



O A X A C A

UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

Detección de bordes en imágenes aéreas con el filtro de Sobel y CUDA

REPORTE DE IMPLEMENTACIÓN

Para el curso de:

Tópicos Selectos De Programación

Presenta:

M.R. ARMANDO LEVID RODRÍGUEZ SANTIAGO

Profesor:

DR. ARTURO TÉLLEZ VELÁZQUEZ

Huajuapan de León, Oaxaca, México, Junio de 2019

Índice

1. Introducción	1
1.1. Planteamiento del Problema	1
1.2. Hipótesis	1
1.3. Objetivos	2
1.3.1. Objetivo General	2
1.3.2. Objetivos Específicos	2
2. Marco Teórico	2
2.1. Extracción de bordes	2
2.1.1. Filtro de Sobel	2
2.2. Cómputo Heterogéneo con CUDA	4
2.2.1. Compute Unified Device Architecture (CUDA)	4
2.2.2. El Kernel	5
3. Implementación	8
3.1. Algoritmo secuencial	8
3.2. Algoritmo paralelizado	8
3.3. Filtro de Sobel en OpenCV	10
3.4. Resultados	13
4. Conclusiones	14
Referencias	15
A. Código en C++ para el filtro de Sobel.	17

1. Introducción

El uso de vehículos aéreos no tripulados UAV (del inglés: Unmanned Aerial Vehicle) mejor conocidos como drones ha ayudado en la solución de diversos problemas. Sin embargo, también se han generado nuevos retos para la investigación de manera que puedan realizar nuevas aplicaciones y se resolver algunas otras. Trabajos como [Zhao and Nevatia, 2003, Li et al., 2014, Bu et al., 2016, Chen et al., 2017] demuestran que la aplicación de los UAV requiere de diversos algoritmos que nos permitan resolver la tarea planteada, para lo cual muchas veces es necesario la extracción de características de las información visual que el UAV proporciona. Por lo que existen diversos algoritmos utilizados para realizar el procesamiento de la información [Venkateswar and Chellappa, 1992, Bowyer et al., 2001, Hu et al., 2007, Gleason et al., 2011, Teng et al., 2019]. Uno de los más populares es el algoritmo de Sobel [Vincent et al., 2009, Gupta and Mazumdar, 2013, Hu et al., 2018] el cual es usado en diversas aplicaciones. Sin embargo a pesar de ser un algoritmo sencillo de implementar su aplicación requiere un alto costo computacional ya que implica realizar operaciones a todos y cada uno de los pixeles de la imagen. Afortunadamente hoy en día se ha desarrollado herramientas de hardware y nuevos paradigmas de programación que pueden realizar esta y muchas otras operaciones de forma rápida y eficiente, este es el caso de las tarjetas gráficas o GPUs y la programación heterogénea con CUDA.

En este trabajo se ha implementado el filtro de Sobel para la detección de bordes en imágenes aéreas de alta resolución. El algoritmo se ha implementado de tres diferentes formas secuencial, heterogénea y como las funciones de OpenCV. Para poder realizar una comparativa de los tiempos de ejecución y determinados si la programación heterogénea ofrece ventajas en cuanto a la mejora de los tiempos de ejecución del filtro de Sobel.

1.1. Planteamiento del Problema

El desarrollo de sistemas que permitan la segmentación, detección o extracción de características de información visual requiere de la implementación de técnicas que permitan la extracción de la información buscada. Un algoritmo suficientemente valido en este tipo de procesos es el filtro de Sobel el cual, además de extraer bordes también suaviza la imagen eliminando el ruido presente mediante la aplicación de máscaras de convolución. Sin embargo este es un proceso computacional sumamente costoso en tiempo de ejecución, especialmente cuando se trabaja con imágenes de alta resolución ya que las máscaras son aplicadas a todos los pixeles, tanto en dirección x como en y .

Por lo tanto, se propone implementar el filtro de Sobel de forma heterogénea de manera que sea posible la reducción del tiempo computacional consumido por el algoritmo para la extracción de bordes en una imagen.

1.2. Hipótesis

Es posible minimizar el tiempo de procesamiento del algoritmo del filtro de Sobel mediante cómputo heterogéneo.

1.3. Objetivos

1.3.1. Objetivo General

Implementar el filtro de Sobel de forma secuencial y en computo heterogéneo con CUDA C/C++ para comparar los tiempos de ejecución correspondientes a cada implementación.

1.3.2. Objetivos Específicos

1. Implementar en C++ el algoritmo discreto del filtro de Sobel de forma secuencial.
2. Implementar en CUDA C/C++ el algoritmo discreto del filtro de Sobel de forma paralela.
3. Comparar el rendimiento del algoritmo secuencial y paralelo del filtro de Sobel.
4. Determinar el factor de mejora entre los algoritmos secuencial y paralelo del filtro de Sobel.

2. Marco Teórico

2.1. Extracción de bordes

Los bordes de una imagen digital se pueden definir como transiciones entre dos regiones de niveles de gris significativamente distintos. Suministran una valiosa información sobre las fronteras de los objetos y puede ser utilizada para segmentar la imagen, reconocer objetos, etc. La detección de bordes en general es un elemento importante para la visión artificial, hoy en día existen muchas aplicaciones en robótica móvil, procesamiento de imágenes y reconocimiento de patrones, por mencionar algunas [Patnaik and Yang, 2012, Vincent et al., 2009, Alegre et al., 2016]

La mayoría de las técnicas para detectar bordes emplean operadores locales basados en distintas aproximaciones discretas de la primera y segunda derivada de los niveles de grises de la imagen. Este es el caso del operador o filtro de Sobel.

2.1.1. Filtro de Sobel

El operador Sobel es utilizado en el procesamiento digital de imágenes, especialmente cuando se desea realizar la detección de bordes. Técnicamente es un operador diferencial discreto que calcula una aproximación al gradiente de la función de intensidad de una imagen para cada punto de la imagen a procesar, el resultado es tanto el vector gradiente correspondiente como la norma de éste vector.

