Reporte Prácticas 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

I. INTRODUCCIÓN

Este documento es un reporte de las diferentes prácticas de la materia Computación Paralela y Distribuida, en el cual se desarrollará la aplicación de un filtro o filtros sobre una imagen y paralelizar el proceso mediante diferentes métodos en cada una de las prácticas. 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

Práctica 1: POSIX

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 escala 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:

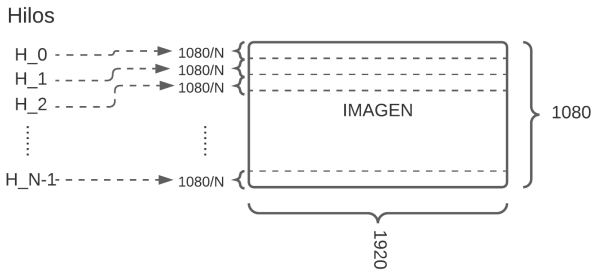


Fig. 1 Esquema modelo de paralelización POSIX/OMP.

Práctica 2: OMP

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

Los 4 algoritmos a utilizar serán:

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

B. Explicación sobre los filtros

La explicación y aplicación es igual que en la primera práctica de POSIX

C. Modelo de paralelización

El modelo de paralelización es igual que en la primera práctica de POSIX

Práctica 3: CUDA

A. Elección de filtros

Se aplicarán 3 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

Los 3 algoritmos a utilizar serán:

1. Promedio RGB
2. Luma
3. Capas de Gris

B. Explicación sobre los filtros

La explicación y aplicación es igual que en la primera práctica de POSIX

C. Modelo de paralelización

El modelo de paralelización es distinto al utilizado en POSIX y OMP debido a que la tecnología nos brinda muchos recursos en cuanto que tanto podemos paralelizar.

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 *Cantidad de Pixeles / #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 de Ancho * Alto Pixeles y N hilos invocados:

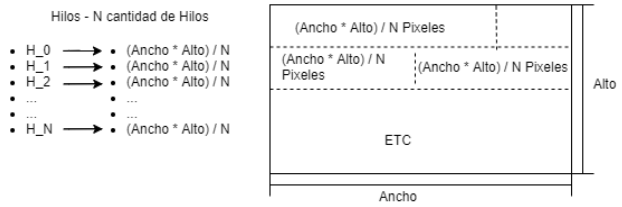


Fig. 2 Esquema modelo de paralelización CUDA/MPI.

Práctica 4: MPI

A. Elección de filtros

Se aplicarán 3 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

Los 3 algoritmos a utilizar serán:

4. Promedio RGB
5. Luma
6. Capas de Gris

B. Explicación sobre los filtros

La explicación y aplicación es igual que en la primera práctica de POSIX

C. Modelo de paralelización

El modelo de paralelización es igual que en la tercera práctica de CUDA

III. PROGRAMAS,EXPERIMENTOS Y RESULTADOS

A. Hardware

Practica 1,2,3: POSIX, OMP y CUDA

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

Práctica 4: MPI

Se va a ejecutar el programa resultante en GCP (Google Cloud Platform) con diferentes Compute Engines e2.micros que se encuentra corriendo Ubuntu 20.04 de forma nativa, 1GB de RAM y 2 CPU virtuales.

B. Librerías y Compilación

Practica 1,2,3,4: POSIX, OMP, CUDA y MPI

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:
POSIX y OMP:

```
g++ image-effects.cpp -o filtros -lpthread -fopenmp `pkg-config --cflags --libs opencv4`
```

CUDA:

```
nvcc image-effects.cu -o filtros `pkg-config --cflags --libs opencv4`
```

MPI:

```
sudo mpic++ image-effects-mpi.cpp -o filtros_mpi -fopenmp `pkg-config --cflags --libs opencv4`
```

C. Ejecución

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

POSIX y OMP:

```
./filtros [nombre_src] [nombre_dst] [filtro] [parametros de filtro] [num_hilos]
```

El primer parámetro (nombre_src) 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 (nombre_dst) 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 (filtro) 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.

CUDA:

```
./filtros [nombre_src] [nombre_dst] [filtro] [parametros de filtro] [num_hilos] [num_bloques]
```

Los parámetros son iguales que en POSIX y OMP a excepción del último que indica la cantidad de bloques que se quieren lanzar.

MPI:

```
mpirun -np [num_nodes] --hostfile mpi_hosts ./filtros_mpi [nombre_src]
[nombre_dst] [filtro] [parametros de filtro]
```

Los parámetros son iguales que en POSIX y OMP a excepción del último que la cantidad de procesos a lanzar o hilos se da antes en el parámetro “num_nodes”, y en el archivo “mpi_hosts” se deben especificar las direcciones IP de los nodos disponibles.

D. Script de Ejecución

Se programó un script en Python para cada una de las prácticas 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:
POSIX y OMP:

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

CUDA:

```
chmod +x scrip_cuda.py
./scrip_cuda.py
```

MPI:

```
chmod +x script_mpi.py
./script_mpi.py
```

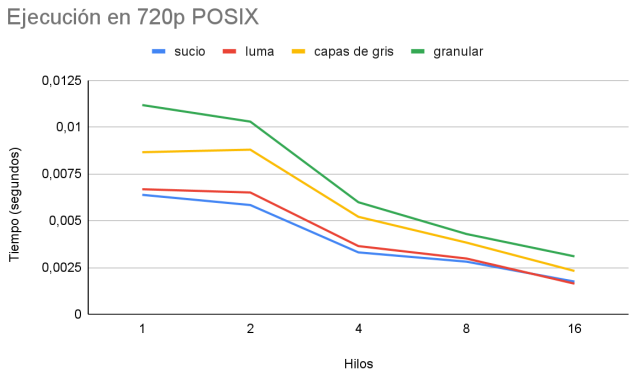
Ya finalizado el script, se guarda toda la información de los

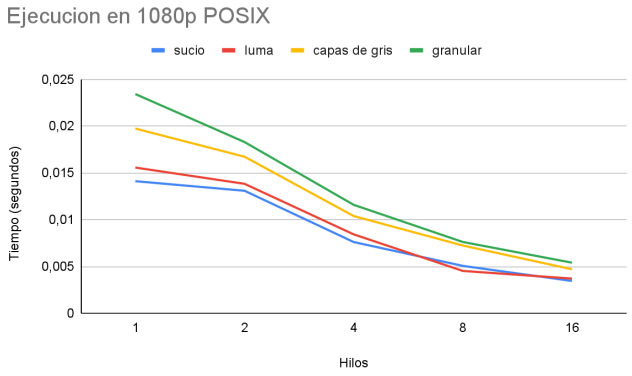
tiempos de ejecución en un archivo “.csv” en la carpeta data_csv , al importar este archivo en Excel graficamos los tiempos de ejecución de cada filtro dependiendo de la cantidad de hilos, y su respectivo *speedup*.

