

# Implementación de un efecto sobre una imagen

## Práctica 1

Ricardo Andrés Calvo Méndez, Jorge Aurelio Morales Manrique

*Universidad Nacional de Colombia  
Bogotá, Colombia*

`rcalvom@unal.edu.co  
jomorales@unal.edu.co`

27 de junio de 2022

**Abstract**—Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. This technique is related to the field of parallel computing in the sense that the filters used are highly parallelizable, the pixels transformations are independent of each other. The purpose of this practice is to apply a filter to images of different sizes and analize the response time and speedup using different numbers of threads.

## I. INTRODUCCIÓN

La computación paralela es un paradigma de cómputo en el cual múltiples cálculos o procesos son ejecutados simultáneamente. Problemas grandes pueden ser divididos en problemas más pequeños que pueden ser ejecutados en diferentes unidades lógicas del procesador para obtener tiempos de ejecución más bajos. La computación paralela es ampliamente usada en el campo de “high performance computing” gracias a las ventajas que ofrece en la ejecución de algoritmos que cumplen con la condición de ser paralelizables. Una de las aplicaciones más reconocidas es el procesamiento de imágenes, por medio de la aplicación de filtros basados en kernels, debido a que las operaciones realizadas sobre los píxeles son independientes entre si, lo que permite hacer una división del cómputo entre varios hilos. El objetivo de esta práctica es aplicar

el filtro de detección de bordes sobre imágenes con resoluciones 720p, 1080p y 4k, por medio de una implementación en lenguaje C que haga uso de la librería “`pthread.h`” para la ejecución con múltiples hilos, la interfaz de programación de aplicaciones multiproceso de memoria compartida OpenMP, la plataforma de computación paralela CUDA y la interfaz basada en paso de mensajes OpenMPI. Finalmente se realiza el respectivo análisis de tiempo de respuesta y speedup para los diferentes tamaños de imagen y los diferentes paradigmas (posix, OpenMP, CUDAy OpenMPI).

## II. DISEÑO DEL PROGRAMA

La aplicación tiene 3 componentes principales:

- **Script de Bash:** realiza la ejecución del algoritmo en C para cada una de las imágenes de prueba variando el número de hilos con valores de 1, 2, 4, 8 y 16. Al finalizar cada caso almacena el tiempo de ejecución en un archivo de texto plano. Una vez se obtienen los resultados para cada caso de prueba se ejecuta el script en Python para la generación de las gráficas y el reporte.
- **Script en Python:** realiza la lectura de los archivos de resultados, agrupando los valores por tipo de imagen, para luego, haciendo uso de la librería **matplotlib** generar las gráficas correspondientes al tiempo de respuesta y speedup para

las imágenes con resolución 720p, 1080p y 4k. Así mismo, el script compila el reporte latex con la última versión de gráficas generadas.

- **Algoritmo en C:** contiene la implementación del filtro para detección de bordes. El kernel utilizado es el siguiente:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

A continuación se muestran fragmentos de pseudocódigo para las implementaciones usando posix y OpenMP:

```

function rgb_to_grayscale(r, g, b)
    Return 0.299 * r + 0.587 * g + 0.114 * b
end function

function grayscale_filter_thread(input)
    For i = input.start to input.end - 1
        input.grayscale_image[i] = rgb_to_grayscale(
            input.input_image[i * 3],
            input.input_image[i * 3 + 1],
            input.input_image[i * 3 + 2]
        )
    Endfor
end function

function border_detection_filter_thread(input)
    For i = input.start to input.end - 1
        input.output_image[i] = input.filter_intensity *
            (pixel_block_gsi * kernel)
    Endfor
end function

function border_detection_filter(input_filename,
    |output_filename, filter_intensity, threads_count)
    # Reading image
    input_image <- stbi_load(input_image)
    # Grayscale filter
    For i = 0 to threads_count - 1
        pthread_create(i, grayscale_filter_thread)
    Endfor
    For i = 0 to threads_count - 1
        pthread_join(i)
    Endfor
    # Border detection filter
    For i = 0 to threads_count - 1
        pthread_create(i, border_detection_filter_thread)
    Endfor
    For i = 0 to threads_count - 1
        pthread_join(i)
    Endfor
    # Saving image
    stbi_write_jpg(output_filename)
end function

