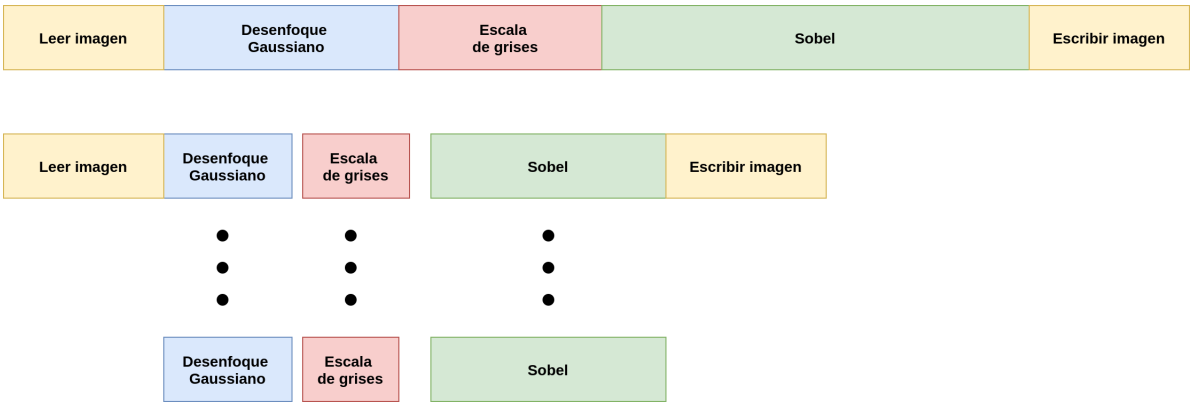
Laboratorio y Trabajo Teórico: Procesamiento de Imágenes

Computadores Avanzados

Juan Romero Cañas. Curso 2020-2021

El proyecto de prácticasy su correspondiente trabajo teórico consiste en la implementación de una aplicación para el procesamiento de imágenes digitales sobre GPU. En este caso, se han implementado tres filtros: **escala de grises**, **desenfocado Guassiano** y **filtro de Sobel** (detección de bordes). Los dos primeros filtros son pasos intermedios para Sobel. Además se ha implementado una versión en CPU para comparar con la versión de GPU.



En esta versión de CPU, los filtro de grises y Gauss se realizan mediante funciones de OpenCV, mientras que Sobel en si se realiza de forma totalmente secuencial. Por su parte, como se puede observar en la imagen superior, en la implementación de GPU los tres filtros están implementados de forma paralela.

Implementación en GPU

Para esta implementación se ha usado la API de CUDA para usar todos los recursos disponibles de la GPU. Todos los pasos para aplicar el filtro de Sobel a una imagen, incluido el prepocesado se realiza en el *device*, mediante sucesivas llamadas a diferentes *kernels*. El primer paso es pasar la imagen a escala de grises. Después, se aplica un filtro Gaussiano o de desenfoque, para reducir el posible ruido de la imagen y finalmente el propio filtro de Sobel o de detección de bordes. La lectura de la imagen, el uso de la webcam como método de entrada o el mecanismo para mostrar la imagen se realiza mendiante la librería de OpenCV. El procesado de una imagen sigue la siguiente secuencia:

- Se lee la imagen y se carga en memoria de *host*.
- Se reserva memoria en *device* para la imagen origen y destino, mediante la función `cudaMalloc()`.
- Se copia la imagen al *device* mediante la función `cudaMemcpy()`.
- Se llama al kernel y se procesa la imagen.
- Se copia el resultado de vuelta al `host`.
- Se liberan los recursos ocupados en el *device*

Excepto el primer paso, los demás se ejecutan tres veces, una por cada filtro que se aplica a la imagen. Una vez ha procesado la imagen por completo, se muestra en pantalla. En caso de ser procesar un vídeo, estos pasos se repiten por cada frame del mismo.

