Paralelización de algoritmo de detección de Bordes

[Mediante filtro de sobel]

Por: Aaron hoffman, Gonzalo Garaban y Jaime Pérez

2019

Índice

[Introducción 3](#_Toc24391726)

[Estado del Arte 5](#_Toc24391727)

[Paralelización 6](#_Toc24391728)

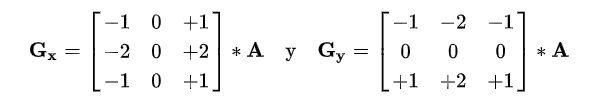
[Pseudocódigo 6](#_Toc24391729)

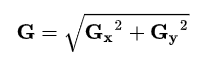
# Introducción

En este trabajo propondremos un solución para paralelizar un algoritmo de detección de bordes. En concreto, utilizaremos el método de Sobel. Este convierte una imagen en blanco y negro, la transforma en una matriz para posteriormente aplicarle unas serie de transformaciones para resaltar sus bordes. En nuestro caso, la imagen será convertida a blanco y negro con una librería externa, ya que es algo que se sale del objetivo de la práctica.

El algoritmo de Sobel funciona aplicando dos filtros separados, que en este área llamamos máscaras. La primera busca bordes verticales y la segunda bordes verticales. Esta primera da como resultado el conocido como gradiente Y, una imagen en escala de grises donde los bordes mas marcados aparecen en negro (255), y los menos van bajando de intensidad hasta que las zonas no detectadas como borde que aparecen en blanco.

De la misma forma la segunda mascara obtiene el gradiente Y con el mismo formato que el Y. En el siguiente ejemplo detalla las operaciones matemáticas que se aplicarían sobre la matriz obtenida a través de una imagen, ‘’A’’, para obtener ambos gradientes Gx y Gy:



Una vez obtenemos gradientes los combinamos para poder generar una foto con los bordes horizontales y verticales, para ello de suman uno a uno los cuadrados de los elementos de las matrices y luego se calcula su raíz. Para la siguiente imagen representa el cálculo:

Una vez obtenemos esta matriz solo queda reconvertirla a imagen. El siguiente cuadro muestra las diferentes etapas de este proceso este proceso:

|  |  |
| --- | --- |
| Imagen | Proceso |
|  | El primer paso es pasar la imagen a blanco y negro |
|  | A continuación, aplicamos el gradiente X |
|  | Luego aplicamos el gradiente Y |
|  | Y por último combinamos las imágenes |

# Estado del Arte

Conmputer visión (CV) es un campo que esta empezando a ver un incremento en su desarrollo en casos prácticos en los últimos 15 años. La popularización de redes neuronales y su implementación a este campo ha expandido enormemente las posibilidades de este campo. No obstante, el tratamiento de imágenes en vivo acarrea un gran problema debido a las cada vez más estrictas leyes protección de datos.

Debido a que las imágenes no pueden ser almacenadas por parte de la compañía para su posterior tratamiento, en muchos casos necesario realizar todas las operaciones antes de almacenar las imágenes. Esto implica que si queremos realizar CV sobre la imágenes de una cámara, solo podremos calcular tantos FPS (fotogramas por segundo) a la calidad que deseemos, como podamos procesar.

Como solución a este problema proponemos paralelizar la aplicación de las mascaras sobre las imágenes, lo cual es una operación muy realizada en este ámbito. Hemos elegido el filtro de Sobel, porque se trata de una convolución (operación que modifica el valor de los pixeles de una imagen) con una dependencia de datos, ya que para obtener el resultado final es necesario combinar los datos de dos cálculos previos como previamente indicado. Pero este tipo de optimización es aplicable a muchas otras operaciones como máscaras de bluring (como la gaussiana) o incluso otros métodos de detección de líneas (como Robert Cross).

Otra de la razones por las que hemos elegido la máscara de Sobel, es porque es muy utilizada para utilizar filtros de Canny en CV. Esta combinación de técnicas es uno, sino el que más, de los algoritmos más utilizado en CV.

# Paralelización

El siguiente esquema representa nuestra propuesta de paralelización del algoritmo entendiendo los diferentes cuadrados.

