Paralelización Del Filtro De Sobel Para La Detección De Bordes De Una Imagen Mediante La Tecnología CUDA

Carolina Jiménez Gómez, German David Gómez, y Jhon Edinson Acevedo

Abstract—This article discusses the implementation of the Sobel filter with CUDA technology to take advantage of the Graphic Processing Unit (GPU), making a comparison between the different types of memory that the GPU has and it's optimizations in different computational algorithms. We give a brief description of the algorithm of Sobel and an example of convolution on the images, to after deepen into the differences between different types of memory. We show comparison tables between the different algorithms and makes an analysis of the acceleration between them.

Index Terms—CUDA, GPU, C++, OpenCV, CPU, Paralelismo, Optimización, Filtro Sobel, Procesamiento de imágenes.

1 Resumen

En este artículo se hablará sobre la implementación del filtro de Sobel con la tecnología CUDA para el aprovechamiento de la Unidad de Procesamiento Gráfico (GPU, por sus siglas en Inglés), haciendo una comparación entre los diferentes tipos de memoria que tiene la GPU y con las cuales se pueden hacer optimizaciones de diferentes algoritmos computacionales. Se da una breve descripción del algoritmo de Sobel y un ejemplo de convolución sobre las imágenes, para después adentrarnos en las diferencias sustanciales entre los diferentes tipos de memoria. Se muestran gráficas de comparación entre los diferentes algoritmos y se hace un análisis de aceleración entre ellos.

2 Introducción

El filtro de Sobel se utiliza en el procesamiento de imágenes y visión por computadora, particularmente dentro de los algoritmos de detección de bordes. Esto se logra gracias a la información proporcionada por las fronteras de los objetos que aparecen en una imagen como las discontinuidades en los niveles de grises. En concreto el filtro de Sobel se trata de un filtro de aproximación al gradiente. El cálculo de la derivada direccional de una función nos habla de cómo se producen los cambios en tal dirección, tales cambios, que se asocian con las altas frecuencias, suelen corresponder a los bordes de los objetos presentes en las imágenes [1].

Para este fin se debe procesar la imagen pixel a pixel, lo que se traduce en muchas operaciones secuenciales y por ende mucho tiempo de procesamiento, por lo que se plantea como solución a este problema la paralelización por medio de la tecnología CUDA [2] de Nvidia con la cual se espera obtener un rendimiento computacional mucho mayor.

El objetivo de este artículo es mostrar las ventajas de usar paralelización usando una GPU y el lenguaje CUDA de Nvidia para procesamiento de grandes cantidades de datos. En este caso los datos a procesar son imágenes, por esta razón, en este artículo no se pretende demostrar las bases matemáticas sobre las cuales funcionan la conversión a escala de grises, convolución y el filtro de Sobel, que son procesos aplicados a cada imagen.

3 Materiales y métodos

Los algoritmos fueron procesados en un computador Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz de arquitectura x86_64, con 8 CPUs, L1d cache y L1i cache de 32K, L2 cache de 256K y L3 cache de 8192K; un Xeon E3-1200 v3/4th Gen Core Processor Integrated Graphics Controller de Intel Corporation.

Se utiliza la tecnología CUDA para la paralelización de los algoritmos, la librería de OpenCV [3] para el tratamiento general de imágenes y para la implementación del algoritmo de forma secuencial; se utiliza el lenguaje de programación C++ [4] para las implementaciones.

La librería OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel. OpenCV pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado realizando su programación en código C y C++ optimizados, aprovechando las capacidades que proveen los procesadores multinúcleo [5].

Se utiliza OpenCV para la carga de las imágenes y la posterior escritura de la imagen resultante en todos los algoritmos.

Para la realización de las pruebas experimentales se tuvieron en cuenta diez imágenes con tamaños entre 2K y 8K sobre las cuales cada algoritmo fue ejecutado veinte veces para posteriormente realizar un promedio de estos valores y tener un tiempo de ejecución mucho más confiable.