Para procesar la imagen se usa un *grid* bidimensional compuesto por bloques también bidimensionales. Los bloques esta compuesto de 16x16 hilos, mientras que el numero de bloque variará según el tamaño de la imagen. Por ejemplo, para una imagen de 640x480 píxeles de usará un un *grid* de 40x30 bloques. El tamaño de bloque es fijo ya que es el que maximiza el rendimiento, asi como el uso de GPU (se puede observa el campo *occupancy* usando el `nvvp`). Para almacenar la imagen se usa una array unidimensional, de forma que se facilita el acceso a los datos.

Entorno de pruebas

Las pruebas se han llevado a acabo en un sistema con una CPU i7-7500U, 12 GB de memoria RAM y una tarjeta gráfica GeForce 920MX. Esta se caracteriza por tener 256 núcleos CUDA y 2 GB de memoria. Como lenguaje se ha usado C++ y como sistema operativo Ubuntu 20.04.1 LTS.

Respecto a software externo usado hay que destacar que para todo lo relacionado con la lectura de imágenes se ha usado OpenCV. Adicionalmente, para el tratamiento de argumentos en el propio programa se ha usado la librería `cxxopts`[1].

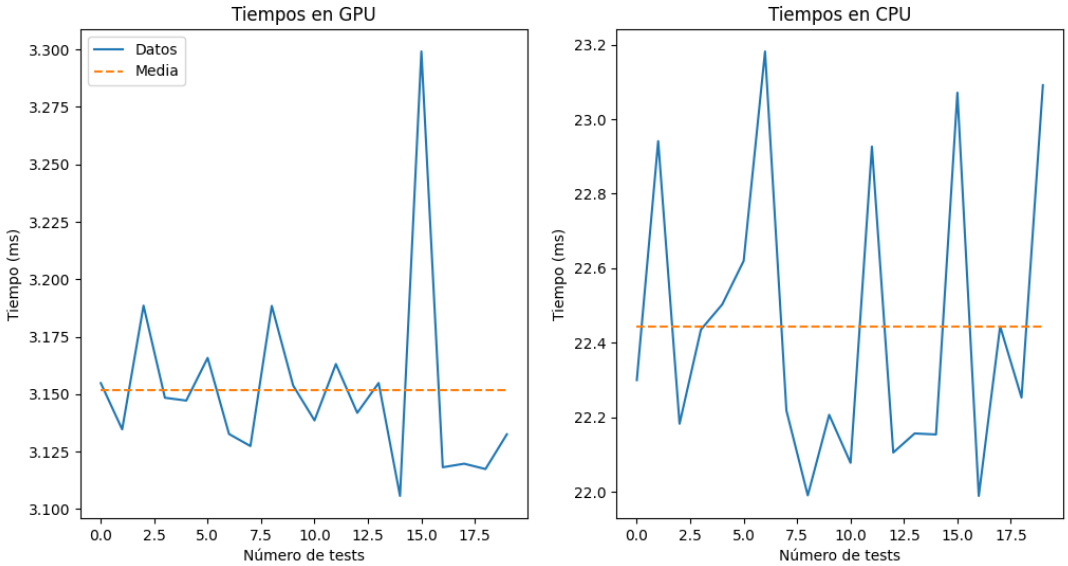


Figura 1: Imagen SD: tiempos CPU vs GPU

Rendimiento

Para comparar los resultados obtenido se usará el tiempo de procesamiento en caso de las imágenes y los FPS obtenidos, en caso de los vídeos. La imágenes usadas tienen las siguientes resoluciones:

- Calidad SD: 640x480.
- Calidad HD: 1280x720.
- Calidad FHD: 1920x1080.
- Calidad 4K: 3840x2160.
- Calidad 8K: 7680x4320.

Para realizar las comparaciones y automatizar la obtención de resultados se adjuntan varios scripts de Python3 que ejecutan el programa varias veces y representa gráficamente los resultados. Podemos ver en la Figura 2 la diferencia de tiempos de ejecución según la resolución de la imagen.

