Reporte Práctica 1 Computación Paralela

Diego Alejandro Bayona Cardozo, Gabriel Pérez Santamaría
Facultad de Ingeniería, Universidad Nacional de Colombia
Bogotá, Colombia
dbayonac@unal.edu.co
gaperezsa@unal.edu.co

***Abstract*— Este documento es un ejemplo de formato apegado a las normas de IEEE para escribir artículos representativos de un proyecto realizado. Los autores deben seguir las instrucciones, incluyendo formato y tamaño de papel para mantener el estándar de publicación. Este documento puede interpretarse como un set de instrucciones para escribir su artículo o como una plantilla para hacerlo. Como habrá notado, esta primera sección es para generar un resumen muy corto y a alta escala del alcance del proyecto.**

I. INTRODUCCIÓN

Este documento es un reporte de la práctica 1 de la materia Computación Paralela y Distribuida, en la cual se desarrollara la aplicación de un filtro o filtros sobre una imagen y paralelizar el proceso mediante POSIX. Gracias a librerías de manipulación de imágenes como opencv se pueden fabricar los filtros de manera secuencial píxel por píxel de la imagen. Este proceso, dependiendo del tamaño de la imagen a procesar y el filtro a aplicar, puede generar una alta carga computacional a nuestro procesador, es por esto que se evaluará el impacto de la paralelización de estos algoritmos.

II. DISEÑO DE FILTROS Y PARALELIZACIÓN

A partir de esta sección, se desarrollan los contenidos del tema, de una forma ordenada y secuencial. Nótese que la sección debe ir organizada usando títulos como el anterior para cada tema nuevo incluido. Aparte, se incluyen subtítulos como el siguiente.

A. Elección de filtros

Se aplicarán 4 filtros de paso a escalas de gris, esto es, se tomarán 3 imágenes a color (720 p, 1080p y 4K) y se le aplicarán 4 filtros/algoritmos que producirán diferentes candidatos de equivalentes en escalas de grises de un solo canal.

Los 4 algoritmos a utilizar serán:

1. Promedio RGB
2. Luma
3. Capas de Gris
4. Granular/ruido

B. Explicación sobre los filtros

Los filtros de paso a escala de grises siguen 3 pasos en general:

1. Descomponer y guardar los valores RGB de un pixel
2. Usar fórmulas matemáticas para convertir dichos valores en un único valor de gris.
3. Reemplazar en la nueva imagen el valor calculado en los tres canales o en el único canal de salida (dependiendo si la salida de la imagen es RGB o de canal único).

En nuestro caso, la imágen de salida es de canal único, por lo que simplemente se colocará en valor calculado en ese píxel. A la hora de explicar brevemente los filtros escogidos se dará énfasis al paso 2, es decir, las fórmulas matemáticas que retornan el valor de gris para cada pixel.

El primer filtro es el más básico de todos (también el más rápido y "sucio"), el método promedio. Como su nombre lo indica, se promedian los tres valores de los canales RGB para devolver ese valor por un único canal:

$$Gray = (Red + Green + Blue) / 3$$

El segundo filtro es el método luma o luminance, que es utilizado debido a que se considera que la densidad del cono del ojo humano no es uniforme en todos los colores, entonces lo que se ejecuta es una conversión que permita distinguir mejor ciertos canales, de la siguiente manera:

$$Gray = (Red * 0.2126 + Green * 0.7152 + Blue * 0.0722)$$

El tercer filtro es un número personalizado de capas de grises, este recibe como parametro el numero de capas que deseamos tener en la imagen para que, dado un valor inicial de gris (puede tomarse un valor inicial desde promedio o luma), se haga un redondeo hacia la capa más cercana posible de la siguiente manera:

$$ConversionFactor = 255 / (NumberOfShades - 1)$$
$$AverageValue = (Red + Green + Blue) / 3$$
$$Gray = Integer((AverageValue / ConversionFactor) + 0.5) * ConversionFacto$$

El cuarto filtro es un número personalizado de capas de gris con dithering o granularidad, el cual consiste en agregar al factor de conversión un "error acumulativo", dependiendo su implementación este error se puede notar de distinta manera:

$$greyTempCalc = promedio(RGB)$$
$$greyTempCalc \mathrel{+}= Error$$
$$greyTempCalc = Integer((greyTempCalc / ConversionFactor) + 0.5) * ConversionFac$$
$$Error = promedio(RGB) + Error - greyTempCalc$$
$$if\ greyTempCalc > 255: Gray = 255$$
$$else\ if\ greyTempCalc < 0: Gray = 0$$
$$else\ Gray = greyTempCalc$$

C. Modelo de paralelización

El modelo escogido será por partición de datos *Blockwise*; es decir, siendo una imagen una matriz de píxeles, se ejecutarán los algoritmos recorriendo de izquierda a derecha-arriba a abajo cada bloque.