Todos los algoritmos implementados pueden ser encontrados en [6]

3.1 Tecnología CUDA

La tecnología CUDA es una arquitectura de computación paralela desarrollada por uno de los mayores fabricantes

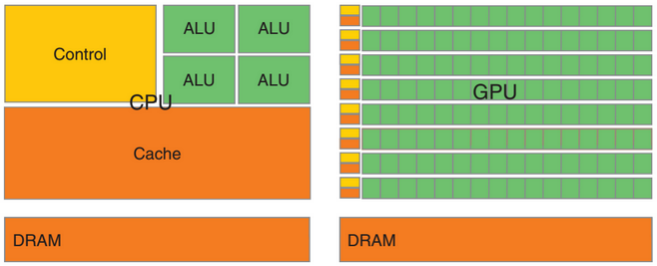


Figure 1. Diferencia de arquitecturas entre CPU (izquierda) y GPU(derecha). Extraído de [8]

de tarjetas gráficas del mercado, NVIDIA. Fue introducida en Noviembre de 2006, y se basa en la utilización de un elevado número de nodos de procesamiento para realizar operaciones sobre un gran volumen de datos en paralelo, consiguiendo reducir el tiempo de procesado y obteniendo altas prestaciones [7].

Uno de los principales objetivos de esta tecnologıa es resolver problemas complejos que llevan asociados una alta carga computacional sobre la CPU de la forma más eficiente, haciendo uso de las GPUs incorporadas en la tarjeta gráfica. Para ello se realiza un procesado masivo de los datos, consiguiendo reducir considerablemente los tiempos de ejecución de la aplicación.

Antes de comenzar a paralelizar algún algoritmo o aplicación, es necesario llevar a cabo un estudio específico con el fin de identificar el mejor enfoque para el uso de los recursos ofrecidos por CUDA en cada caso. En la terminología de CUDA, la función o el código de programa que se ejecuta en la GPU reciben el nombre de *kernel*. Un *kernel* se procesa en paralelo por un conjunto de hilos que ejecutará las instrucciones de ese *kernel* en una parte diferente de los datos en memoria. Los *kernel* se lanzan en *grids* y sólo un *kernel* se ejecuta en un momento dado en una GPU. La GPU tiene varios multiprocesadores, por lo que los hilos son agrupados en bloques. La ejecución de bloque se lleva a cabo en los multiprocesadores. Al conjunto de bloques se le denomina *grid*.

Hay varios tipos de memoria disponible en CUDA. Para este proyecto se hará uso de la memoria global, la memoria constante y la memoria compartida. Cada una de estas memorias tiene diferente tipo de acceso y capacidad de almacenamiento.

La memoria global es la más utilizada en una GPU por su tamaño, aunque es de alta latencia pero mayor capacidad. Este tipo de memoria puede ser accedida desde cualquier multiprocesador durante el periodo de vida del programa.

La memoria compartida es más rápida y de menor latencia que la memoria global. Cada multiprocesador tiene un límite total de memoria compartida que es particionada entre todos los hilos de los bloques. Cada bloque tiene su propio espacio de memoria compartida, esto significa que cada hilo sólo puede acceder a su propio espacio de memoria compartida. Esta memoria es declarada dentro de la función kernel pero compartiendo el mismo tiempo de vida con los hilos del bloque.

La memoria constante, como su propio nombre indica, se usa para albergar datos que no cambian durante el transcurso de ejecución de un kernel. La capacidad de memoria constante disponible depende de la capacidad de cómputo de la GPU, y su principal ventaja es que en algunas situaciones el uso de memoria constante en lugar de la memoria global pueda reducir considerablemente el ancho de banda de memoria requerido por la aplicación. Dado que la memoria constante no puede ser cambiada en tiempo de ejecución, no es posible usarla para guardar los datos de las imágenes, por lo que en este caso particular, esta memoria sólo será usada para almacenar las máscaras de convolución las cuales son vectores de sólo lectura.