El gradiente de una imagen en cualquier punto se define como un vector bidimensional dado por la siguiente ecuación:

$$G[f(x, y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} f(x, y) \\ \frac{\partial}{\partial y} f(x, y) \end{bmatrix} \quad (1)$$

Donde el vector G apunta en la dirección de variación máxima de f en el punto (x, y) por unidad de distancia con la magnitud y dirección dadas por:

$$|G| = \sqrt{G_x^2 + G_y^2}; \quad \Phi = \tan^{-1} \frac{G_y}{G_x} \quad (2)$$

Una práctica habitual es aproximar la magnitud del gradiente con valores absolutos

$$|G| \approx |G_x| + |G_y| \quad (3)$$

Cualquiera de las ecuaciones (2) y (3) se puede utilizar para obtener la magnitud del gradiente, ya que va en relación a un determinado umbral, si el valor de la magnitud supera el umbral este punto se considera como un borde. Para calcular la derivada en (1) se pueden utilizar las diferencias de primer orden entre dos pixeles adyacentes, estos es:

$$|G_x| = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}, \quad |G_y| = \frac{f(y + \Delta y) - f(y - \Delta y)}{2\Delta y}, \quad (4)$$

El problema de obtener una magnitud de gradiente absoluta apropiada para los bordes, radica en el método utilizado. Los operadores de Sobel al igual que los operadores de gradiente tienen la tarea de suavizar la imagen de tal manera que se elimina un poco de ruido de la imagen si es que lo tiene, por lo consiguiente se puede desaparecer falsos bordes. Para dicha tarea existen 2 máscaras de 3×3 (ver Tabla 1), una para el gradiente horizontal G_x y una para el gradiente vertical G_y , que se utilizan como operadores de Sobel. A partir de eso las derivadas basadas en los operadores de Sobel son los mostrados en (5):

Tabla 1: Máscaras de Sobel

Región de la imagen 3×3	Máscara usada para G_x	Máscara usada para G_y
$\begin{bmatrix} z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 \\ z_7 & z_8 & z_9 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

$$\begin{aligned} G_x &= (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \\ G_y &= (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \end{aligned} \quad (5)$$

Para obtener los valores del gradiente de la imagen se utilizan las expresiones (5) y la magnitud se puede obtener con la expresión (2) o (3). Con estas expresiones obtenemos el valor del gradiente en dichos pixeles, para obtener el siguiente valor, las máscaras se mueven al pixel siguiente, es decir, la nueva posición. Una vez que se ha obtenido la magnitud del gradiente, se puede decir si un pixel es un borde o no, con la expresión (6), obteniendo una imagen binaria en la salida.

$$g(x, y) = \begin{cases} 1 & \text{si } G[f(x, y)] > T \\ 0 & \text{si } G[f(x, y)] \leq T \end{cases} \quad (6)$$

2.2. Cómputo Heterogéneo con CUDA

El modelo de computación con tarjetas gráficas consiste en usar conjuntamente una CPU (Central Processing Unit) y una GPU (Graphics Processing Unit) de manera que formen un modelo de computación heterogéneo (Figura 1). Siguiendo este modelo, la parte secuencial de una aplicación se ejecutaría sobre la CPU (comúnmente denominada host) y la parte más costosa del cálculo se ejecutaría sobre la GPU (que se denomina device). Desde el punto de vista del usuario, la aplicación simplemente se ejecutará más rápido porque está utilizando las altas prestaciones de la GPU para incrementar el rendimiento.

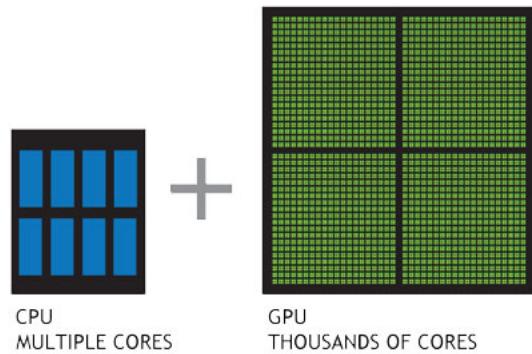


Figura 1: Modelo de cómputo heterogéneo

El problema inicial del uso de las tarjetas gráficas para el cálculo científico de propósito general (en inglés GPGPU - General-Purpose Computing on Graphics Processing Units) era que se necesitaba usar lenguajes de programación específicos para gráficos como el OpenGL o el Cg para programar la GPU. NVIDIA fue consciente del potencial que suponía acercar este enorme rendimiento a la comunidad científica y decidió investigar la forma de modificar la arquitectura de sus GPUs para que fueran completamente programables para aplicaciones científicas además de añadir soporte para lenguajes de alto nivel como C y C++. De este modo NVIDIA introdujo a sus tarjetas gráficas la arquitectura CUDA (Compute Unified Device Architecture), una nueva arquitectura para el cálculo paralelo de propósito general, con un nuevo repertorio de instrucciones y un nuevo modelo de programación paralela, con soporte para lenguajes de alto nivel y constituidas por cientos de núcleos que pueden procesar de manera concurrente miles de hilos de ejecución. En esta arquitectura, cada núcleo tiene ciertos recursos compartidos, incluyendo registros y memoria. La memoria compartida integrada en el propio chip permite que las tareas que se están ejecutando en estos núcleos compartan datos sin tener que enviarlos a través del bus de memoria del sistema.

2.2.1. Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA): arquitectura de dispositivos de cómputo unificados. Es un modelo de programación introducido por NVIDIA para soportar la ejecución conjunta entre CPU y GPU en una aplicación. CUDA es una plataforma y un modelo de programación para GPU que proporciona extensiones para lenguaje C/C++ y una API para la programación y administración de GPU [Represa Pérez et al., , Putt, 2018].

El modelo de programación CUDA asume un sistema compuesto por un host (CPU) y un device (GPU), y cada uno de ellos con su espacio de memoria, por lo que necesitaremos funciones específicas para reservar y liberar la memoria del dispositivo, así como funciones para la transferencia datos entre la memoria del host y del device. Para poder reservar espacio en la zona de memoria global del device y poder acceder a ella desde el host se utiliza la función *cudaMalloc()*.

Una vez que tenemos un espacio reservado en la memoria global de nuestro dispositivo, lo siguiente que podemos hacer es transferir datos entre esta memoria y la de nuestra CPU. Para ello utilizamos otra función parecida a la que disponemos en C estándar para tal efecto, sólo que en este caso tendremos algún parámetro adicional que nos permita especificar el origen y el destino de los datos. Esta función es *cudaMemcpy()*.