```

Figura 1. Pseudocódigo algoritmo para detección de bordes usando posix

Las funciones “rgb\_to\_grayscale” y “grayscale\_filter\_thread” corresponden a la conversión de cada pixel de RGB a escala de grises, esto se realiza ya que para la detección de bordes no es necesario conocer los valores de color de los 3 canales, se requiere conocer la intensidad del pixel, por lo cual se usa la escala de grises. Las funciones “border\_detection\_filter\_thread” y “border\_detection\_filter” corresponden a la

aplicación del filtro de detección de bordes para el cual el valor de cada pixel se calcula a través de una operación de convolución entre la matriz 3x3, que corresponde al pixel actual y sus vecinos, y el kernel.

```

1  function rgb_to_grayscale(r, g, b)
2      Return 0.299 * r + 0.587 * g + 0.114 * b
3  end function
4
5  function border_detection_filter(input_filename,
6      |output_filename, filter_intensity, threads_count)
7      # Reading image
8      input_image <- stbi_load(input_filename)
9      # Grayscale filter
10     grayscale_image <- malloc(image_size)
11     data <- thread_input[threads_count]
12     For i = 0 to threads_count - 1
13         data[i].start = image_size / threads_count * i
14         data[i].end = image_size / threads_count * (i + 1)
15     Endfor
16     #pragma omp parallel num_threads(threads_count)
17     id = omp_get_thread_num()
18     For i = data[id].start to data[id].end - 1
19         grayscale_image[i] = rgb_to_grayscale(
20             input_image[i * 3],
21             input_image[i * 3 + 1],
22             input_image[i * 3 + 2]
23         )
24     Endfor
25     # Border filter
26     output_image = malloc(image_size)
27     #pragma omp parallel num_threads(threads_count)
28     id = omp_get_thread_num()
29     For i = data[id].start to data[id].end - 1
30         output_image[i] = filter_intensity *
31             (pixel_block_gsi * kernel)
32     Endfor
33     # Saving image
34     stbi_write_jpg(output_filename, output_image)
35     free()

```

Figura 2. Pseudocódigo algoritmo para detección de bordes usando OpenMP

A diferencia del pseudocódigo anterior, en este caso se hace uso de dos funciones principales. En primer lugar “rgb\_to\_grayscale” que al igual que antes, tiene como propósito convertir un pixel de RBG a escala de grises. La segunda función realiza la conversión de la imagen a escala de grises para luego aplicar el filtro de detección de bordes con la operación de convolución para cada pixel.

Las funciones para los filtros de escala de grises y detección de bordes son ejecutados utilizando múltiples hilos de acuerdo al argumento que indique el usuario. Para realizar la distribución de datos o balanceo de carga se utiliza el método clockwise (división de los datos en segmentos iguales), el cual se describe en el siguiente diagrama:

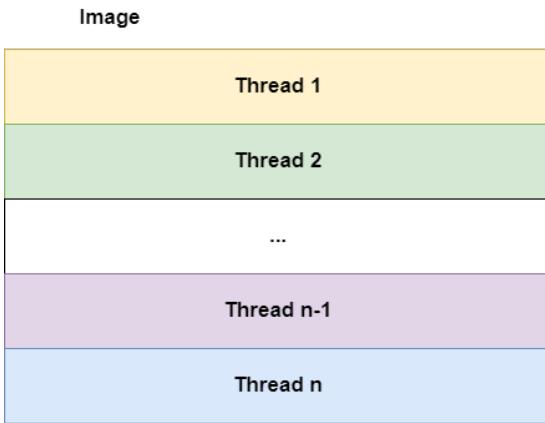


Figura 3. Método clockwise para balanceo de carga

Así mismo en el caso de CUDA, se tiene el siguiente balanceo, teniendo en cuenta bloques e hilos:

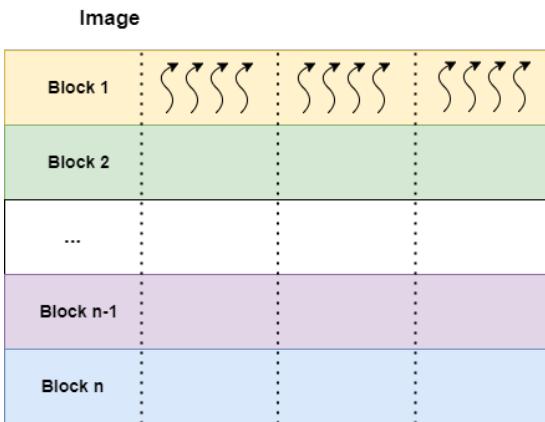


Figura 4. Balanceo de carga CUDA

Para el caso de OpenMPI se tiene la siguiente estructura:



Figura 5. Balanceo de carga OpenMPI

### III. EXPERIMENTOS

Para la ejecución del programa, se han escogido 3 imágenes diferentes con tamaños 4k, 1080p y 720p; a cada una de las imágenes les será aplicado el filtro de detección de bordes usando ejecución secuencial y ejecuciones paralelas de 2, 4, 8 y 16. Combinando los tres tamaños de las imágenes y los cuatro tipos de ejecución, en total serán 12 casos de prueba. Para el caso de CUDA se aumenta el número de hilos a 32, 64 y 128 y se manejan número de bloques entre 1, 2, 4, 8, 16, 32 y 64, lo cual representa 56 casos de prueba por tipo de imagen.

Las siguientes son las imágenes a las cuales se les va a aplicar el filtro:



Figura 6. Miniatura de imagen con tamaño 4k que va a ser procesada



Figura 7. Miniatura de imagen con tamaño 1080p que va a ser procesada



Figura 8. Miniatura de imagen con tamaño 720p que va a ser procesada

Para la automatización de la ejecución de los anteriores casos de prueba se utiliza el bash script *execute\_script.sh* que ejecutará cada uno de los casos y almacenará los resultados en archivos de texto plano para su posterior análisis.

#### IV. RESULTADOS

Para la ejecución de los casos de prueba mencionados anteriormente se ha utilizado un procesador AMD Ryzen 5 4500U el cuál cuenta con 6 núcleos a 2.3 GHz de frecuencia de reloj en el caso de pthread y OpenMP, en cuanto a CUDA se hizo uso de un entorno de ejecución en Google Colab con una GPU de NVIDIA de las siguientes características:

CUDA Device Query (Runtime API) version (CUDART static linking)	
Detected 1 CUDA Capable device(s)	
Device 0: "Tesla T4"	
CUDA Driver Version / Runtime Version	11.2 / 11.0
CUDA Capability Major/Minor version number:	7.5
Total amount of global memory:	15360 MB/bytes (15843721216 bytes)
Number of multiprocessors, (64) CUDA Cores/MP:	2560 CUDA Cores
GPU Max Clock rate:	1590 MHz (1.59 GHz)
Memory Clock rate:	5081 MHz
Memory Bus Width:	256-bit
L2 Cache Size:	4194304 bytes
Maximum Texture Dimension Size (x,y,z):	1D-(131072), 2D-(121072, 65536), 3D-(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers:	1D-(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers:	2D-(32768, 32768), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	65536
Warp size:	32
Maximum number of threads per multiprocessor:	1024
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 493, 64)
Max dimension size of a grid size (x,y,z):	(16384, 48824, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 3 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Supports host page-locked memory mapping:	Yes
Alignment requirement for surfaces:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Device supports Managed Memory:	Yes
Device supports Compute Memory:	Yes
Supports Cooperative Kernel launch:	Yes
Supports Multidevice Co-op Kernel Launch:	Yes
Device PCI Domain ID / Bus ID / location ID:	0 / 0 / 4
Compute Mode:	< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Figura 9. Arquitectura de la GPU de NVIDIA

Los resultados obtenidos de ejecutar el filtro en las 3 imágenes mencionadas anteriormente usando posix, OpenMP y CUDA son los siguientes:

##### IV-A. Imagen de tamaño 4k

La imagen obtenida de aplicar el filtro es la siguiente:



Figura 10. Miniatura de imagen con tamaño 4k con el filtro de detección de bordes aplicado usando posix.