La utilización de una memoria u otra, va a depender de la aplicación que se esté desarrollando y de las características de la GPU con la que se cuente, pues dependiendo de la capacidad de la misma, varía el tamaño de memorias.

3.2 Convolución y Filtro de Sobel

Antes de explicar en qué consiste el filtro de Sobel es importante mencionar que para que el filtro funcione, la imagen sobre la cual éste opera debe estar en escala de grises. Para esto se hace un procesamiento sobre la imagen original. Este proceso consiste en recorrer cada píxel de la imagen a color y modificar los canales RGB (Red, Green, Blue) con unos valores ya determinados [9] que juntos forman y dan color a un píxel.

La convolución es una operación matemática que consiste en pasar un vector, llamado máscara de convolución, por cada uno de los pixeles vecinos de una imágen, multiplicar dichos pixeles y obtener su suma que serán almacenadas en el píxel donde nos encontramos. Los operadores de Sobel combinan el suavizado y la diferenciación de Gauss, por lo que el resultado es más o menos resistente al ruido. Los valores de las máscaras están dados por dichos operadores, con los cuales se calculan los cambios horizontales y verticales en la imagen [10].

1	0	1
-1	0	2
1	0	1

(a) Máscara de convolución.

5	6	4	2	1
2	5	4	1	3
3	5	5	4	5
2	4	4	4	1
3	1	2	3	2

(b) Imagen de entrada, se aplica la máscara sobre el píxel del centro y se suma la multiplicación de los pesos con sus vecinos (región azul).

5	6	4	2	1
2	5	4	1	3
3	5	17	4	5
2	4	4	4	1
3	1	2	3	2

(c) Salida del píxel del centro.

Figure 2. Elaboración propia

Una vez se hayan detectado los cambios en X y Y con el recorrido de las máscaras de convolución, se haya la norma y la dirección del gradiente, el cual está expresado de la siguiente forma:

$$G = \sqrt[2]{Gx^2 + Gy^2} \tag{1}$$

$$\theta = atan\left(\frac{Gy}{Gx}\right) \tag{2}$$

[1] [11]

4 Implementación y resultados experimentales

Se realizará una comparación de los tiempos obtenidos por cada implementación en los diferentes tipos de memoria de la GPU, y una implementación extra el cual mostrará los resultados de dicho algoritmo en la CPU (secuencial) por medio de la librería OpenCV.

En la tabla 1. se realiza una descripción de las características de cada una de las imágenes.

Table 1
Características de las imágenes. Elaboración propia.

Imagen	Tamaño Imagen (Bytes)	Píxeles
1	46398	550 x 340
2	116519	580 x 580
3	86406	638 x 640
4	391282	1366 x 768
5	616630	2560 x 1600
6	2171320	4928 x 3264
7	10178230	5226 x 4222
8	2335593	12000 x 6000
9	6977173	12000 x 9000
10	5950639	19843 x 8504

4.1 Implementación secuencial

OpenCV ofrece gran cantidad de funciones para procesamiento de imágenes, entre ellas se encuentra el filtro de Sobel. Para hacer uso de tal funcionalidad, es necesario proveerle una imagen de entrada, junto a otros parámetros, para ejecutar el filtro de Sobel. OpenCV usa internamente los hilos de la CPU para realizar estos procesos, es decir, la librería optimiza las funciones que implementa. Aún así, se puede lograr un mejor desempeño usando la GPU con CUDA, dado a las diferencias de la arquitectura ya antes mostrada entre la CPU y la GPU.

En la tabla 2 se muestran los tiempos de ejecución obtenidos al realizar el filtro de sobel con la librería Opencv para cada imagen.

Table 2
Tiempos de ejecución con OpenCV.