Un bloque se define como *#de filas / #hilos* y cada bloque será consecutivo al anterior sin sobrelape. Dada esta partición de los datos, cada hilo invocado se encargará de aplicar el filtro escogido a cada bloque de la imágen. Un esquema de este proceso se muestra continuación con una imagen hipotética de 1080p y N hilos invocados:

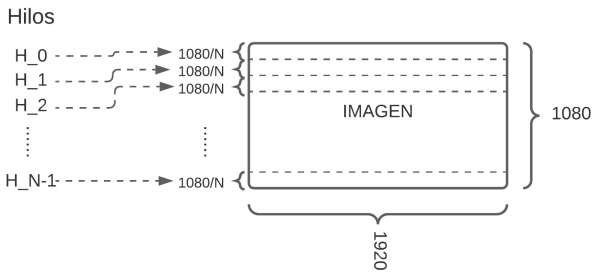


Fig. 1 Esquema modelo de paralelización.

III.PROGRAMAS,EXPERIMENTOS Y RESULTADOS

La máquina en la que se va a ejecutar el programa resultante se encuentra corriendo Ubuntu 20.04 de forma nativa, 16GB de RAM (3600MHz DDR4) y una CPU Ryzen 7 3700 X :

-8 núcleos
-16 hilos
-Reloj base 3,6 GHz
-Overclock a 4,4 GHz
-Configurada para correr a 4,2GHz promedio
-L1 de 512KB
-L2 de 4MB
-L3 de 32MB

Se instalaron las dependencias necesarias que no hacen parte de la librerías básicas de C/C++, es decir, OpenCV. Para la instalación de OpenCV se tuvo en cuenta que se debía hacer uso de CMake entre otras dependencias y así compilar la última versión de la librería en Ubuntu 20.04.

Una vez instalada la librería, se observó que la mayoría de la documentación oficial está basada en C + + por lo que se decidió realizar los filtros en este lenguaje.

El programa resultante tiene la siguiente sintaxis de compilación:

```
g++ main.cpp -o filtros -lpthread `pkg-config --cflags --libs opencv4`
```

Y la siguiente sintaxis para su ejecución y los argumentos de entrada:

```
./filtros 1080p.jpg 1080p_withfilter2.jpg 2 4
./filtros 1080p.jpg 1080p_withfilter3.jpg 3 3 4
```

El primer parámetro (en este caso 1080p.jpg) hace referencia al nombre de la imagen que se encuentra en la misma carpeta a la cual se le quiere aplicar el filtro.

El segundo parámetro (en este caso 1080p_withfilter2.jpg o 1080p_withfilter3.jpg) hace referencia al nombre de la imagen transformada que se tendrá como salida de la ejecución. Si existe una imagen con este nombre, se sobreescribirá, si no existe una imagen con este nombre, se creará.

El tercer parámetro (en este caso 2) hace referencia al filtro que se aplicará, 1 = sucio, 2 = Luma, 3 = capas de gris, 4 = granular.

El cuarto parámetro y el posible quinto dependerá del filtro que se haya escogido, por eso se dan dos ejemplos en la foto anterior. Si el filtro escogido fue el 1 o el 2, el cuarto parámetro será la cantidad de hilos con los que se quiere que se ejecute el programa. Si el filtro escogido fue el 3 o el 4, el cuarto parámetro será la cantidad de capas de gris que se quiere usar para el filtro (número entre 0-255 inclusive, cualquier número mayor a 255 será equivalente a 255 y el análogo con los números menores a 0). El quinto parámetro será la cantidad de hilos con los que se quiere ejecutar el programa.

Es decir, el primer ejemplo corre el filtro 2 con 4 hilos mientras que el segundo ejemplo corre el filtro 3 usando 3 capas de gris y 4 hilos.