Las gráficas que resumen y comparan los tiempo de respuesta y los speedup de la aplicación del filtro a esta imagen son los siguientes (en el caso de CUDA cada gráfica tiene múltiples líneas cada una indicando un número diferente de bloques):

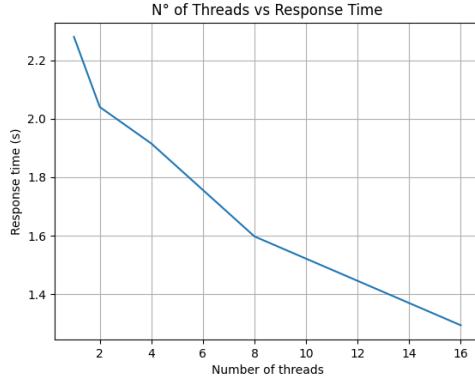


Figura 11. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando posix

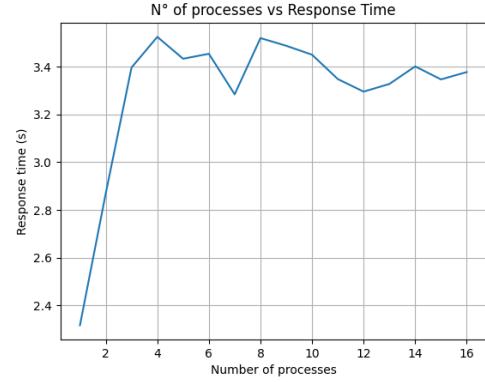


Figura 14. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando OpenMPI

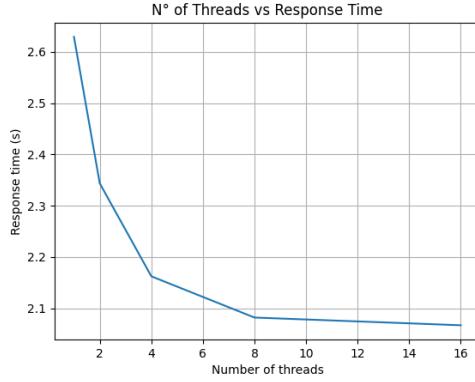


Figura 12. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando OpenMP

En este caso se puede observar como los tiempos de ejecución son siempre decrecientes, a pesar de tener únicamente 6 núcleos, los tiempos se redujeron inclusive usando 16 hilos con lo que se puede inferir que los tiempos del sistema operativo pudieron interferir positivamente en estos últimos casos. Sin embargo es bastante notorio como el uso de múltiples hilos para el procesamiento de la imagen, redujeron los tiempos de procesamiento desde un poco mas de 2.2 segundos hasta 1.3 segundos en el mejor caso.