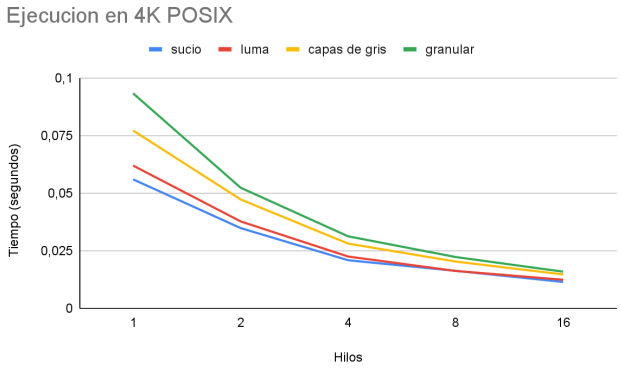
IV .RESULTADOS

POSIX

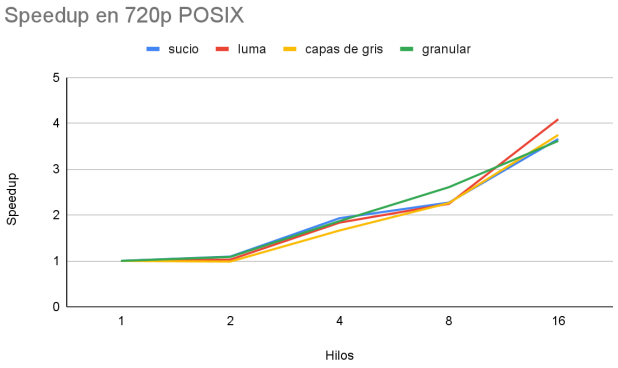
Tiempo de ejecución:

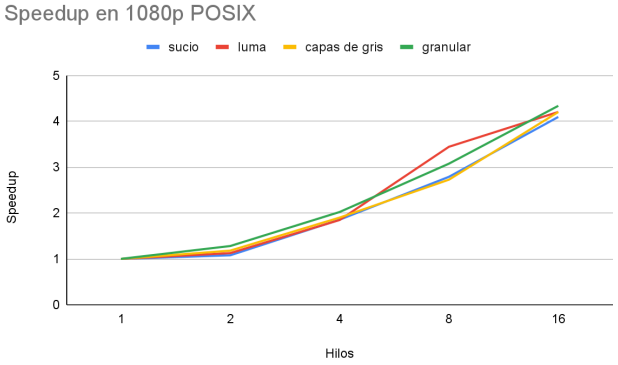


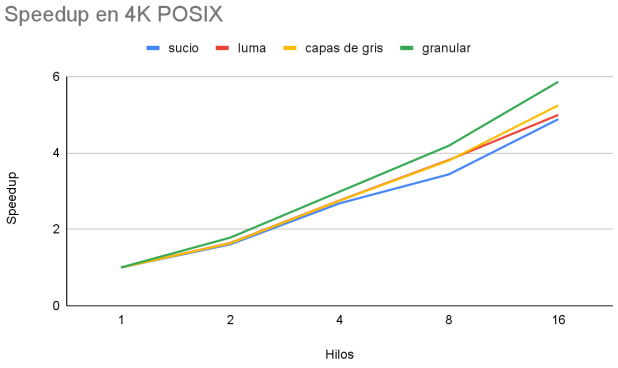




Speed Up:

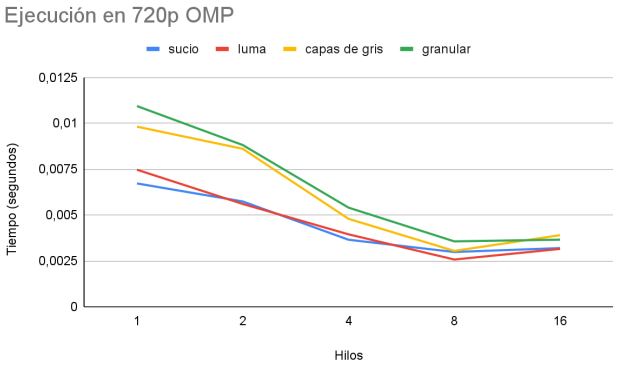




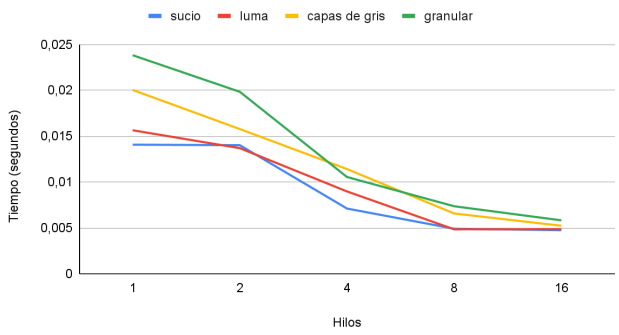


OMP

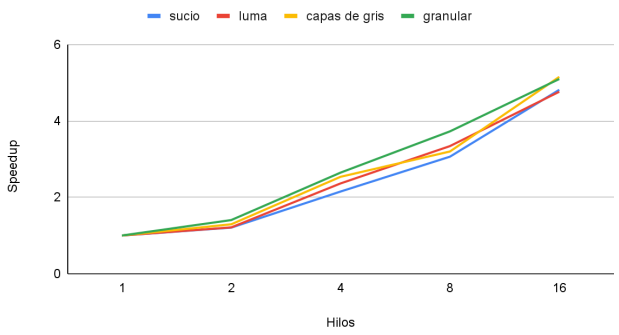
Tiempo de ejecución:



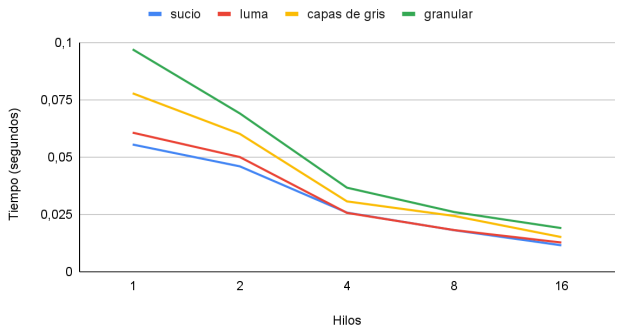
Ejecucion en 1080p OMP



Speedup en 4K OMP

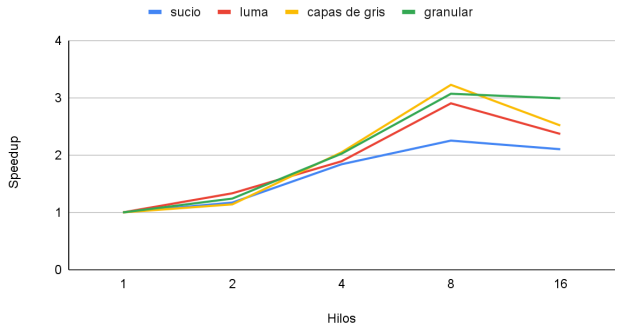


Ejecucion en 4K OMP

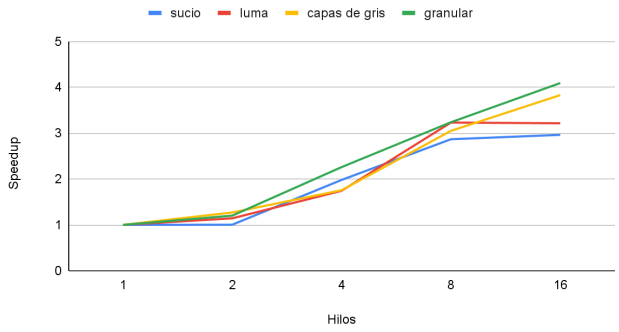


Speed Up:

Speedup en 720p OMP



Speedup en 1080p OMP

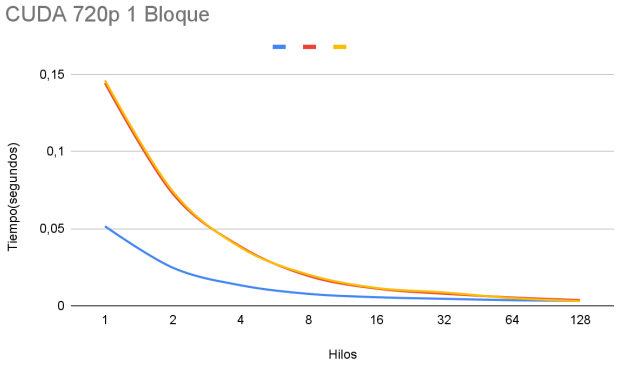


CUDA

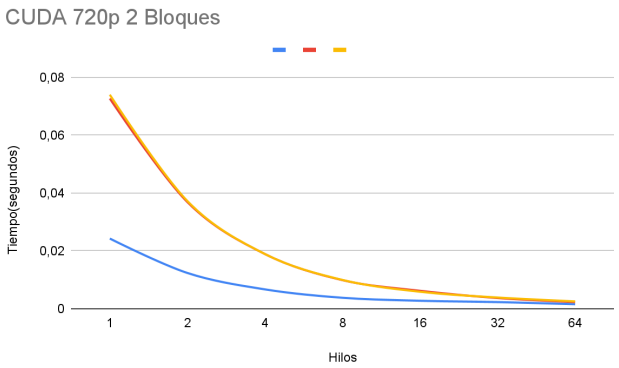
Rojo - Luma
Amarillo - Sucio
Azul - Capas de Gris

Tiempo de ejecución:

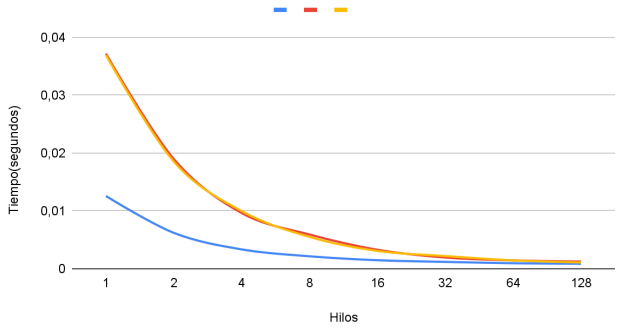
CUDA 720p 1 Bloque



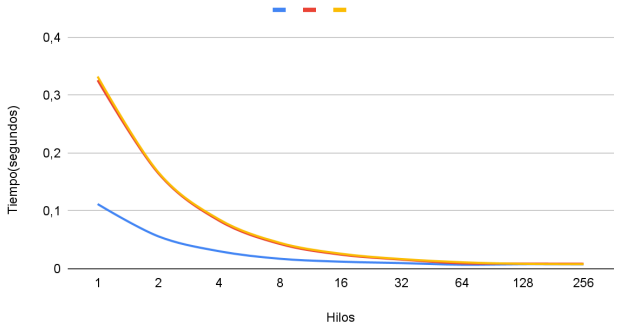
CUDA 720p 2 Bloques



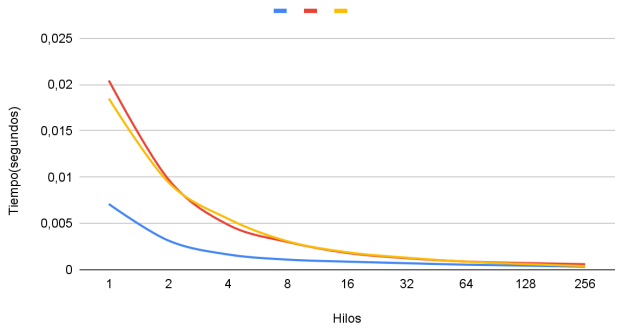
CUDA 720p 4 Bloques



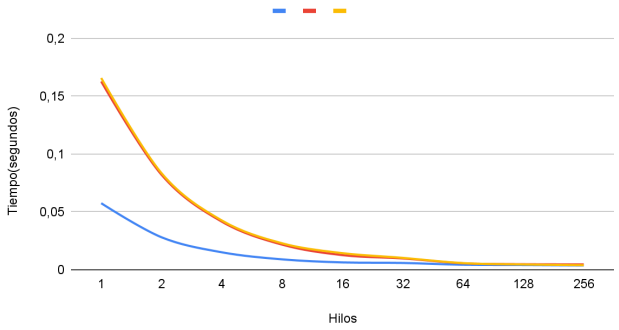
CUDA 1080p 1 Bloques



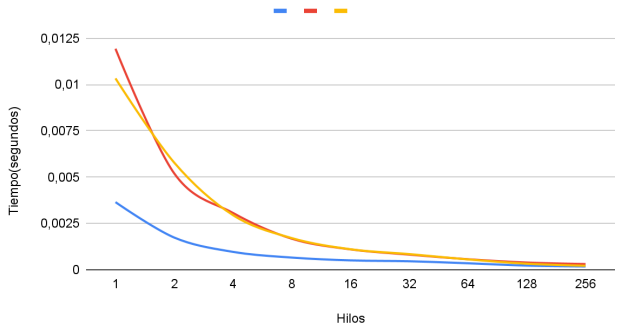
CUDA 720p 8 Bloques



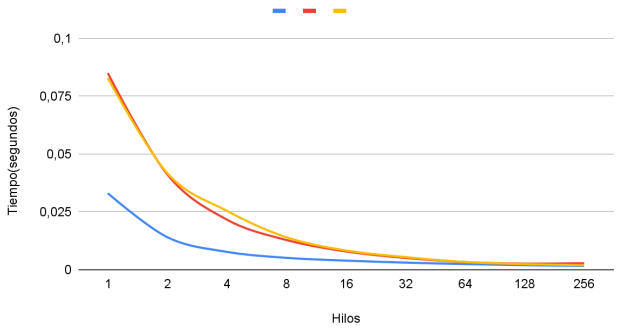
CUDA 1080p 2 Bloques



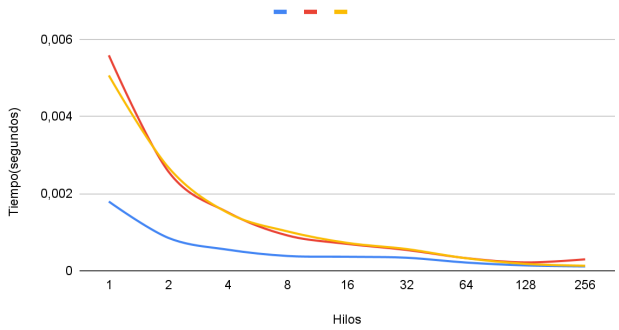
CUDA 720p 16 Bloques



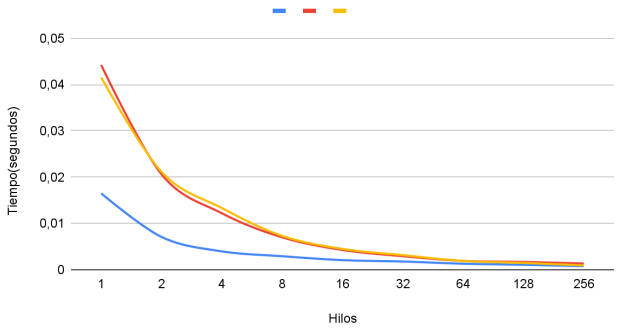
CUDA 1080p 4 Bloques