Se programó un script en Python encargado de correr el programa múltiples veces aplicando todos los filtros a todas las imágenes con 1(secuencial), 2, 4, 8 y 16 hilos

El script de ejecución de los filtros se encarga de correr el programa múltiples veces con los diferentes formatos: 720p, 1080p y 4K. El script asume que la foto de 720 píxeles de resolución está guardada con el nombre 720p.jgp, y así sucesivamente para 1080p y 4K.

El script recorrerá el filtro aplicado, la cantidad de hilos utilizados y la imágen, al finalizar cada ejecución se guardará una imagen por cada uno de los filtros aplicados con la sintaxis:

{resolución}_withfilter{#filtro}.jpg

por ejemplo:

720p_withfilter2.jpg

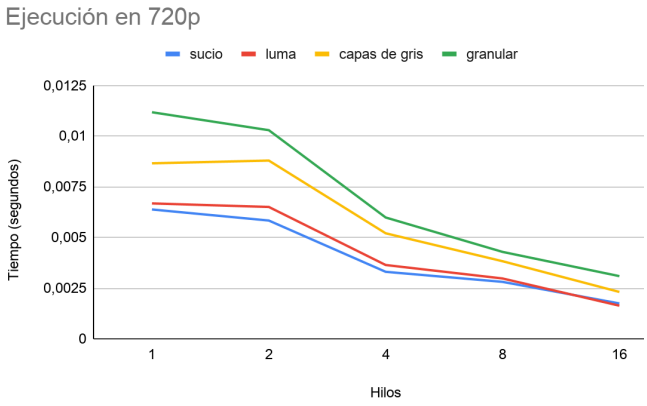
Este script además se encarga de leer la salida por consola de la ejecución de nuestro programa para así poder almacenarlo en un archivo de texto plano en formato csv (comma separated values) que nos facilita la lectura de los datos de los tiempos de ejecución de los programas. Con el objetivo de obtener tiempos de ejecución más confiables, el script correrá cada combinación 5 veces, guardando el tiempo de salida en un arreglo y finalmente escribiendo el promedio. Se tomaron algunas libertades en los casos de los filtros que requerían usar un número de capas usando siempre el valor: 3 (se puede cambiar el valor de la variable “capas” en la declaración inicial dentro del script). La sintaxis para la ejecución del script es:

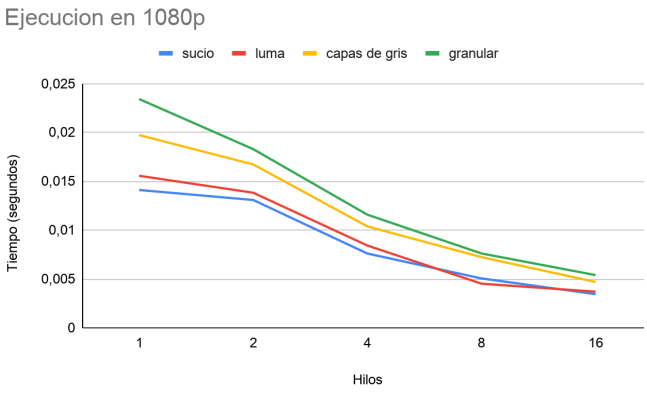
```
./script_ejecutar_todo.py
```

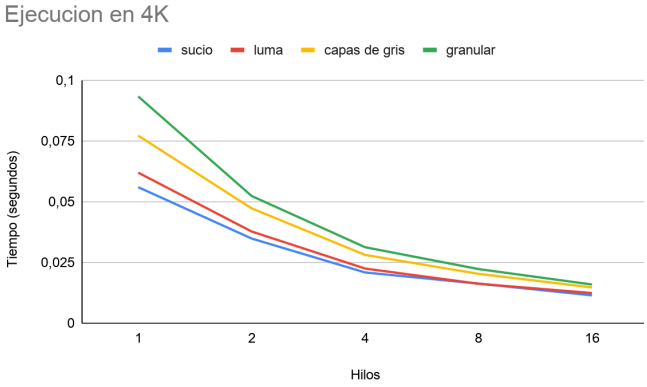
Si el script no posee permisos ejecutar el comando:

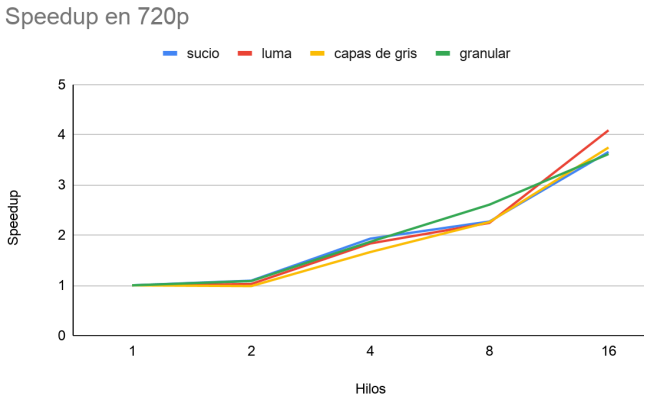
```
chmod +x script_ejecutar_todo.py
```

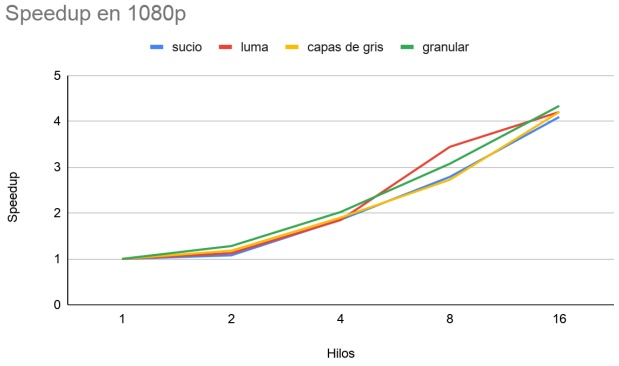
Ya finalizado el script, se guarda toda la información de los tiempos de ejecución en el archivo “data.csv” , al importar este archivo en Excel graficamos los tiempos de ejecución de cada filtro dependiendo de la cantidad de hilos, y su respectivo *speedup*. Estos son los resultados:

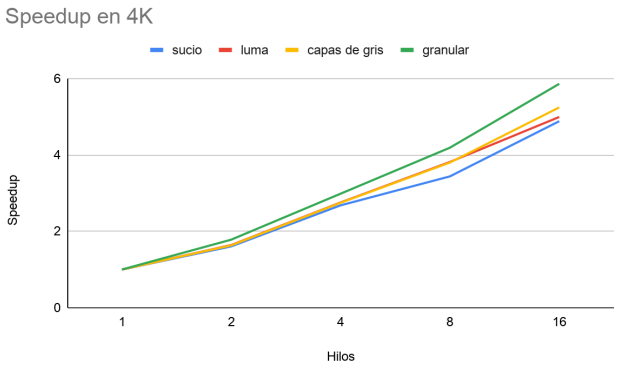












IV.CONCLUSIONES

Como se puede observar en las gráficas de ejecución, los filtros que necesitan de más operaciones por pixel efectivamente tardan más tiempo; por ejemplo, la aplicación del filtro de granularidad siempre fue el que más tardó de los 4 aplicados.

La paralelización del trabajo bajo el esquema escogido efectivamente reduce nuestros tiempos de ejecución por factores pertinentes. Como nuestros speedup indican, cuando se paraleliza con el uso de 16 hilos, algunos filtros podían llegar a correrse 4 o 5 veces más rápido que sus equivalentes secuenciales.

Yendo a más detalle, se puede notar en las gráficas correspondientes a las imágenes 720p y 1080p la forma en que el paso del programa secuencial al uso de 2 hilos no proporciona mejoras significativas, los speedup resultan siendo de factores entre 1 y 1,3. Las estructuras a declarar, la creación de hilos, la sincronización/tiempos de comunicación de los dos hilos resultan muy grandes en relación a la partición de trabajo y ahorro de tiempo de ejecución que pretenden, resultando en tiempos de ejecución muy similares al programa secuencial original. Para la imágen 4K sucede el mismo fenómeno, sin embargo, el speedup resulta un poco mayor, de aproximadamente 1,6-1,7; Esto debido a que la cantidad de trabajo aumentada contrarresta un poco los tiempos de declaración, creación y sincronización de los que se hablaba.

REFERENCIAS

[1] Wikipedia, “Grayscale”, 2021 [Online]. Available:, https://en.wikipedia.org/wiki/Grayscale

[2] T. Helland, “Seven grayscale conversion algorithms (with

pseudocode and VB6 source code)", 2011, [Online]. Available: https://tannerhelland.com/2011/10/01/grayscale-image-algorithm-vb6.html