Tenemos cuatro posibilidades y corresponden a los distintos sentidos de transferencia de datos entre el host y el device (ver Tabla 1). Por último, la memoria reservada también se puede liberar, y para ello se utiliza la función *cudaFree()* [Harris, 2017].

Tabla 2: Tipos de transferencias de datos en CUDA.

Tipo de Transferencia	Sentido de transferencia
cudaMemcpyHostToHost	Host \Rightarrow Host
cudaMemcpyHostToDevice	Host \Rightarrow Device
cudaMemcpyDeviceToHost	Device \Rightarrow Host
cudaMemcpyDeviceToDevice	Device \Rightarrow Device

En CUDA el procesamiento de datos en una, dos y tres dimensiones implica la linealización de la memoria es decir, que todo arreglo multidimensional se linealiza a una sola dimensión (ver Figura 2). Por lo tanto, para acceder a un elemento de la matriz de la imagen de entrada se realiza como se muestra en la Figura 3.

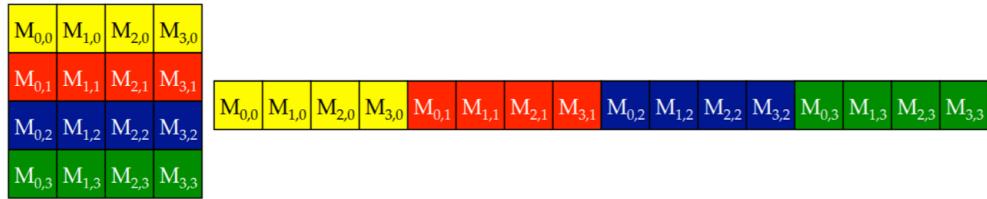


Figura 2: Linealización de la memoria

2.2.2. El Kernel

Un kernel es una función que puede ejecutarse paralelamente en procesos que exhiben paralelismo de datos. Básicamente es la función que se le asigna un hilo en CUDA. Un kernel podría, por ejemplo, realizar la multiplicación de matrices, asignando a cada hilo la operación realizada en cada elemento de una matriz \mathbf{P} . La sintaxis es muy similar a C, tal como se muestra en la Tabla 3.

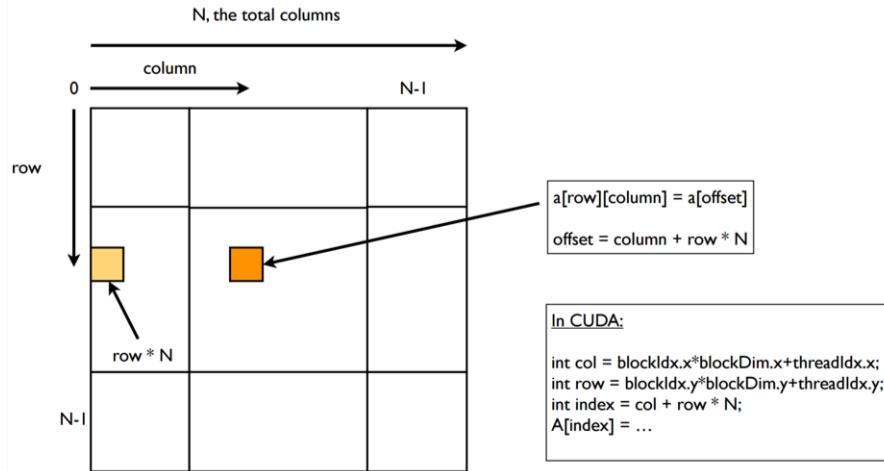


Figura 3: Acceso a los elementos de una matriz

Tabla 3: Similitud de la sintaxis de C y CUDA .

C	CUDA
1 // Hello World - C	1 // Hello World - CUDA
2 void c_hello(){	2 __global__ void cuda_hello(){
3 printf ("Hello World!\n");	3 printf ("Hello World from GPU!\n");
4 }	4 }
5 int main() {	5 int main() {
6 c_hello();	6 cuda_hello<<<1,1>>>();
7 return 0;	7 return 0;
8 }	8 }
9	9

La principal diferencia entre la implementación de C y CUDA es el especificador `__global__` y la sintaxis `<<< ... >>>`. El especificador `__global__` indica una función que se ejecuta en el device (GPU). Dicha función se puede llamar a través del código del host, por ejemplo, en el `main()`. Cuando se llama a un kernel, su configuración de ejecución se proporciona a través de la sintaxis `<<< ... >>>`, por ejemplo. `cuda_hello <<< 1, 1 >>> ()`. En la terminología de CUDA, esto se llama “lanzamiento del kernel”. La sintaxis de la configuración de ejecución del kernel es la siguiente:

`Kernel_name <<< M, T >>> (...);`

Lo que indica que un kernel se inicia con una cuadrícula de M bloques de hilos y cada bloque de hilo tiene T hilos paralelos. Los hilos son unidades de cómputo paralelo y cada hilo adopta la función definida por el kernel. CUDA usa la configuración de ejecución del kernel para determinar en tiempo de ejecución cuántos subprocessos se inician en la GPU. CUDA organiza los hilos en un grupo llamado “thread block” (“bloques de hilos”) los cuales pueden representar operaciones de datos de hasta en 3 dimensiones. Estos bloques no deben contener más de 1024 hilos. Por otro lado el kernel puede lanzar múltiples bloques de hilos, los cuales son organizados en estructura de “malla” o “Grid”. Este tipo de bloques y su organización en una mallas son mostrados en la Figura 4, esta jerarquía es bastante útil cuando se trata de procesamiento digital de imágenes, donde los datos están organizados en arreglos de dos dimensiones, recordando que por conveniencia, los bloques y las mallas pueden tener una, dos o tres dimensiones.