Imagen	Tiempo Promedio (segundos)
1	0,15288965
2	0,16600140
3	0,16263135
4	0,22078000
5	0,29979610
6	0,77888230
7	0,53865560
8	1,68595235
9	2,60327430
10	3,82147200

4.2 Implementación con memoria global

Esta versión utiliza la memoria global (DRAM) de la GPU para almacenar, cargar y guardar el resultado del procesamiento en las imágenes. Esta memoria es de latencia alta pero es la de mayor capacidad, por esto es útil para almacenar los datos completos de las imágenes y máscaras. Esta memoria tiene un canal directo de comunicación con el host (CPU) por medio del puerto PCIExpress.

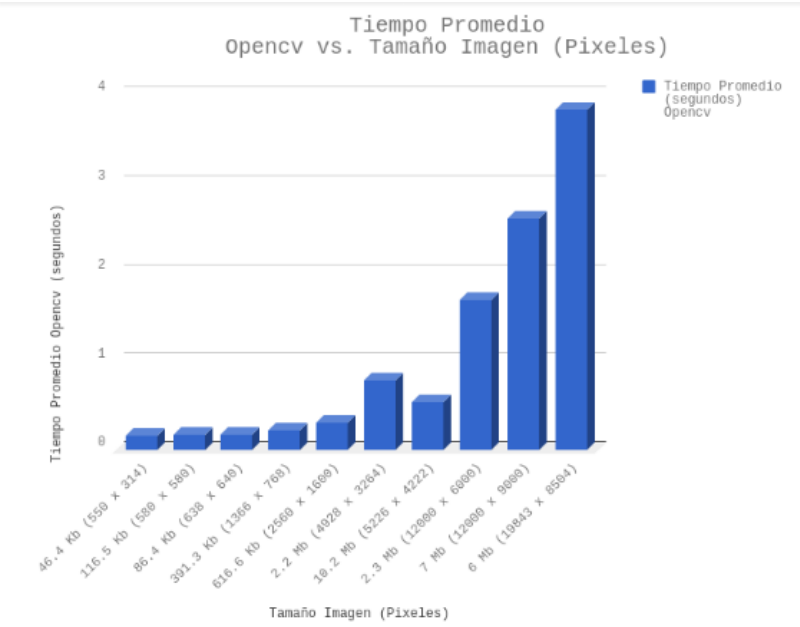


Figure 3. Elaboración propia

La figura 4 muestra la imagen de entrada al algoritmo, la conversión a escala de grises y la imagen resultado con el filtro de Sobel.

(a) Imagen de entrada

(b) Imagen en escala de grises

(c) Image con el filtro e Sobel

Figure 4. Elaboración propia

En la tabla 3 se muestran los tiempos de ejecución obtenidos al realizar el filtro de Sobel con la memoria global para cada imagen.

Table 3
Algoritmo con memoria global.

Imagen	Memoria Global (segundos)
1	0,00031665
2	0,00055765
3	0,00066810
4	0,00165485
5	0,00562505
6	0,02804830
7	0,02051475
8	0,08429345
9	0,12623075
10	0,19830240

