el gradiente?. El **gradiente** de una imagen mide cómo esta cambia en términos de color o intensidad. La magnitud del gradiente nos indica la rapidez con la que la imagen está cambiando, mientras que la dirección del gradiente nos indica la dirección en la que la imagen está cambiando más rápidamente.

En el caso del operador Sobel, no es más que una **aproximación**, más o menos precisa, para calcular el gradiente de una imagen.Se define a través de kernels cuadrados. El operador Sobel aplica un alisamiento Gausssiano común y estima las derivadas parciales a lo largo de X e Y. Utilizando los argumentos yorder y xorder, se puede especificar la dirección de las derivadas a tomar, vertical u horizontal, respectivamente. También se puede especificar el tamaño del kernel utilizando el argumento ksize. Cuando ksize = -1, se utiliza el kernel Scharr 3×3 que da mejores resultados que el kernel Sobel 3×3.

1.2. Formulación

Matemáticamente, el operador utiliza dos kernels de 3×3 elementos para aplicar convolución a la imagen original para calcular aproximaciones a las derivadas, un kernel para los cambios horizontales y otro para las verticales. Si definimos A como la imagen original, el resultado, que son las dos imágenes G_x y G_y que representan para cada punto las aproximaciones horizontal y vertical de las derivadas de intensidades, es calculado como:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

Dado que los núcleos de Sobel se pueden descomponer como los productos de un núcleo de diferenciación y promediado, calculan el gradiente con suavizado. Por ejemplo, se puede escribir como:

$$G_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * (\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * A) \tag{1}$$

$$G_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * (\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * A) \tag{2}$$

En cada punto de la imagen, los resultados de las aproximaciones de los gradientes horizontal y vertical pueden ser combinados para obtener la magnitud del gradiente, mediante el teorema de Pitágoras:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3}$$

Con esta información, podemos calcular también la dirección del gradiente:

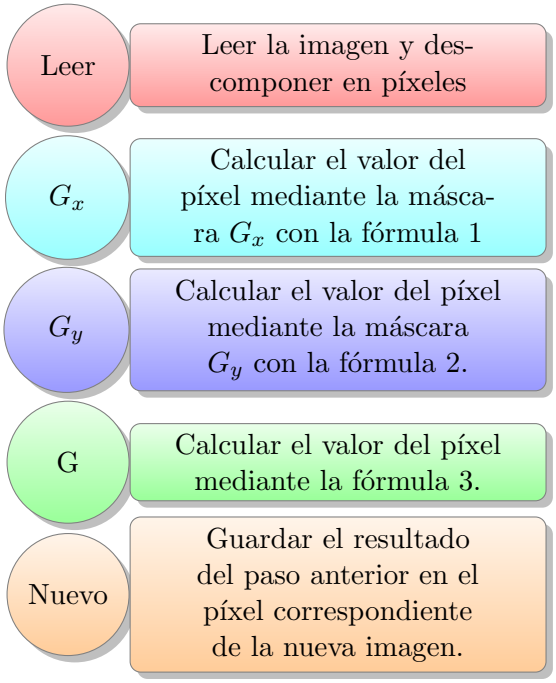
$$\theta = arctan(G_x/G_y)$$

2. Implementación en el código

2.1. Idea Principal

La idea general es almacenar los datos de una imagen que introducimos mediante línea de comandos, almacenar sus datos de forma que luego los podamos recuperar por píxeles, de manera sencilla y ordenada. Una conseguido esto, aplicamos el Operador Sobel y posteriormente guardamos la imagen de forma que se distinga de la original.

Una vez tenemos esa idea general,podemos realizar un diagrama para saber que debemos hacer en cada momento y que **pasos** tendremos que realizar en orden secuencial.