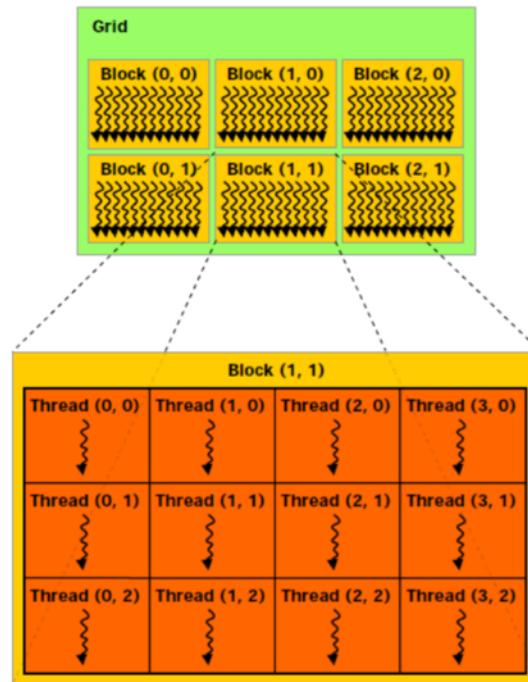


Figura 4: Organización de hilos, bloques y mallas en CUDA [Represa Pérez et al.,]

3. Implementación

En esta sección se describe la implementación y las pruebas del algoritmo secuencial y paralelo del filtro de Sobel añadiendo también una implementación más con las librerías de OpenCV. Cabe señalar que ambos algoritmos se basan en el análisis matemático realizado en la sección 2.1.1. El código completo se encuentra al final del documento en el apéndice A y se encuentra de forma publica en el repositorio [[Levid Rodriguez, 2019](#)].

Las pruebas son realizadas con tres tipos de imágenes de diferentes resoluciones, la primera es la imagen de Lena, muy popular en el procesamiento de imágenes. Las siguientes dos son imágenes aéreas de la Universidad Tecnológica de la Mixteca UTM, capturadas por un dron DJI Mavic Pro a una altura de 50 mts., una de 1280×720 pixeles, y la ultima de 3000×4000 pixeles.

La compilación del software se ha utilizado CUDA Toolkit 10 en un sistema operativo Ubuntu 18.04.2 LTS y una tarjeta gráfica NVIDIA Testla T4. Todo el procesamiento se realizo utilizando le servicio gratuito de Google Colab [[Google, 2019](#), [Tech, 2019](#)]. Para tener una tercera opción de comparación de la eficiencia del algoritmo paralelizado, se implementa el filtro de Sobel utilizando las librerías de OpenCV con la intención de comparar ésta implementación con un framework conocido.

3.1. Algoritmo secuencial

La implementación del algoritmo secuencial se basa en el diagrama de flujo de la Figura 5. Básicamente se puede resumir en la aplicación de la ecuación (5) para cada valor de pixel de la imagen de entrada, es decir el gradiente de cada valor de pixel tanto en dirección x como en dirección y . Posteriormente con la ecuación (2) se obtiene la magnitud de cada gradiente y con ello se determinar los bordes presentes en la imagen.

La implementación en C++ se muestra con detalle en el Apéndice A en la función llamada ***SobelFilterCPU()***. Los resultados de la detección de bordes mediante el algoritmo secuencial de Sobel de las tres imágenes de prueba se muestran en las Figura 6 indicando cada una el tiempo promedio de procesamiento.

3.2. Algoritmo paralelizado

De acuerdo al diagrama de la Figura 5 se observa que para aplicar las operaciones de gradiente es necesario realizar el recorrido de toda la imagen, es decir, se realiza un barrido del arreglo bidimensional en sus direcciones x y y . Sin duda esta es la parte del algoritmo que más recursos computacionales consume, afortunadamente es también la parte del algoritmo que es posible paralelizar, asignando un hilo de ejecución a cada operación de gradiente y su respectiva magnitud para formar la nueva imagen con los bordes detectados (ver Figura 7).

Antes de implementar la paralización del algoritmo es necesario configurar los recursos de la unidad de procesamiento gráfico (GPU) es decir, configurar el número de hilos de ejecución que se requieren para el procesamiento, así como también la forma en que deben organizarse en los respectivos bloques y mallas.

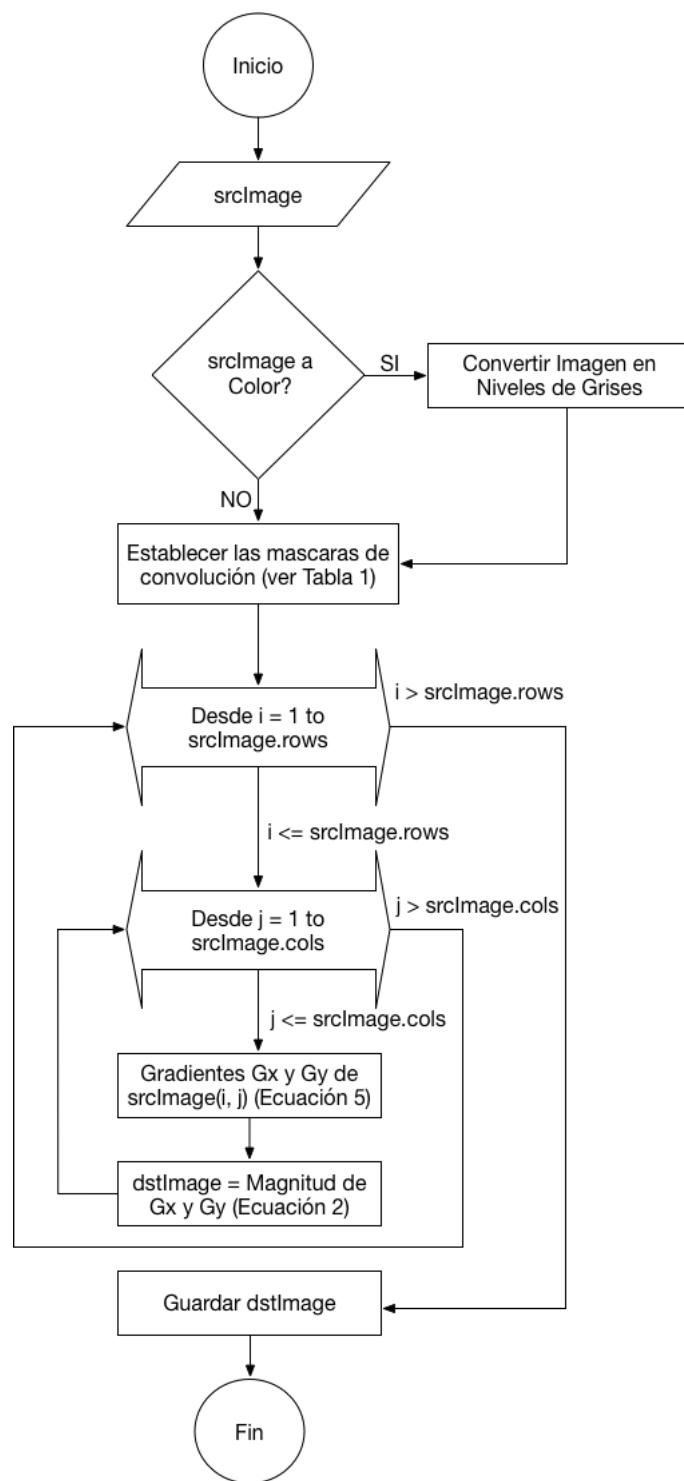


Figura 5: Algoritmo secuencial del filtro de Sobel



(a) Lena - 512x512 pixeles.
Tiempo promedio de ejecución:
2,179ms

(b) UTM área deportiva -
1280x720 pixeles. Tiempo pro-
medio de ejecución: 6,796ms