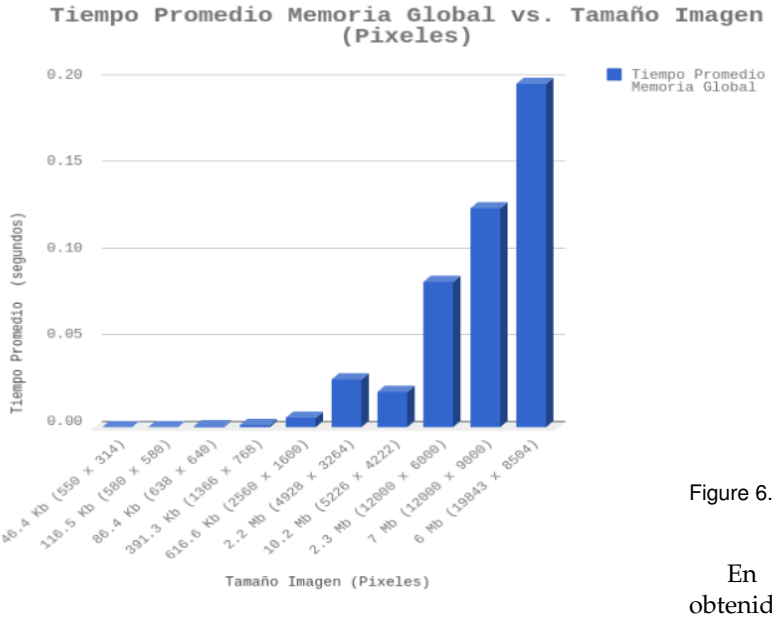
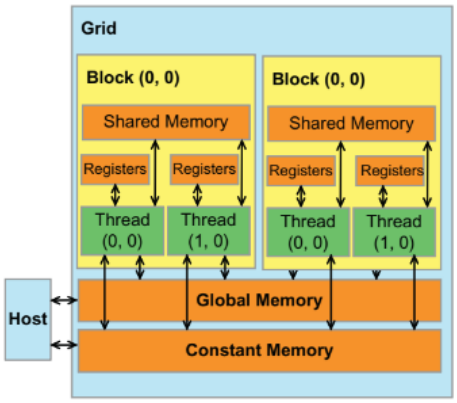
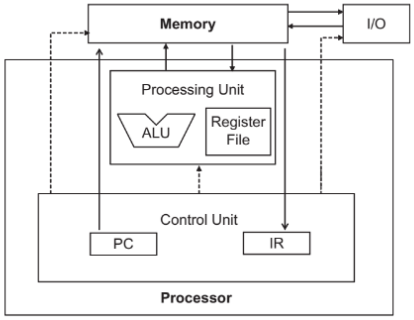


Figure 5. Elaboración propia



(a) Arquitectura de la GPU.



(b) Arquitectura de una CPU moderna basada en el modelo de John Von Neumann.

Figure 6. Extraída de [8]

4.3 Memoria Compartida

La memoria compartida es otro tipo de memoria existente en la GPU (cache L1), con la diferencia de la global en que es accesada mucho más rápido por los hilos de ejecución y es compartida por un conjunto de hilos de cada bloque, también es más pequeña que la memoria global. Se hace necesario el uso de memoria compartida puesto que si todos los hilos de la GPU están tratando de acceder a la memoria global, se creará un cuello de botella, haciendo que nuestros programas se ejecuten más lento. Al hacer uso de la memoria compartida se aumenta la complejidad de entendimiento e implementación del algoritmo, puesto que se deben tener en cuenta muchas más variables, como de que se deben estar sincronizando los hilos de cada bloque para que no accedan a datos que todavía no han sido modificados por los otros hilos y el mapeo que debe hacerse a la memoria compartida desde la memoria global.

La figura 6 muestra la comparación entre la arquitectura de la GPU con una arquitectura moderna basada en el modelo propuesto por *John Von Neumann*; se puede apreciar que la memoria compartida es de acceso mucho más rápido ya que se encuentra dentro del chip del procesador, mientras que acceder a la memoria global (DRAM) es mucho más ”lejana” y por lo tanto más lenta al tratar de acceder a ella.

En la tabla 4 se muestran los tiempos de ejecución obtenidos al realizar el filtro de Sobel con la memoria compartida para cada imagen.

Table 4
Algoritmo con memoria compartida

Imagen	Memoria Compartida (segundos)
1	0.00024280
2	0.00040695
3	0.00047495
4	0.00111000
5	0.00400425
6	0.01429470
7	0.01944510
8	0.06232435
9	0.08929670
10	0.13908525

4.4 Memoria constante

CUDA pone a nuestra disposición otro tipo de memoria conocida como memoria constante. Como su nombre lo indica, utilizamos la memoria constante para datos que no cambiarán en el transcurso de la ejecución del kernel. NVIDIA proporciona 64 KB de memoria constante que trata de manera diferente a como trata la memoria global estándar. En algunas situaciones, usar memoria constante en lugar de memoria global reducirá el ancho de banda de memoria requerido. En esta memoria sólo se almacenará la máscara de convolución puesto que no permite escritura en tiempo de ejecución [12].