2.2. Kernel

Como hemos visto anteriormente, en las ecuaciones 1 y 2, se puede calcular el gradiente de X e Y. En el código, veremos como la multiplicación de la matriz y el vector ya está realizada en el kernel.

```
1  __global__ void filtroSobelGPU(unsigned char* srcImg, unsigned char* dstImg, const unsigned int
       ancho, const unsigned int alto){
2      int x = threadIdx.x + blockIdx.x * blockDim.x;
3      int y = threadIdx.y + blockIdx.y * blockDim.y;
4      if( x > 0 && y > 0 && x < ancho-1 && y < alto-1) {
5          float dx = (-1* srcImg[(y-1)*ancho + (x-1)]) + (-2*srcImg[y*ancho+(x-1)]) +
               (-1*srcImg[(y+1)*ancho+(x-1)]) +
6              (     srcImg[(y-1)*ancho + (x+1)]) + ( 2*srcImg[y*ancho+(x+1)]) + (
                   srcImg[(y+1)*ancho+(x+1)]);
7          float dy = (     srcImg[(y-1)*ancho + (x-1)]) + ( 2*srcImg[(y-1)*ancho+x]) + (
               srcImg[(y-1)*ancho+(x+1)]) +
8              (-1* srcImg[(y+1)*ancho + (x-1)]) + (-2*srcImg[(y+1)*ancho+x]) +
                   (-1*srcImg[(y+1)*ancho+(x+1)]);
9          dstImg[y*ancho + x] = sqrt( (dx*dx) + (dy*dy) ) > 255 ? 255 : sqrt( (dx*dx) + (dy*dy) );
10     }
11 }
```

Calculamos las aproximaciones de las derivadas dx, dy, y una vez que las tenemos, entonces, hacemos uso de la fórmula 3, el teorema de Pitágoras.

Cada llamada, se va guardando en el píxel correspodiente de la nueva imagen, por lo que al realizar todos los píxeles, la imagen estará lista, por lo que pasaremos a copiar los datos que hemos obtenido de las operaciones realizadas en el kernel (operaciones con las máscaras), a una imagen nueva, que veremos en el uso de cv2.

Los valores que tendremos para el kernel como entrada, serán la imagen original, una imagen destino (la nueva después de aplicar el operador sobel), la columna y la fila.

Se hará este proceso con todos los píxeles de la imagen, una vez que recorra todos, entonces ya tendremos en nuestra variable **dstImg** el resultado de la imagen después de aplicar el Operador Sobel.

Figura 1: Imagen original, avión aterrizando

Figura 2: Filtro Sobel, avión aterrizando

Figura 3: Imagen original, New York

Figura 4: Filtro Sobel, New York

2.3. Uso de CV2

Para cargar, escribir y guardar la imagen hacemos uso de la librería cv2. Para ello debemos instalar la librería de opencv.

```
$ sudo apt-get install libopencv-dev
```

Una vez la tengamos instalada, debemos colocar de forma correcta los includes. Buscaremos con el **find** desde la carpeta raíz, los módulos que necesitemos. Una vez los encontremos, copiaremos la dirección donde se encuentran. En mi caso sería como vemos a continuación.

```
#include </usr/include/opencv4/opencv2/opencv.hpp>
#include </usr/include/opencv4/opencv2/core.hpp>
#include </usr/include/opencv4/opencv2/opencv_modules.hpp>
#include </usr/include/opencv4/opencv2/core/mat.hpp>
```

Cuando lleguemos a este punto estaremos listos para usarlo.

Para cargar la imagen, tenemos este método, que carga los datos de la imagen en una variable de tipo Mat.

```
// Mtodo para cargar la imagen.
cv::Mat loadImage(char image_name[]){
    using namespace cv;

    Mat image = imread(image_name, 255);
    cv::cvtColor(image, image, cv::COLOR_RGB2GRAY);

    if(image.empty()) {
        std::cout << ANSI_COLOR_RED "Error: La imagen no se ha cargado correctamente"
            ANSI_COLOR_RESET<< std::endl;
    }
    return image;
}
int main(int argc, char *argv[]){

// Llamada del main al mtodo donde cargaremos la imagen
    Mat image;
    try{
    image = loadImage(argv[2]);
    }catch(Exception e){
        std::cout << ANSI_COLOR_RED "Error: El nombre de imagen que esta seleccionando no
            existe" ANSI_COLOR_RESET<< std::endl;
        return -1;
    }
}
```

Una vez consigamos cargar la imagen, tendremos dentro de la variable los datos de cada píxel. Gracias a esta libería podremos obtener de forma sencilla los valores de la imagen tanto por filas, como por columnas, lo que usaremos en el kernel para operar con las máscaras correspondientes G_x y G_y.