(c) UTM plaza principal -
4000x3000 pixeles. Tiempo pro-
medio de ejecución: 99,521ms

Figura 6: Resultados del algoritmo secuencial de Sobel

Esto se realizó con una variable global que representa el número de hilos de ejecución por bloque que serán utilizados, teniendo en cuenta que al ser un arreglo bidimensional se necesitan hilos para cada dimensión de la imagen de entrada sin exceder los 1024 hilos por bloque. Por lo tanto, se reservan 30 hilos de ejecución por bloque (línea 12 ver Apéndice A) que serán suficientes para el procesamiento de la información de entrada. Por otro lado, el número de bloques se determina mediante la división del total de renglones y columnas de la imagen de entrada por el número de hilos seleccionado, de ésta forma se han configurado las mallas a utilizar, que en este caso fue solo de una dimensión ya que las imágenes a procesar son en escala de grises, por lo cual con una única malla es suficiente. En esta implementación solo es lanzado un kernel para realizar el procesamiento, sin embargo se configuran los streams y se sincronizan los eventos para optimizar los accesos a memoria.

Después de las configuraciones del device, se implementa el kernel llamado ***sobelFilterGPU()***. Como se muestra en el diagrama de flujo de la Figura 7, en el kernel se implementan las ecuaciones (5) y (2) para realizar el gradiente y la magnitud de cada componente respectivamente. Los resultados del filtro de Sobel programado de forma heterogénea son mostrados en la Figura 8, donde también se indica el tiempo promedio de ejecución de cada una de las imágenes.

3.3. Filtro de Sobel en OpenCV

La implementación del filtro de Sobel con las librerías de OpenCV resulta bastante sencilla, pues las funciones ya se encuentran integradas en el framework, por lo cual, solo se configuran los parámetros correspondientes y se guardan los resultados obtenidos, tal como se muestran en la función ***sobelFilterOpenCV()*** (ver Apéndice A) y los resultados obtenidos son presentados en la Figura 9 indicando también el tiempo de procesamiento correspondiente.

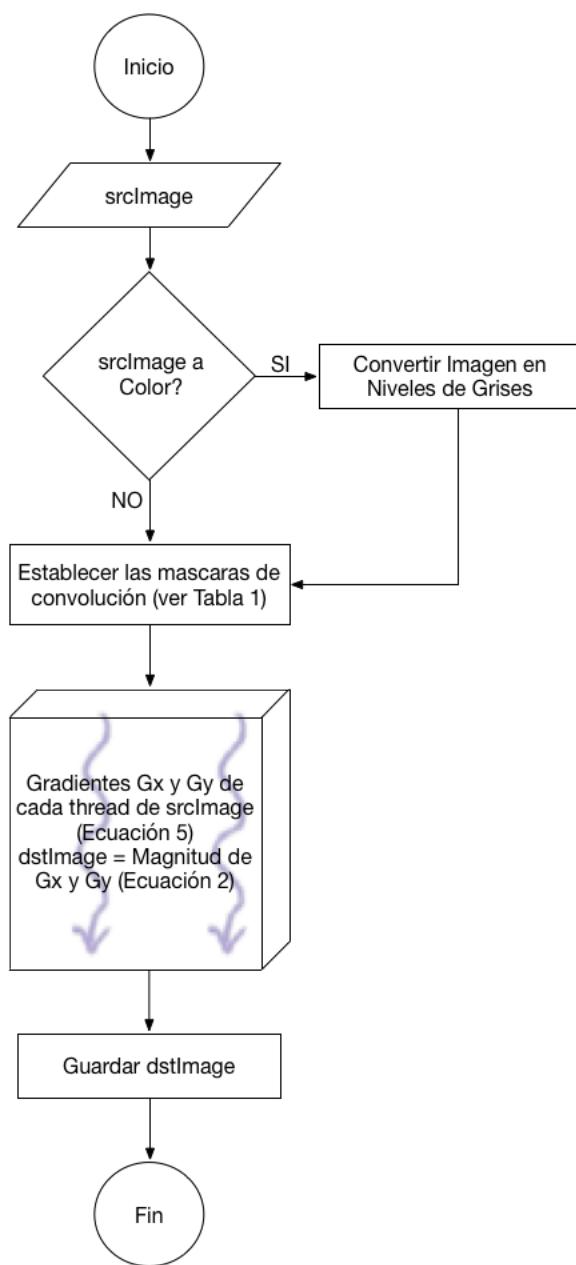


Figura 7: Algoritmo paralelizado del filtro de Sobel



(a) Lena - 512x512 pixeles.
Tiempo promedio de ejecución:
0,044ms



(b) UTM aérea deportiva -
1280x720 pixeles. Tiempo pro-
medio de ejecución: 0,122ms



(c) UTM plaza principal -
4000x3000 pixeles. Tiempo pro-
medio de ejecución: 0,889ms

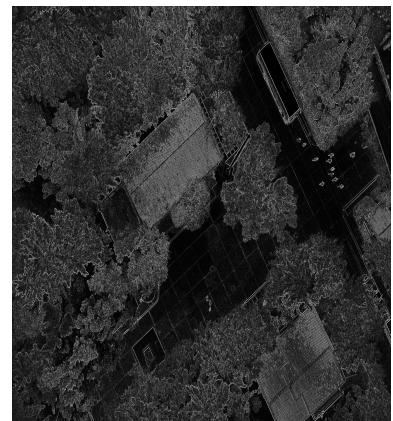
Figura 8: Resultados del algoritmo paralelizado de Sobel



(a) Lena - 512x512 pixeles.
Tiempo promedio de ejecución:
2,179ms



(b) UTM aérea deportiva -
1280x720 pixeles. Tiempo pro-
medio de ejecución: 8,746ms



(c) UTM plaza principal -
4000x3000 pixeles. Tiempo pro-
medio de ejecución: 54,641ms

Figura 9: Resultados del filtro de Sobel con OpenCV

3.4. Resultados

Al observar los resultados obtenidos por cada una de las implementaciones se observa una mejora considerable de los tiempos de procesamiento, principalmente en comparación con la implementación en CUDA. Esta comparación se muestra en la Tabla 4, donde se encuentran las dimensiones de cada imagen, su tiempo de procesamiento en cada implementación y la mejora con respecto a la paralelización realizada.