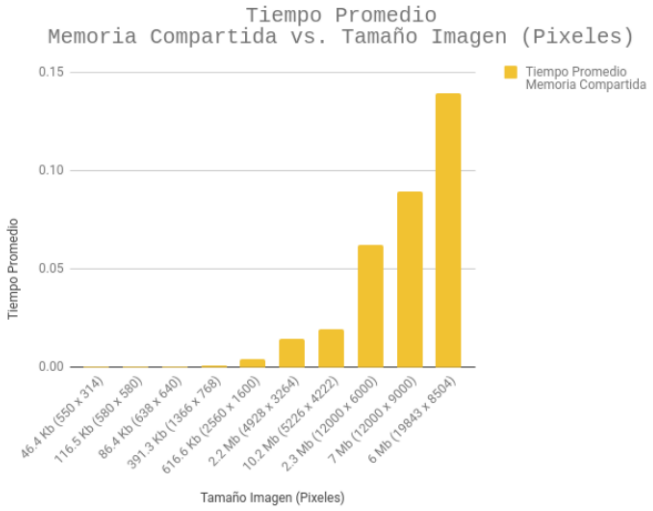


Figure 7. Elaboración propia

La figura 8 muestra cómo está distribuida la memoria constante en la GPU.

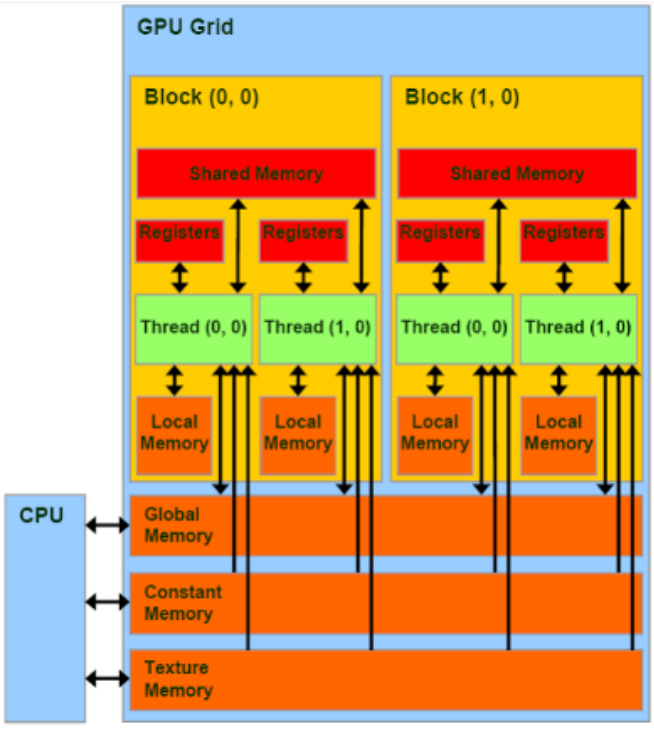


Figure 8. Extraído de [8]

En la tabla 5 se muestran los tiempos de ejecución obtenidos al realizar el filtro de Sobel con la memoria constante para cada imagen.

5 Comparación de resultados

En la figura 10 se encuentran las comparaciones en tiempos de cada uno de los algoritmos implementados.

Table 5
Algoritmo con memoria constante

Imagen	Memoria Constante (segundos)
1	0,00028225
2	0,00045420
3	0,00052085
4	0,00108675
5	0,00347660
6	0,01665020
7	0,01216705
8	0,05257680
9	0,11910089
10	0,11962325