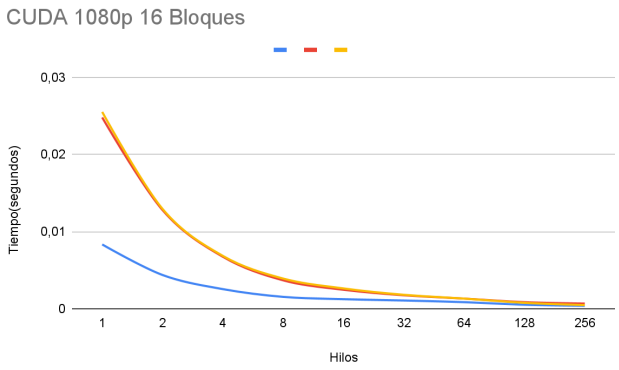


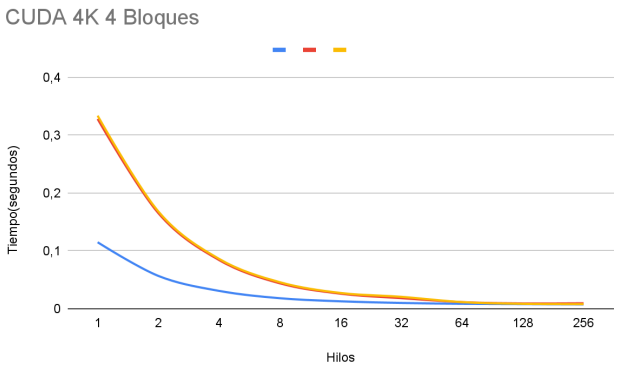
CUDA 720p 32 Bloques

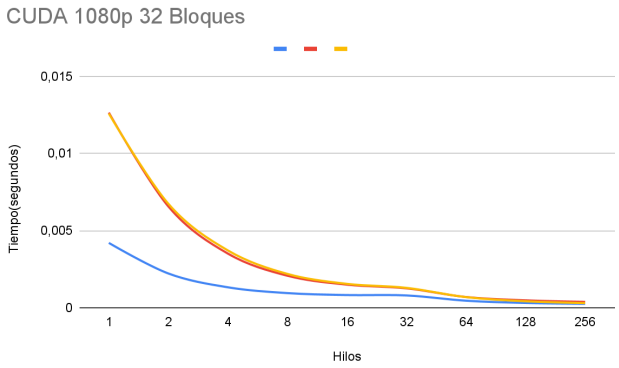


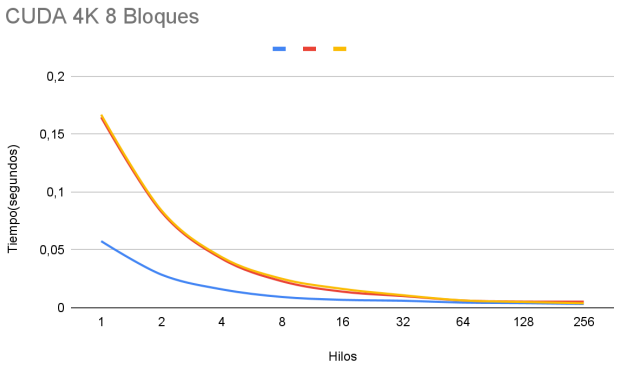
CUDA 1080p 8 Bloques

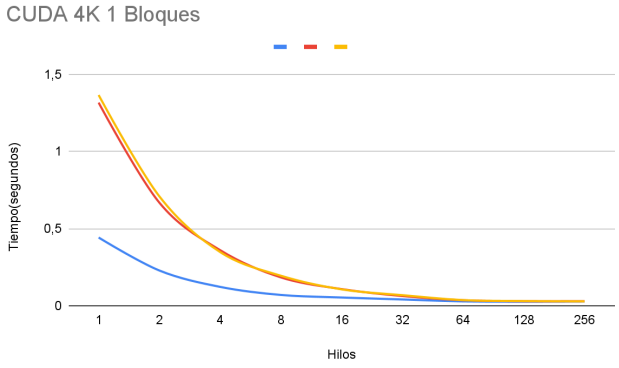


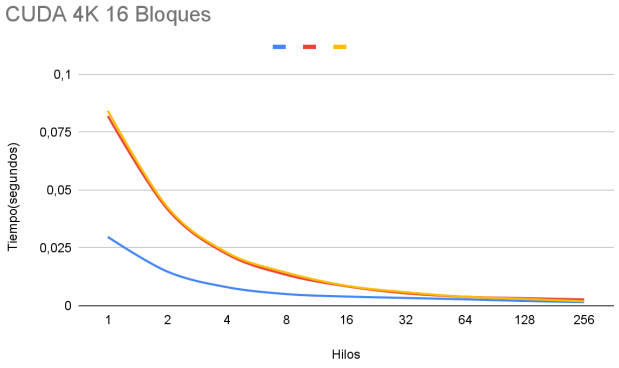


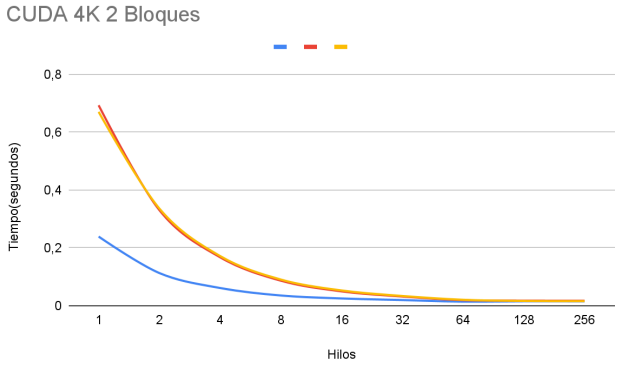


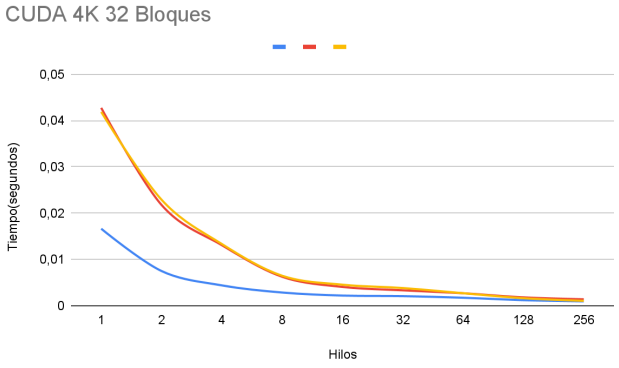






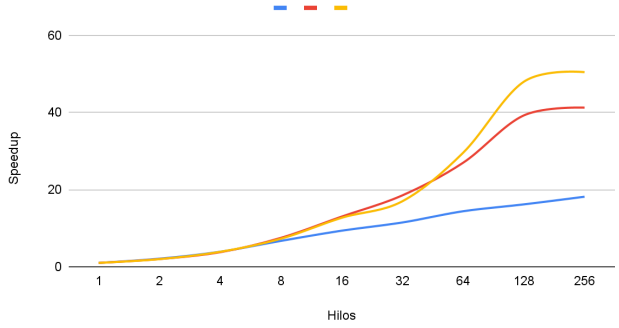




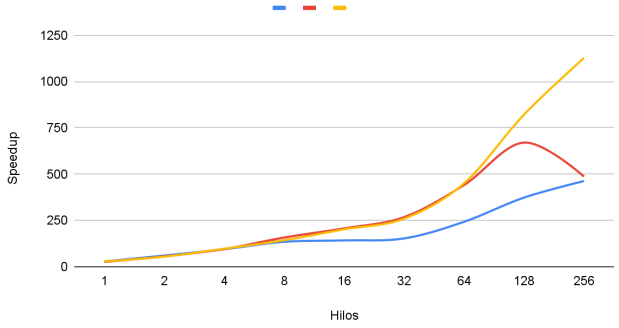


Speed Up CUDA:

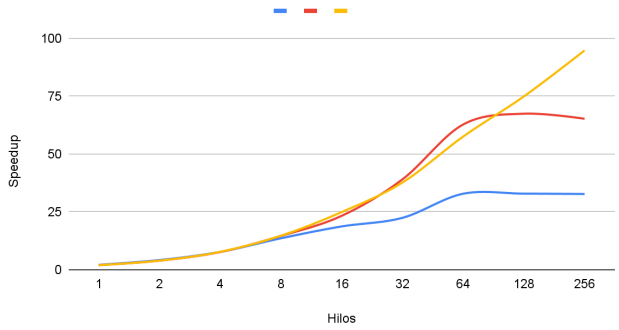
CUDA SpeedUp 720p 1 Bloque



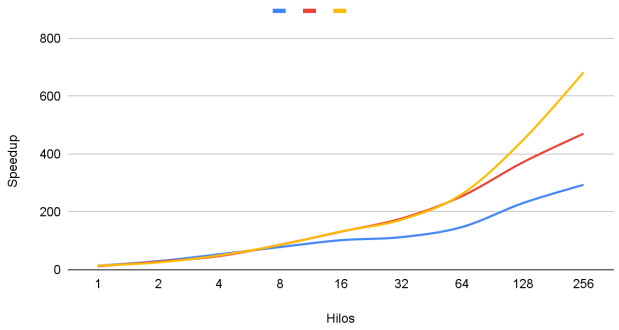
CUDA SpeedUp 720p 32 Bloques



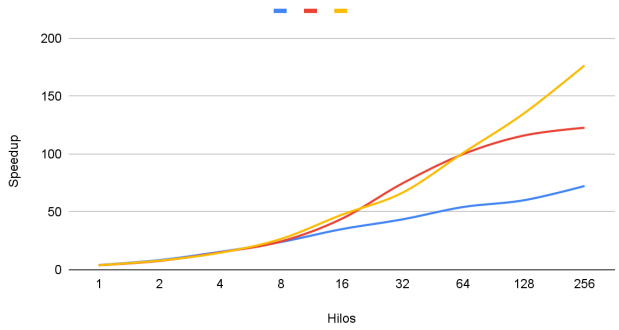
CUDA SpeedUp 720p 2 Bloques



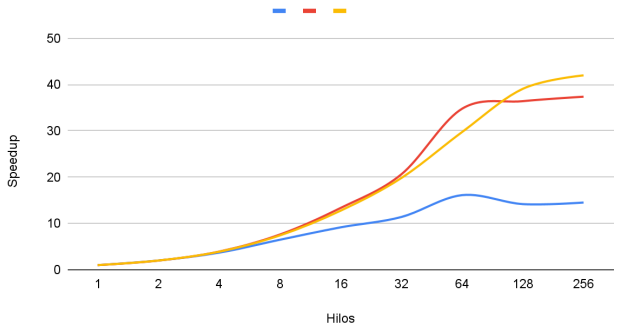
CUDA SpeedUp 720p 16 Bloques



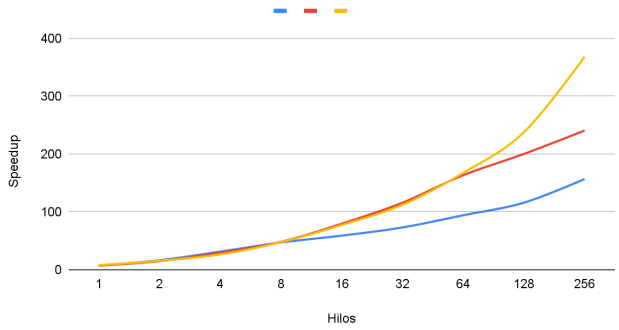
CUDA SpeedUp 720p 4 Bloques



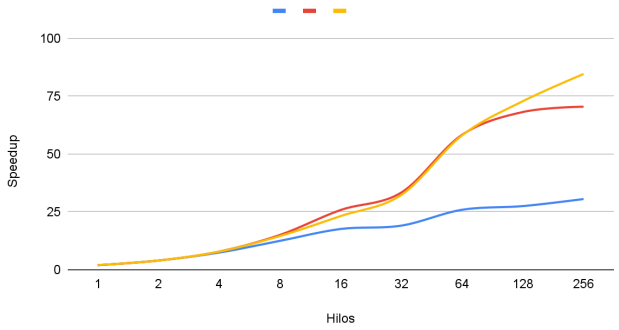
CUDA SpeedUp 1080p 1 Bloque



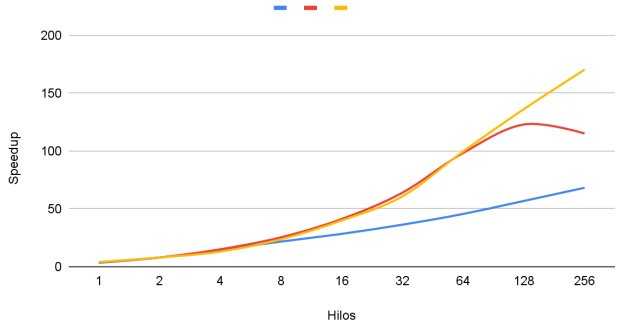
CUDA SpeedUp 720p 8 Bloques



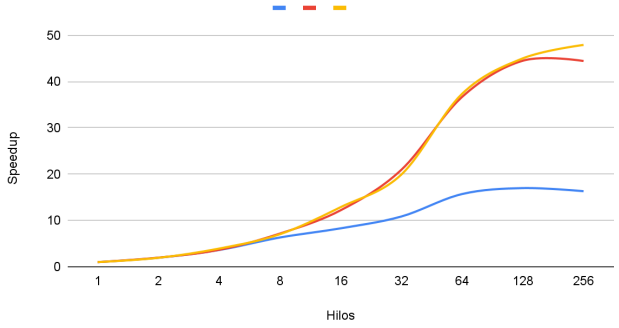
CUDA SpeedUp 1080p 2 Bloques



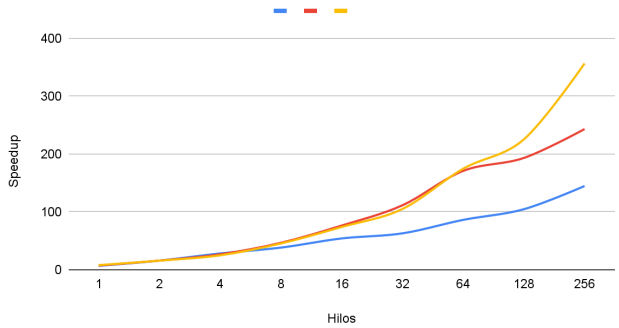
CUDA SpeedUp 1080p 4 Bloques