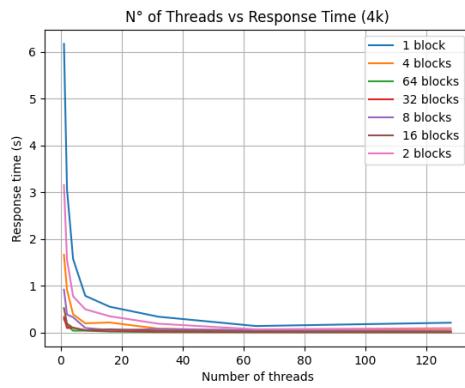


Figura 13. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando CUDA

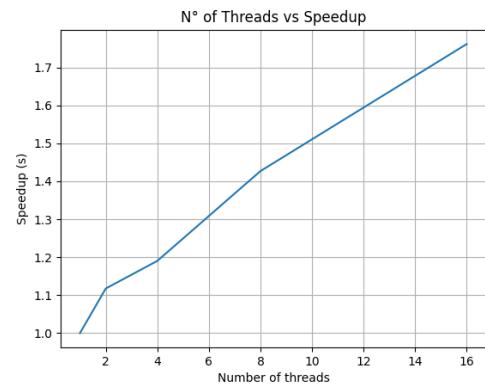


Figura 15. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando posix

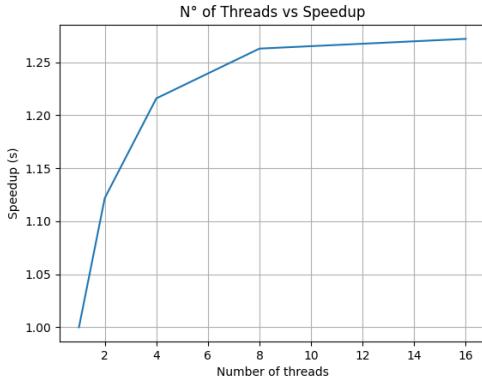


Figura 16. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando OpenMP

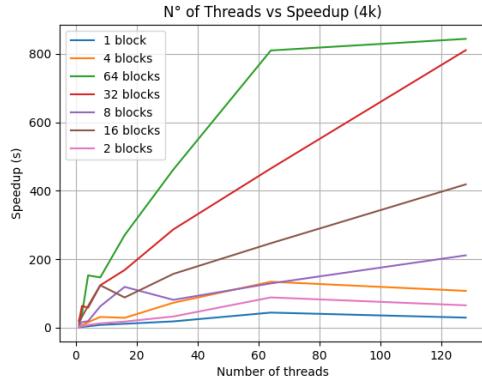


Figura 17. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando CUDA

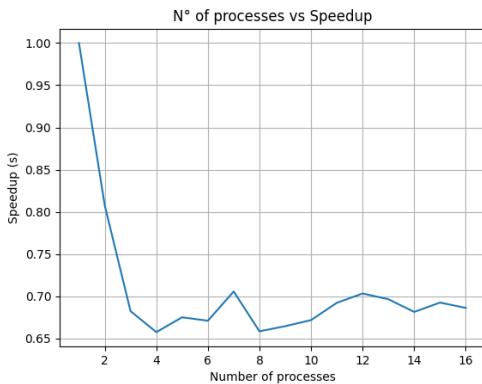


Figura 18. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 4k usando OpenMPI

En el caso del speed up, se puede notar como este siempre es positivo mostrando una notoria mejora cuando se aumenta el uso de núcleos para el procesamiento de la imagen.

#### IV-B. Imagen de tamaño 1080p

La imagen obtenida de aplicar el filtro es la siguiente:



Figura 19. Miniatura de imagen con tamaño 1080p con el filtro de detección de bordes aplicado usando posix.

Las gráficas que resumen y comparan los tiempo de respuesta y los speed up de la aplicación del filtro a esta imagen son los siguientes:

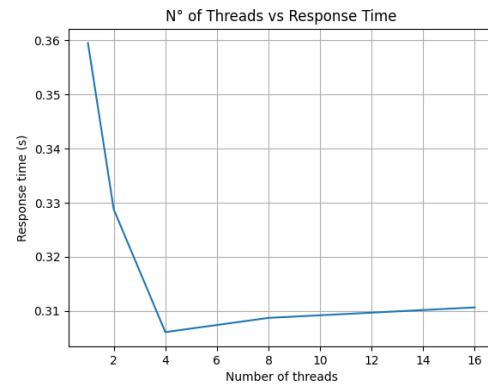


Figura 20. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando posix

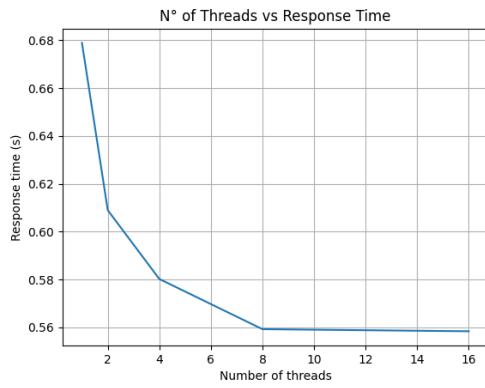


Figura 21. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando OpenMP

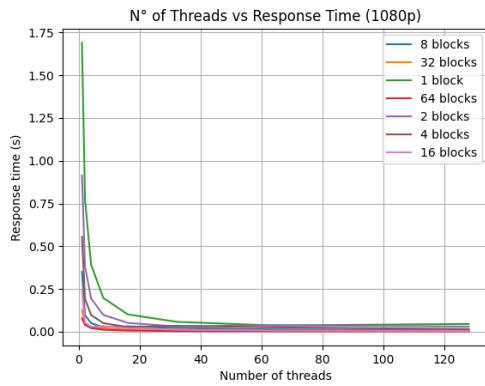


Figura 22. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando CUDA

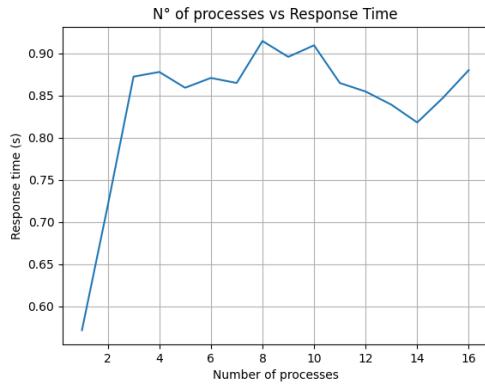


Figura 23. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando OpenMPI

En este caso llama la atención como en los casos de procesamiento con 8 y 16 núcleos los tiempos crecieron, esto se puede justificar en los 6 núcleos del procesador, en estos casos solo 6 hilos se ejecutaban al mismo tiempo mientras que los demás trabajaban su parte del filtro de manera concurrente.

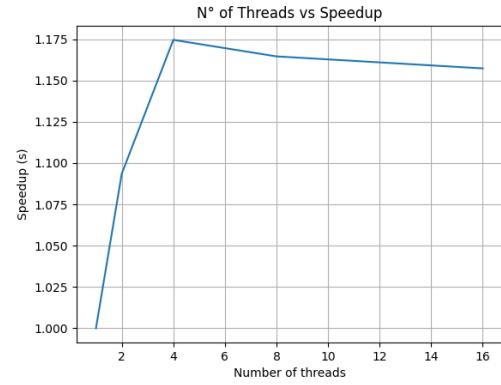


Figura 24. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando posix

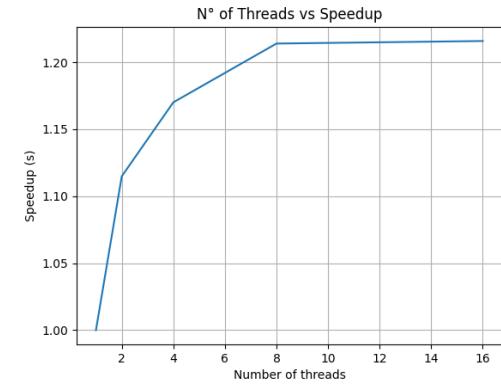


Figura 25. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando OpenMP

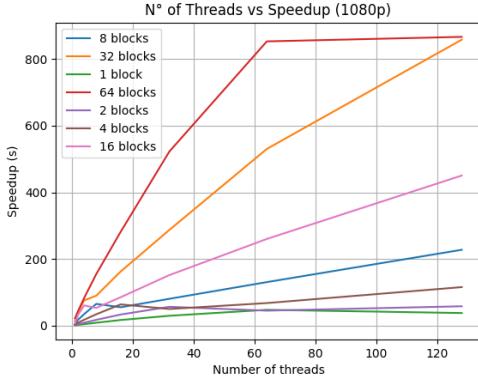


Figura 26. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando CUDA



Figura 28. Miniatura de imagen con tamaño 720 con el filtro de detección de bordes aplicado usando posix.

Las gráficas que resumen y comparan los tiempo de respuesta y los speed up de la aplicación del filtro a esta imagen son los siguientes:

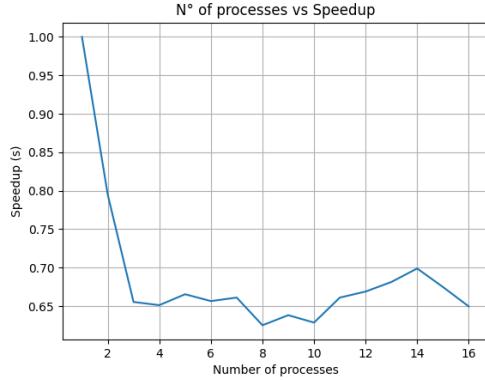


Figura 27. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 1080p usando OpenMPI