Una vez hayamos operado en el kernel y tengamos los datos listos, debemos escribir en la nueva imagen. He creado una nueva, que se llame de la misma forma que la original, añadiendo la extensión **_sobel.png** donde copiaremos los datos.

```
int main(int argc, char *argv[]){
    char new_image[50];
    strcpy(new_image, argv[2]);
    strcat(new_image, "_sobel.png");
    cv::imwrite(new_image,image);
    std::cout << ANSI_COLOR_MAGENTA "Imagen Sobel: = " ANSI_COLOR_RESET <<
        new_image<<std::endl;
    return 0;
}
```

2.4. Control de errores

Para **controlar los errores**, hemos recopilado conocimientos anteriores, por lo que si el resultado de una función de CUDA es distinta de cudaSuccess, significará que no se está completando de forma correcta, por lo que debemos lanzar una excepción que contendrá un mensaje de aviso, y además, abortará el programa.

```
1  // Comprobar errores de CUDA
2  cudaError_t testCuErr(cudaError_t result)
3  {
4    if (result != cudaSuccess) {
5      std::cout << ANSI_COLOR_RED "Cuda error: %s." ANSI_COLOR_RESET <<
           result<<std::endl;
6      assert(result == cudaSuccess); // si no se cumple, se aborta el programa
7    }
8    return result;
9  }
10
11
12 int main(int argc, char *argv[]){
13
14 // Ejemplos de donde usaremos testCuErr
15     testCuErr(cudaMalloc( (void**)&gpu_src, (image.cols * image.rows)));
16     testCuErr(cudaMalloc( (void**)&gpu_sobel, (image.cols * image.rows)));
17     testCuErr(cudaMemcpy(gpu_src, image.data, (image.cols*image.rows),
           cudaMemcpyHostToDevice));
18     testCuErr(cudaMemset(gpu_sobel, 0, (image.cols*image.rows)));
19
20 }
```

3. Manual de Usuario

Para realizar más sencilla la utilización de esta herramienta, he creado un Makefile, con las siguientes opciones.

Para su uso, deberemos realizar lo siguiente en la terminal, donde *opción*, corresponde a una de las alternativas que se dan a continuación.

```
$ make opcion
```

3.1. Opciones de comando

- **compile**: Sirve para compilar teniendo en cuenta las librerías y los flags necesarios.
- **run**: Sirve para ejecutar el archivo creado después de la compilación, al que se le da un número de hilos y el nombre de la imagen. Ambos datos se pueden modificar desde el propio Makefile sin necesidad de volver a compilar.
- **cleanPhoto**: Eliminar cualquier imagen que tenga la extensión creada por esta herramienta, es decir, la extensión *_sobel.png*.
- **cleanExe**: Elimina los ejecutables que coincidan con el nombre *filtro* como el ejecutable que se crea en el compile.
- **clean**: Es la unión de estos dos últimos, elimina tanto el ejecutable como las imágenes creadas.

Para acceder al **código completo**, busque en la siguiente dirección:

`https://github.com/alvaroc20/CA_FiltroImagen`

4. Bibliografía

- La librería de visión artificial OPENCV aplicación a la docencia e investigación. V.M.Arévalo, J. González, G. Ambrosio
- https://github.com/lukas783/CUDA-Sobel-Filter
- https://github.com/Thyix/sobel-filter-cuda/tree/master/CudaLab1
- https://github.com/LevidRodriguez/Sobel_with_OpenCV-CUDA
- https://www.micc.unifi.it/bertini/download/parallel/2016-2017/11_gpu_cuda_2.pdf
- https://calcul.math.cnrs.fr/attachments/spip/IMG/pdf/cours_gpu_1.pdf