# Pseudocódigo

def *funSerie*(img):

    mask1 = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]

    mask2 = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]

    img\_temp = copy(img)

    res = copy(img)

*for* row in range(1, img.height - 1):

*for* element in range(1, img.width - 1):

            temp\_value = img\_temp[row][element] \* mask1[0][0] + img\_temp[row + 1][element + 1] \* \

                mask[1][1] + img\_temp[row][element + 1] \* mask1[0][1] + \

                img\_temp[row + 1][element] \* mask1[1][0]

            res[row][element] = temp\_value

*for* row in range(1, img.height - 1):

*for* element in range(1, img.width - 1):

            temp\_value = img\_temp[row][element] \* mask2[0][0] + img\_temp[row + 1][element + 1] \* \

                mask2[1][1] + img\_temp[row][element + 1] \* mask2[0][1] + \

                img\_temp[row + 1][element] \* mask2[1][0]

            res[row][element] += temp\_value \*\* 2

            res[row][element] = sqrt(res[row][element])

*return* res

def *funParalelo*(img):

    masks = [[[-1, 0, 1],

              [-2, 0, 2],

              [-1, 0, 1]],

             [[-1, -2, -1],

              [0, 0, 0],

              [1, 2, 1]]]

    local\_data\_masks = []

    img\_tmp = [img.width][img.height]

    Scatter(masks, 1, local\_data\_masks)

    Scatter(tmp\_img, 1, \*tmp\_row)

*for* i in range(0, img.width):

        temp\_value = (tmp\_img[counter-1][j-1] \* local\_data\_masks[0][0]) + (tmp\_img[counter-1][j] \* local\_data\_masks[0][1]) +

        (tmp\_img[counter-1][j+1] \* local\_data\_masks[0][2]) + (tmp\_img[counter][j-1] \* local\_data\_masks[1][0]) +

        (tmp\_img[counter][j] \* local\_data\_masks[1][1]) + (tmp\_img[counter][j+1] \* local\_data\_masks[1][2]) +

        (tmp\_img[counter+1][j-1] \* local\_data\_masks[2][0]) + (tmp\_img[counter+1][j] \* local\_data\_masks[2][1]) +

        (tmp\_img[counter+1][j+1] \* local\_data\_masks[2][2])

        tmp\_row[i] += temp\_value \*\* 2

*if* semaforo:

        tmp\_row = [sqrt(x) *for* x in tmp\_row]

    Gather()

    Gather()

*return* img\_tmp

# Algoritmo en Serie

## Instrucciones de ejecución

Para compilar el programa, es necesario tener instalada la librería de OpenCV4. Una vez hecho eso, en una terminal Unix ejecutamos el comando: *mpic++ $(pkg-config --cflags --libs opencv4) -std=c++17 serie.cpp -o serie*

De esta manera tenemos el archivo compilado. Una vez hecho eso, para ejecutarlo hacemos: *./serie <dirección relativa de la imagen>* En caso de que no pongamos una dirección relativa, se cogerá la imagen por defecto.

## Descripción

Para poder medir la eficacia y el tiempo que se tarda en ejecutar el algoritmo en cada caso, utilizaremos imágenes grandes (de 4k), para que cualquier diferencia en el speedup sea notable. Dentro del algoritmo en serie podemos notar diferentes cosas que pueden llamar la atención. Primero, podemos ver que cada resultado de multiplicar la matriz por los pixeles de la imagen es posteriormente dividido por 30. Este número ha sido escogido por ser el recomendado para eliminar ruido de la imagen. De hecho, si se elimina, Sobel detecta absolutamente todos los bordes de la imagen, lo cual no es lo deseable. Lo que queremos detectar son solamente los bordes que resultan mas o menos evidentes y por eso está ahí ese valor.

Tras aplicar las máscaras, podemos ver que hay otra función que recorre todos los pixeles de la imagen. En ella, lo que hacemos es juntar los resultados de las operaciones anteriores y aplicar otro descarte de bordes. En este caso, los bordes que sean demasiado ligeros, vuelven a ser descartados quedándonos así con los bordes más evidentes. El tiempo que tarda nuestro algoritmo en ejecutarse sobre la imagen que le proporcionamos es de**: 5069149 microsegunods**, o lo que es lo mismo; **5 segundos**. El resultado del proceso:

|  |  |
| --- | --- |
| Imagen Original | A close up of a busy city street in front of a building  Description automatically generated |
| Imagen tras la ejecución | A black and white photo of a city  Description automatically generated |

# Objetivo al paralelizar

Viendo el algoritmo en serie, queda claro qué es lo que más tiempo toma: Recorrer todos los pixeles de la imagen 3 veces, lo que implica una complejidad de O^2. Creemos que podemos reducir esta cantidad a drásticamente, aunque es poco probable que podamos reducir la complejidad computacional final, ya que el proceso de juntar las operaciones probablemente vaya a estar siempre presente y tendremos que recorrer la imagen en algún momento.