Tabla 4: Comparación de tiempos de procesamiento y mejor respecto a CUDA

Imagen	Dimensiones	Dispositivo	Tiempo [ms]	Factor de Mejora
Lena	512 x 512 pixeles	CPU	2.179	49.5
		OpenCV	0.865	19.7
		GPU	0.044	
UTM aérea deportiva	1280 x 720 pixeles	CPU	6.796	55.7
		OpenCV	8.746	71.7
		GPU	0.122	
UTM plaza principal	4000 x 3000 pixeles	CPU	99.521	111.9
		OpenCV	54.641	61.5
		GPU	0.889	

Por otra parte en la Tabla 5 se muestran el perfil de la aplicación en CUDA, es decir, un listado de todas las tareas realizadas por el algoritmo paralelizado. En esta tabla se indica el tiempo de ejecución requerido para cada tarea realizada, así como el número de llamadas que se realizaron a cada una durante la ejecución del programa. Con este análisis se puede observar la copia de información entre el host y el device y en el kernel son las tareas que requiere más tiempo de procesamiento. Sin embargo, a pesar de ser las de mayor porcentaje de tiempo aun se encuentran en el orden de milisegundos, incluso en microsegundos para el caso de kernel, por lo que el tiempo sigue siendo casi imperceptible.

Tabla 5: Perfil de la aplicación

Time(%)	Time	Calls	Avg	Min	Max	Name
45.89 %	2.6871ms	1	2.6871ms	2.6871ms	2.6871ms	[CUDA memcpy HtoD]
37.68 %	2.2062ms	1	2.2062ms	2.2062ms	2.2062ms	[CUDA memcpy DtoH]
16.40 %	960.38us	1	960.38us	960.38us	960.38us	sobelFilterGPU()
0.02 %	1.2160us	1	1.2160us	1.2160us	1.2160us	[CUDA memset]

4. Conclusiones

En este trabajo se ha implementado el filtro de Sobel para la detección de bordes en imágenes aéreas de alta resolución. El algoritmo se ha implementado de tres diferentes formas, la primera mediante programación estructurada o secuencial, la segunda con programación heterogénea y como tercer aplicación se utilizó las funciones de OpenCV. Con estas tres implementaciones se obtuvieron resultados suficientemente buenos en la extracción de bordes de las imágenes de ejemplo. Por su parte los tiempos de procesamiento mediante la programación heterogénea y CUDA quedo demostrado que son reducidos considerablemente pues, se ha obtenido un factor de mejora de hasta mas de 100 veces mejor con respecto a un procesamiento en CPU y poco más de 70 veces mejor que una aplicación con OpenCV. La mejora en cuanto al tiempo de procesamiento es evidente, con lo cual la hipótesis plateada queda demostrada de forma afirmativa.

Por otro lado, y a manera de sugerencia también, se debe prestar especial atención a la configuración del device, ya que el número de hilos por bloque de ejecución no debe superar los 1024, por lo que deben ser configurados de forma adecuada para que sea posible procesar todos los elementos del vector de entrada. Recuerde que CUDA trabaja con arreglos unidimensionales y por ello la forma de acceder a los elementos de arreglos multidimensionales cambia y no debe confundirse entre los índices de mallas, bloques e hilos de ejecución.

Los tiempos obtenidos hasta ahora son bastante aceptables sin embargo, se planea, en trabajos futuros, implementar el calculo de los gradientes en kernels independientes y con ayuda de los streams de CUDA se espera el tiempo de procesamiento pueda disminuir aun más. De igual forma se plantea la integración de este algoritmo en uno dedicado a la extracción de características que permita, en la posterioridad, ayudar al stitching de imágenes. Así mismo se pretende llevar este tipo de programación a sistemas más complejos como los son las redes neuronales profundas, donde el tiempo de procesamiento juega un papel muy importantes y este tipo de optimizaciones son requeridas.

Referencias

- [Alegre et al., 2016] Alegre, E., Pajares, G., and De la Escalera, A. (2016). Conceptos y métodos en visión por computador. *España: Grupo de Visión del Comité Español de Automática (CEA)*.
- [Bowyer et al., 2001] Bowyer, K., Kranenburg, C., and Dougherty, S. (2001). Edge detector evaluation using empirical roc curves. *Computer Vision and Image Understanding*, 84(1):77–103.
- [Bu et al., 2016] Bu, S., Zhao, Y., Wan, G., and Liu, Z. (2016). Map2dfusion: Real-time incremental UAV image mosaicing based on monocular slam. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 4564–4571. IEEE.
- [Chen et al., 2017] Chen, Y., Liu, L., Gong, Z., and Zhong, P. (2017). Learning CNN to pair UAV video image patches. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(12):5752–5768.
- [Gleason et al., 2011] Gleason, J., Nefian, A. V., Bouyssounousse, X., Fong, T., and Bebis, G. (2011). Vehicle detection from aerial imagery. In *2011 IEEE International Conference on Robotics and Automation*, pages 2065–2070. IEEE.
- [Google, 2019] Google (2019). Google colaboratory. Online: [<https://colab.research.google.com/notebooks/welcome.ipynb>].
- [Gupta and Mazumdar, 2013] Gupta, S. and Mazumdar, S. G. (2013). Sobel edge detection algorithm. *International journal of computer science and management Research*, 2(2):1578–1583.
- [Harris, 2017] Harris, B. M. (2017). An even easier introduction to cuda. Online: [<https://devblogs.nvidia.com/even-easier-introduction-cuda>].
- [Hu et al., 2007] Hu, J., Razdan, A., Femiani, J. C., Cui, M., and Wonka, P. (2007). Road network extraction and intersection detection from aerial images by tracking road footprints. *IEEE Transactions on Geoscience and Remote Sensing*, 45(12):4144–4157.
- [Hu et al., 2018] Hu, T., Wang, Y., Chen, Y., Lu, P., Wang, H., and Wang, G. (2018). Sobel heuristic kernel for aerial semantic segmentation. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3074–3078. IEEE.
- [Levid Rodriguez, 2019] Levid Rodriguez, L. R. (2019). Sobel_with_opencv-cuda. Online: [https://github.com/LevidRodriguez/Sobel_with_OpenCV-CUDA.git].
- [Li et al., 2014] Li, J., Ai, M., Hu, Q., and Fu, D. (2014). A novel approach to generating dsm from high-resolution uav images. In *Geoinformatics (GeoInformatics), 2014 22nd International Conference on*, pages 1–5. IEEE.
- [Patnaik and Yang, 2012] Patnaik, S. and Yang, Y.-M. (2012). *Soft computing techniques in vision science*, volume 395. Springer Science & Business Media.