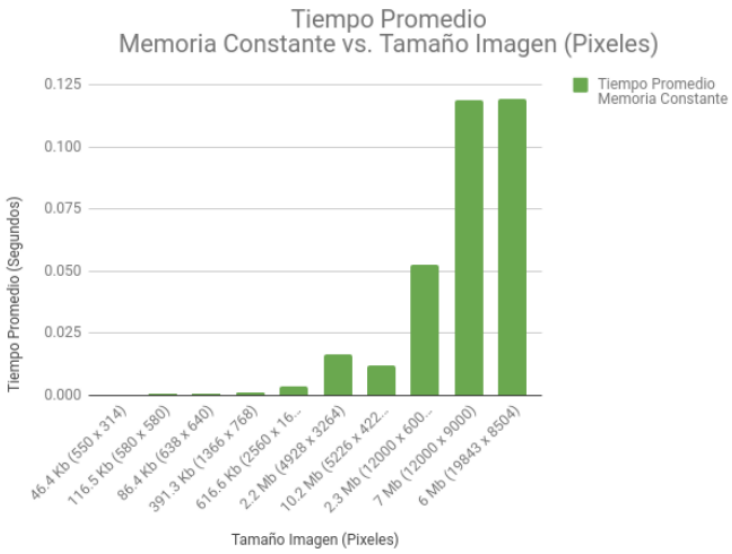


Figure 9. Elaboración propia

6 Conclusiones

- El uso del paralelismo (en las tres implementaciones con los diferentes tipos de memorias) muestra una evidente mejora en el tiempo de procesamiento sobre cada una de las imágenes en comparación con la versión secuencial (implementación ofrecida por OpenCV)
- La mejora de procesamiento al usar paralelismo con GPU y CUDA se hace más evidente cuando la cantidad de datos a procesar es muy grande. En este caso se hace evidente con las imágenes de mayor cantidad de píxeles, como las imágenes 8(12000x6000px), 9(12000x9000px) y 10(19843x8504px)
- Se logra evidenciar como el uso de memorias de baja latencia como la constante y la compartida permite una mejora en el rendimiento de la implementación.
- En la implementación de memoria compartida se usó una combinación entre esta memoria, para guardar temporalmente los datos de la imagen, y la memoria constante para almacenar las máscaras de convolución. Se esperaba que esta implementación fuera la más rápida. La razón por la que no sucede esto es porque la cantidad de operaciones que realiza cada hilo de la GPU accediendo a los datos de la memoria

(a) Comparación de tiempos entre memorias de la GPU y algoritmo secuencial

(b) Comparación de tiempos entre memorias de la GPU

Figure 10. Elaboración propia

compartida es muy poca, dicho de otra forma, cada hilo sólo ejecutará la cantidad de operaciones dada por el ancho máximo de la máscara. Son muy pocas operaciones en las cuales no se aprovecha al máximo esta técnica si se compara con un producto punto.

7 Trabajos Futuros

- Mejorar el desempeño con el uso de warps aprovechando también mayor capacidad de procesamiento de la GPU
- Hacer uso de la memoria compartida junto con la memoria constante para compararlo con los resultados ya obtenidos en este informe.

Referencias

[1] OpenCV. Sobel derivatives. [Online]. Available: https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html
[2] NVIDIA. Cuda zone. [Online]. Available: https://developer.nvidia.com/cuda-zone
[3] OpenCV. Opencv. [Online]. Available: https://opencv.org/
[4] C++. Cpp reference. [Online]. Available: http://en.cppreference.com/w/cpp/language
[5] CuatroRíosTecnologías. Opencv librería de visión por computador. [Online]. Available: http://www.cuatrorios.org/index.php?option=com_content&view=article&id=169:opencv-librer\%C3\%ADa-de-visi\%C3\%B3n-por-computador&catid=39:blogsfeeds
[6] Secondpartialhpc. [Online]. Available: https://github.com/jeacevedo92/SecondPartialHPC
[7] NVIDIA. QuÉ es cuda. [Online]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html
[8] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
[9] OpenCV. Color conversions. [Online]. Available: https://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html
[10] ——. Sobel. [Online]. Available: https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=sobel#sobel
[11] A. W. Robert Fisher, Simon Perkins and E. Wolfart. Sobel edge detector. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
[12] N. Gupta. What is constant memory in cuda. [Online]. Available: http://cuda-programming.blogspot.com.co/2013/01/what-is-constant-memory-in-cuda.html