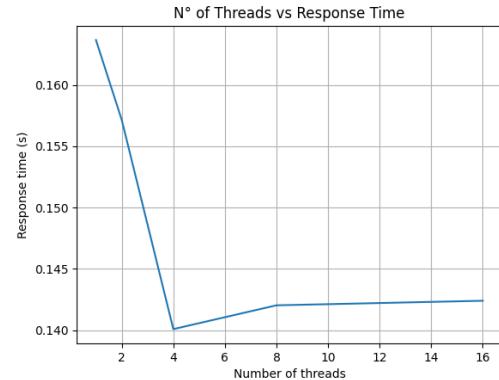


Figura 29. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando posix

De manera similar a los tiempos de respuesta, el resultado para 8 y 16 hilos es irregular y se puede asumir esto debido a los 6 núcleos del procesador. Algo a tener en cuenta es que el speed up en el caso de la imagen 4k fue muy similar al caso de 720p, esto se puede justificar en que independiente mente del tamaño de la imagen siempre se va a estar paralelizando la misma proporción de código.

#### IV-C. Imagen de tamaño 720p

La imagen obtenida de aplicar el filtro es la siguiente:

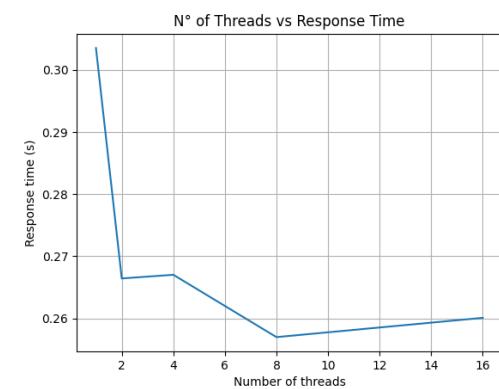


Figura 30. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando OpenMP

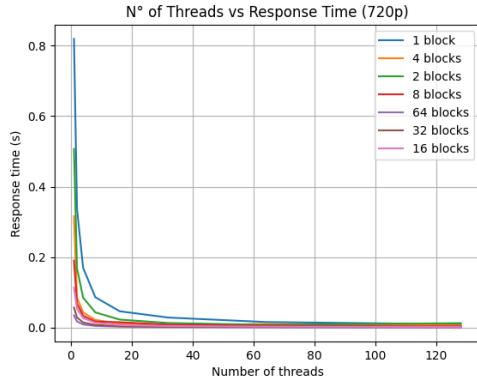


Figura 31. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando CUDA

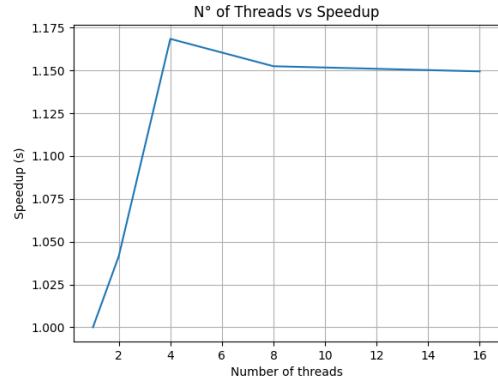


Figura 33. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando POSIX

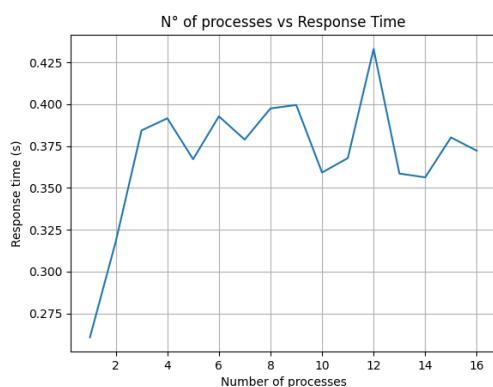


Figura 32. Gráfica de tiempos de respuesta para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando OpenMPI