- [Putt, 2018] Putt, S. (2018). Cuda tutorial. Online: [<https://cuda-tutorial.readthedocs.io/en/latest>].
- [Represa Pérez et al.,] Represa Pérez, C., Cámara Nebreda, J., and Sánchez Ortega, P. Introducción a la programación en cuda, universidad de burgos, área de tecnología electrónica (2016).
- [Tech, 2019] Tech, A. (2019). Online: [<https://www.youtube.com/watch?v=n7RdjB9bDKo>].
- [Teng et al., 2019] Teng, L., Xue, F., and Bai, Q. (2019). Remote sensing image enhancement via edge-preserving multi-scale retinex. *IEEE Photonics Journal*.
- [Venkateswar and Chellappa, 1992] Venkateswar, V. and Chellappa, R. (1992). Extraction of straight lines in aerial images. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (11):1111–1114.
- [Vincent et al., 2009] Vincent, O. R., Folorunso, O., et al. (2009). A descriptive algorithm for sobel image edge detection. In *Proceedings of Informing Science & IT Education Conference (InSITE)*, volume 40, pages 97–107. Informing Science Institute California.
- [Zhao and Nevatia, 2003] Zhao, T. and Nevatia, R. (2003). Car detection in low resolution aerial images. *Image and Vision Computing*, 21(8):693–703.

A. Código en C++ para el filtro de Sobel.

```

1 #include <thread>
2 #include <chrono>
3 #include <time.h>
4 #include <iostream>
5 #include <math.h>
6 #include <opencv2/imgcodecs.hpp>
7 #include <opencv2/highgui/highgui.hpp>
8 #include <opencv2/stitching.hpp>
9 #include <opencv2/core/utility.hpp>
10
11 #define threadsNumber 30.0
12 void sobelFilterCPU(cv::Mat srcImg, cv::Mat dstImg, const unsigned int width,
13                      const unsigned int height);
14 void sobelFilterOpenCV(cv::Mat srcImg, cv::Mat dstImg);
15
16 __global__ void sobelFilterGPU(unsigned char* srcImg, unsigned char* dstImg,
17                               const unsigned int width, const unsigned int height){
18     int x = threadIdx.x + blockIdx.x * blockDim.x;
19     int y = threadIdx.y + blockIdx.y * blockDim.y;
20     if( x > 0 && y > 0 && x < width-1 && y < height-1) {
21         float dx = (-1* srcImg[(y-1)*width + (x-1)] + (-2*srcImg[y*width+(x-1)
22 ] + (-1*srcImg[(y+1)*width+(x-1)] +
23             (srcImg[(y-1)*width + (x+1)]) + ( 2*srcImg[y*width+(x+1)]) + (
24             srcImg[(y+1)*width+(x+1)]);
25
26         float dy = (    srcImg[(y-1)*width + (x-1)] + ( 2*srcImg[(y-1)*width+x
27 ]) + (    srcImg[(y-1)*width+(x+1)] +
28             (-1* srcImg[(y+1)*width + (x-1)] + (-2*srcImg[(y+1)*width+x]) +
29             (-1*srcImg[(y+1)*width+(x+1)]);
30         dstImg[y*width + x] = sqrt( (dx*dx) + (dy*dy) ) > 255 ? 255 : sqrt( (dx
31             *dx) + (dy*dy) );
32     }
33 }
34
35 int main(int argc, char * argv[]){
36     if(argc != 2){
37         std::cout << argv[0] << "Invalid number of command line arguments.
38         Exiting program" << std::endl;
39         std::cout << "Usage: " << argv[0] << " [image.png]"<< std::endl;
40         return 1;
41     }
42     // Verifica las versiones de GPU, CUDA y OpenCV.
43     cudaDeviceProp deviceProp;
44     cudaGetDeviceProperties(&deviceProp, 0);
45
46     time_t rawTime; time(&rawTime);
47     struct tm* curTime = localtime(&rawTime);
48     char timeBuffer[80] = "";
49     strftime(timeBuffer, 80, "%c", curTime);
50     std::cout << timeBuffer << std::endl;
51
52 }
```

```

44     std :: cout << "GPU: " << deviceProp .name << " , CUDA " << deviceProp .major <<
45         " ." << deviceProp .minor << " , " << deviceProp .totalGlobalMem / 1048576 <<
46             " Mbytes " << std :: endl ; // << cores << " CUDA cores \n" << std :: endl ;
47
48     std :: cout << "OpenCV Version: " << CV_VERSION << std :: endl ;
49
50     // Cargar imagen y la transforma a escala de grises
51     cv :: Mat srcImg = cv :: imread ( argv [ 1 ] ) ;
52     cv :: cvtColor ( srcImg , srcImg , cv :: COLOR_RGB2GRAY ) ;
53     cv :: Mat sobel _cpu = cv :: Mat :: zeros ( srcImg . size () , srcImg . type () ) ;
54     cv :: Mat sobel _opencv = cv :: Mat :: zeros ( srcImg . size () , srcImg . type () ) ;
55
56     unsigned char *gpu_src , *gpu_sobel ;
57     auto start _time = std :: chrono :: system _clock :: now () ;
58     // ——START CPU
59     sobelFilterCPU ( srcImg , sobel _cpu , srcImg . cols , srcImg . rows ) ;
60     std :: chrono :: duration < double > time _cpu = std :: chrono :: system _clock :: now () -
61         start _time ;
62     // ——END CPU
63
64     // ——START OPENCV
65     start _time = std :: chrono :: system _clock :: now () ;
66     sobelFilterOpenCV ( srcImg , sobel _opencv ) ;
67     std :: chrono :: duration < double > time _opencv = std :: chrono :: system _clock :: now ()
68     () - start _time ;
69     // ——END OPENCV
70
71     // ——SETUP GPU
72     // Eventos
73     cudaEvent_t start , stop ;
74     cudaEventCreate (& start ) ;
75     cudaEventCreate (& stop ) ;
76     // Streams
77     cudaStream_t stream ;
78     cudaStreamCreate (& stream ) ;
79     // Asignar memoria para las imágenes en memoria GPU.
80     cudaMalloc ( ( void ** ) & gpu_src , ( srcImg . cols * srcImg . rows ) ) ;
81     cudaMalloc ( ( void ** ) & gpu_sobel , ( srcImg . cols * srcImg . rows ) ) ;
82
83     // Transfiera del host al device y configura la matriz resultante a 0s
84     cudaMemcpy ( gpu_src , srcImg . data , ( srcImg . cols * srcImg . rows ) ,
85     cudaMemcpyHostToDevice ) ;
86     cudaMemset ( gpu_sobel , 0 , ( srcImg . cols * srcImg . rows ) ) ;
87
88     // configura los dim3 para que el gpu los use como argumentos , hilos por
89     // bloque y numero de bloques
90     dim3 threadsPerBlock ( threadsNumber , threadsNumber , 1 ) ;
91     dim3 numBlocks ( ceil ( srcImg . cols / threadsNumber ) , ceil ( srcImg . rows /
92     threadsNumber ) , 1 ) ;
93
94     // ——START GPU
95     // Ejecutar el filtro sobel utilizando la GPU.
96     cudaEventRecord ( start ) ;