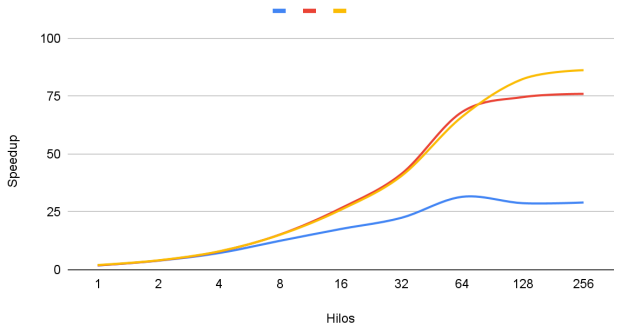
CUDA SpeedUp 4K 1 Bloque



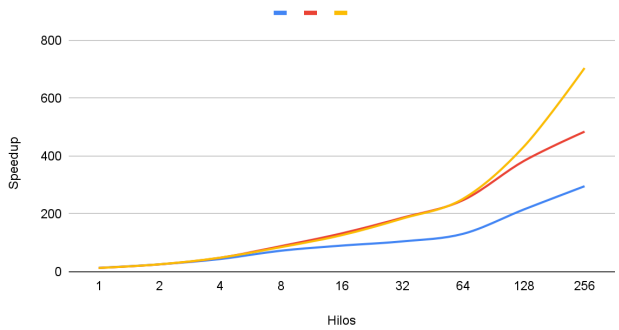
CUDA SpeedUp 1080p 8 Bloques



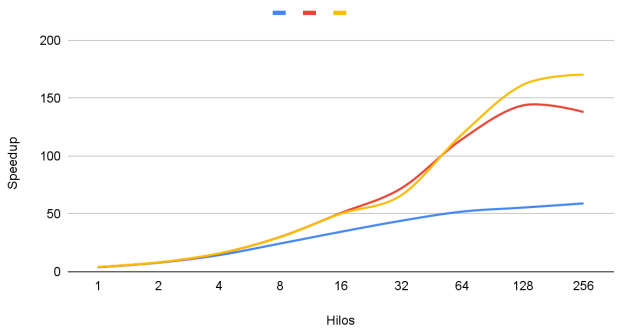
CUDA SpeedUp 4K 2 Bloques



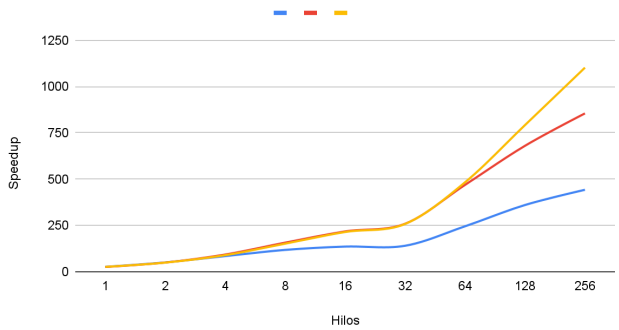
CUDA SpeedUp 1080p 16 Bloques



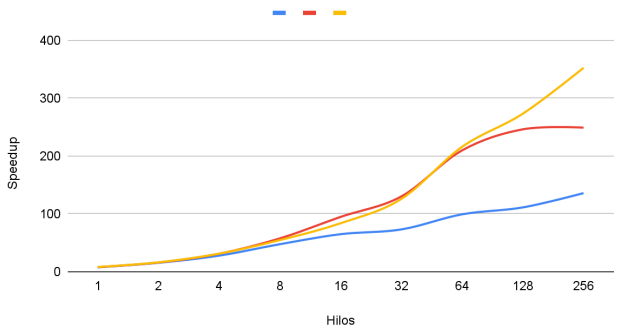
CUDA SpeedUp 4K 4 Bloques

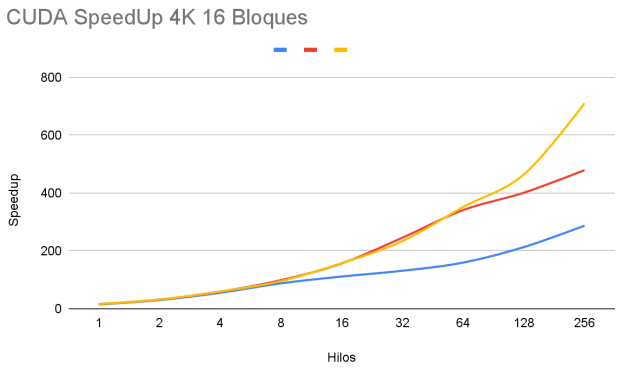


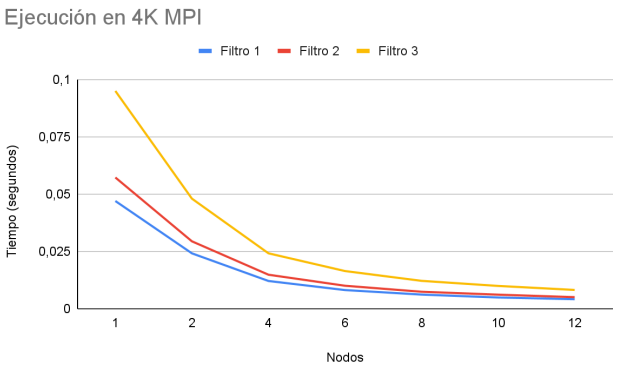
CUDA SpeedUp 1080p 32 Bloques

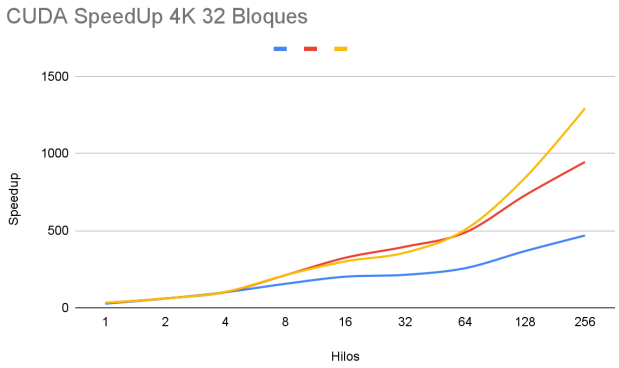


CUDA SpeedUp 4K 8 Bloques

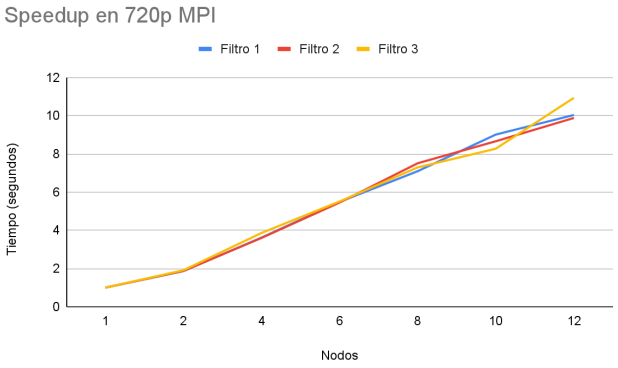






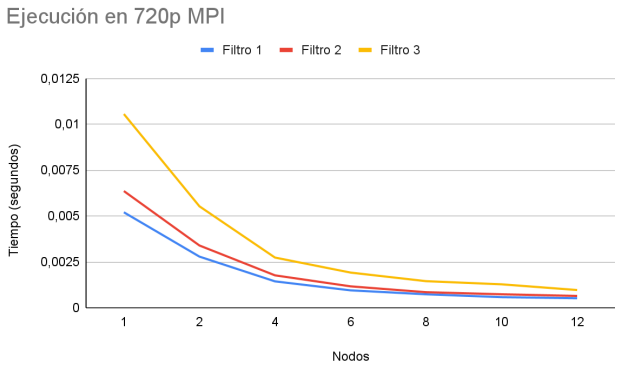


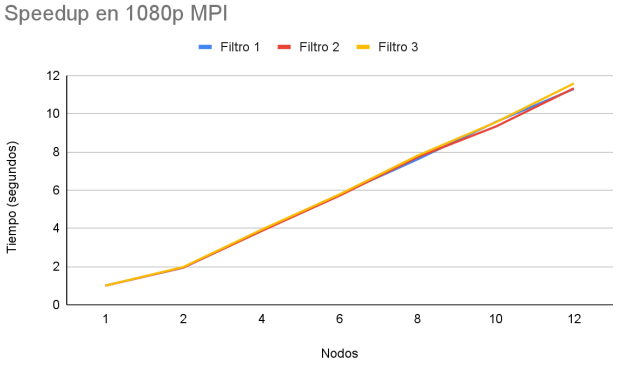
Speed Up:

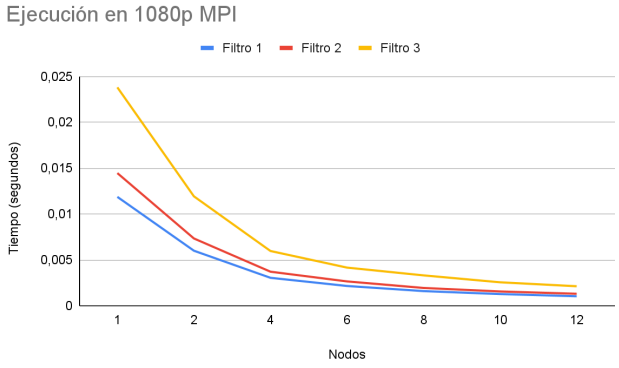


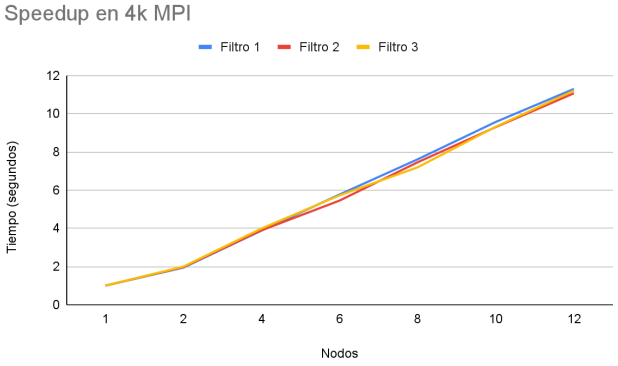
MPI

Tiempo de ejecución:









V.CONCLUSIONES

A. Generales, POSIX y OMP

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.

B. CUDA

En cuanto a CUDA, las gráficas se muestran por cada tipo de imagen y con diferentes potencias de 2 de bloques de invocación, los ejes x de cada gráfica hacen referencia a la cantidad de hilos por bloque con los que fueron ejecutados los programas.

Los resultados expuestos nos enseñan que claramente la GPU logra acelerar la ejecución de los programas significativamente más que los esquemas de paralelización por CPU. La utilización de estas unidades de cómputo diseñadas para paralelización masiva realmente salen a lucirse llegando a dividir la ejecución de un filtro en 32 bloques cada uno con 256 hilos, es decir, total de 8192 divisiones de la imágen. Particularmente para este paradigma, una alta cantidad de hilos y bloques siendo utilizados es importante, se puede notar la debilidad de este paradigma cuando se intentó el procesamiento de una imágen 4K con un solo bloque y un solo hilo por bloque.

La implementación por CUDA tiene desventajas en términos de instalación, compilación, compatibilidad y requiere unas cuantas declaraciones extras de memoria dinámica a cargo del programador pero, más allá de detalles ingenieriles como estos, realmente se muestra como una opción muy poderosa de paralelización con alto escalamiento y potencial.

C. MPI

Utilizando OpenMPI sobre un cluster de 8 máquinas virtuales como paradigma para la computación de nuestros filtros, podemos evidenciar el potencial de la división distribuida del trabajo entre diferentes nodos de una red. Para la medición del tiempo de ejecución expuesto en las gráficas anteriores, se tomó en cuenta desde las funciones encargadas de los cómputos del filtro fueron lanzadas hasta que se sincronizan los nodos ya teniendo calculado su “porción” de la imágen; esto quiere decir que: los tiempos requeridos de preparación y repartición de información previa, ni el tiempo que toma la reconstrucción de la imágen nuevamente en el nodo maestro fueron tenidos en cuenta(esto en pro de hacer la comparación tan fiel como fuese posible al modo en que fueron medidos los anteriores métodos).

Es de resaltar que, entre los paradigmas aquí mostrados, los resultados expuestos por MPI demuestran con mucha más consistencia e integridad un crecimiento lineal en el speedup a medida que más nodos son introducidos. Esto probablemente gracias a la buena repartición sincronizada de trabajo entre nodos y poder computacional equivalente entre todos los nodos.

MPI muestra también ventajas a través de fácil uso de la tecnología de pasos de mensajes pero, también muestra algunas desventajas ingenieriles a la hora de la preparación o setup de un cluster capaz de ser compatible horizontalmente con MPI y otras tecnologías como OpenCV.

D. Finalmente

La computación paralela es un paradigma que llegó para quedarse; la aceleración de procesamiento es un recurso invaluable y a veces necesario para el despliegue factible o realista de un servicio actual. En esta práctica se experimentó con diferentes herramientas para paralelizar el mismo programa, esto otorga perspectiva sobre cada herramienta de paralelización y como estas se comparan con las otras no sólo desde un aspecto netamente relacionado con la mejora temporal cuantificable pero teniendo en cuenta también los recursos necesarios para el correcto funcionamiento de cada herramienta así como la diferencia de proceso de programación de las mismas.

No se puede otorgar un veredicto transversal en cuanto a la mejor tecnología o paradigma con el cuál se pueden paralelizar los algoritmos en general.

Después de haber experimentado con estas 4 opciones, se puede analizar como una opción es más adecuada para este trabajo que las otras.... esta idea si es generalizable a los problemas paralelizables en general. Dependiendo de el tiempo que se quiera dedicar a la paralelización, los recursos con los que se cuenten, los problemas de compatibilidad que puedan salir, la porción paralelizable del problema y un análisis de qué tanto valor se podría extraer de la paralelización del programa, se puede decidir por la herramienta adecuada para el trabajo.

Todo el código fuente, imágenes utilizadas y manipuladas se encuentran en el siguiente repositorio:
https://github.com/VirtualDiegox/ComputacionParalelaUNAL-202101

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