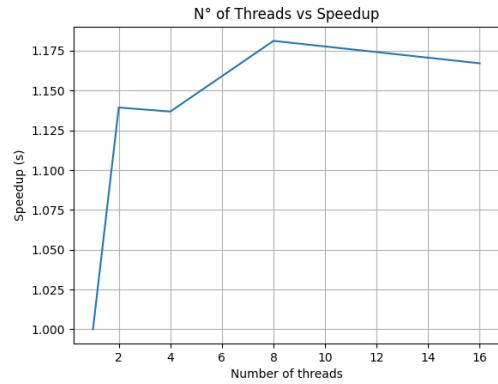


Figura 34. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando OpenMP

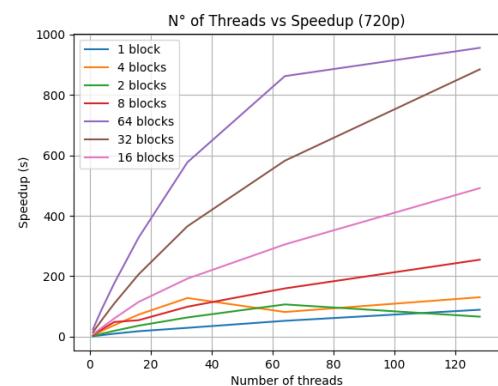


Figura 35. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando CUDA

En este caso nuevamente tenemos una irregularidad en las ejecuciones con 8 y 16 hilos que se justifican con los 6 núcleos del procesador, los tiempos de ejecución total del programa para tamaño 720p están en el orden de 100 ms, por lo que los tiempos del sistema operativo pueden estar interfiriendo con los resultados.

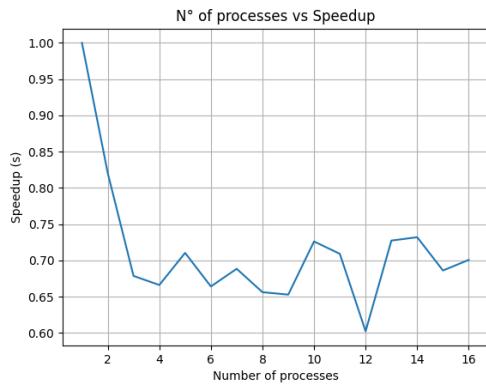


Figura 36. Gráfica de speedup para las ejecuciones de la aplicación del filtro sobre la imagen de tamaño 720p usando OpenMPI

Finalmente en este caso encontramos la irregularidad que ya se mencionó en las ejecuciones con 8 y 16 hilos, además notamos que nuevamente el speed up en los casos de 4 núcleos es muy similar al speed up en los casos de 4k y 1080p debido a que se paralleliza la misma proporción de código.

## V. CONCLUSIONES

- La aplicación de filtros sobre imágenes son procesos altamente paralelizables debido a que cada proceso realizado sobre cada pixel es independiente de los demás.
- Las gráficas de speed up permiten identificar el límite de eficiencia que se puede alcanzar al parallelizar el algoritmo de detección de bordes. Estos resultados están sustentados bajo la ley de Amdahl, la cual establece un límite de speed up a medida que se aumenta el número de hilos usados en la ejecución.
- Los tiempos del sistema operativo afectan considerablemente los tiempos de ejecución de un programa cuando sus tiempos son muy cortos.
- OpenMP es una interfaz de programación que simplifica en gran medida el desarrollo de programas multihilo por medio de directivas y un conjunto de funciones. Es una herramienta muy útil que acelera el desarrollo y abstrae elementos de diseño del programa por medio de múltiples elementos en el core de la librería.
- Los tiempos de ejecución usando POSIX y OpenMP no varian significativamente, sin embargo, se pudo observar en los resultados que el pro-

grama ejecutado con OpenMP presenta tiempos menores del orden de milesimas de segundo.

- CUDA provee una interfaz de programación paralela la cual presenta múltiples opciones para adaptar algoritmos que requieren un número elevado de cálculos para obtener un resultado, con las pruebas realizadas se demostró que al hacer uso de una GPU con un número significativo de cores representa un speedup elevado en comparación a los otros 2 paradigmas utilizados.
- Para el propósito de la práctica se encontró que alternativas de memoria compartida como CUDA y haciendo uso de hardware especializado, presenta una ventaja en comparación a arquitecturas distribuidas como de pudo observar con los resultados de OpenMPI.