```

```
90     start_time = std::chrono::system_clock::now();
91     sobelFilterGPU<<< numBlocks , threadsPerBlock , 0 , stream >>>(gpu_src ,
92     gpu_sobel , srcImg.cols , srcImg.rows);
93     cudaError_t cudaerror = cudaDeviceSynchronize(); // waits for completion ,
94     returns error code
95     // if error , output error
96     if ( cudaerror != cudaSuccess )
97         std::cout << "Cuda failed to synchronize: " << cudaGetErrorName(
98         cudaerror ) << std::endl;
99     std::chrono::duration<double> time_gpu = std::chrono::system_clock::now() -
100    start_time;
101   // ----END GPU
102
103  // Copia los datos al CPU desde la GPU, del device al host
104  cudaMemcpy(srcImg.data , gpu_sobel , (srcImg.cols*srcImg.rows) ,
105  cudaMemcpyDeviceToHost);
106  // Libera recursos
107  cudaEventRecord(stop);
108  float time_milliseconds =0;
109  cudaEventElapsedTime(&time_milliseconds , start , stop);
110  cudaStreamDestroy(stream);
111  cudaFree(gpu_src);
112  cudaFree(gpu_sobel);
113  /** Tiempos de ejecuci\on de cada m\'etodo de filtrado por sobel ***/
114  std::cout << "Archivo: "<< argv[1] << ":" <<srcImg.rows<<" rows x "<<srcImg
115  .cols << " columns" << std::endl;
116  std::cout << "CPU execution time = " << 1000*time_cpu.count() <<" msec"<<
117  std::endl;
118  std::cout << "OPENCV execution time = " << 1000*time_opencv.count() <<" msec"<<std::endl;
119  std::cout << "CUDA execution time = " << 1000*time_gpu.count() <<" msec"
120  <<std::endl;
121
122 // Guarda resultados
123 cv::imwrite("outImgCPU.png" , sobel_cpu);
124 cv::imwrite("outImgOpenCV.png" , sobel_opencv);
125 cv::imwrite("outImgGPU.png" , srcImg);
126
127 return 0;
128 }
129
130 void sobelFilterCPU(cv::Mat srcImg , cv::Mat dstImg , const unsigned int width ,
131 const unsigned int height){
132     for(int y = 1; y < srcImg.rows-1; y++) {
133         for(int x = 1; x < srcImg.cols-1; x++) {
134             float dx = (-1*srcImg.data[(y-1)*width + (x-1)]) + (-2*srcImg.data[
135 y*width+(x-1)]) + (-1*srcImg.data[(y+1)*width+(x-1)]) +
136             (srcImg.data[(y-1)*width + (x+1)]) + (2*srcImg.data[y*width+(x+1)])
137             + (srcImg.data[(y+1)*width+(x+1)]);
138
139             float dy = (srcImg.data[(y-1)*width + (x-1)]) + (2*srcImg.data[(y
140 -1)*width+x]) + (srcImg.data[(y-1)*width+(x+1)]) +
141             (-1*srcImg.data[(y+1)*width + (x-1)]) + (-2*srcImg.data[(y+1)*width
```

```
130         +x]) + (-1*srcImg.data[(y+1)*width+(x+1)]) ;
131         dstImg.at<uchar>(y,x) = sqrt( (dx*dx) + (dy*dy) ) > 255 ? 255 :
132             sqrt( (dx*dx) + (dy*dy) );
133     }
134 }
135
136 void sobelFilterOpenCV(cv::Mat srcImg, cv::Mat dstImg){
137     cv::Mat grad_x, grad_y, abs_grad_x, abs_grad_y;
138     // Gradiente X
139     cv::Sobel(srcImg, grad_x, CV_16S, 1, 0, 3, 1, 0, cv::BORDER_DEFAULT);
140     cv::convertScaleAbs(grad_x, abs_grad_x);
141     // Gradiente Y
142     cv::Sobel(srcImg, grad_y, CV_16S, 0, 1, 3, 1, 0, cv::BORDER_DEFAULT);
143     cv::convertScaleAbs(grad_y, abs_grad_y);
144     // Une los gradientes
145     addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, dstImg );
146 }
```