Se puede observa en la Figura 2 como la implementación en CPU escala bastante mal, mientras la implementación en GPU escala mucho mejor. En caso de la imagen con menor resolución, 640x480, obtenemos una ganancia de 6.8x. Para el extremo opuesto, la imagen 8K tenemos una ganancia de 9.3x. Para el resto de resoluciones la ganancia es cercana a 6x en todos los casos.

Si nos centramos en la ejecución usando la imagen 8K, obtenemos un tiempo de `252.879 ms`. Sin embargo la mayoría de este tiempo se dedica a tareas de I/O, es decir mover los datos de *host* al *device* y viceversa. Por ejemplo, para el caso anterior el tiempo acumulado de los tres kernels es `73.66 ms`, lo que supone que las tareas de I/O supone un 70 % del tiempo total.

[1] https://github.com/jarro2783/cxxopts

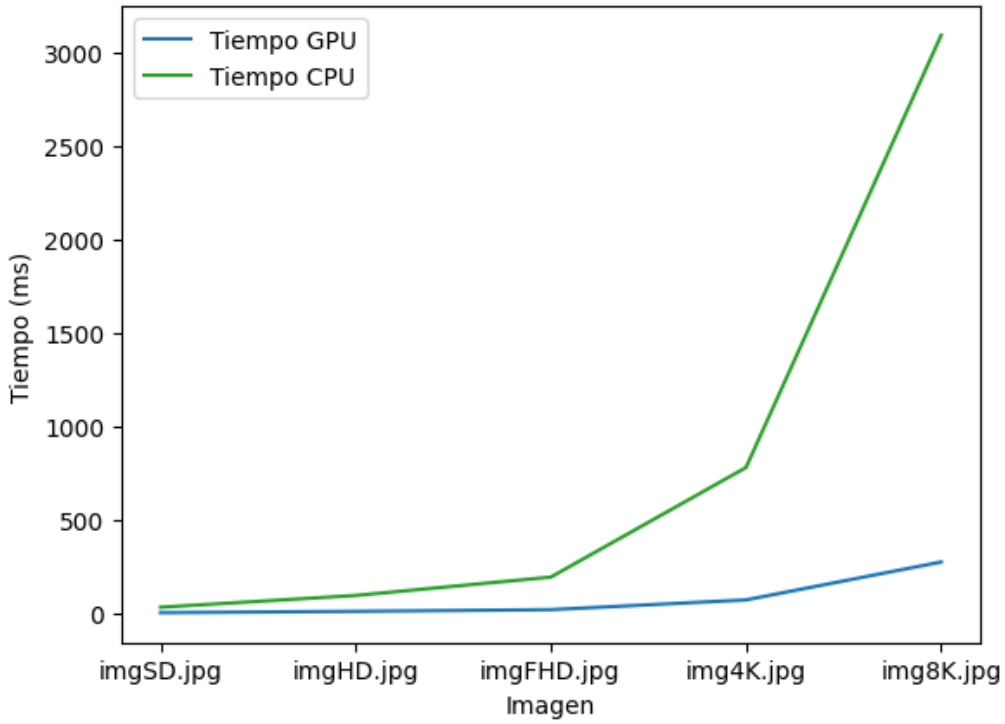


Figura 2: Comparación de tiempos CPU vs GPU

Respecto al procesamiento de vídeos los resultados son similares al los obtenidos con las imágenes, ya que los FPS obtenidos están directamente relacionado con el tiempo de procesamiento de un solo frame o imagen.

Conclusión

Los resultados obtenidos demuestran como la implementación en GPU es muy superior al obtenido con la CPU, tanto en tiempo de ejecución, como en escalabilidad al aumentar el tamaño de la imagen a procesar. Se puede observa con este ejemplo el potencial de las GPU para procesar muchos datos de forma paralela.

Respecto a las **líneas de mejora** podemos destacar el uso de memoria compartida para intentar reducir el tiempo de ejecución de los kernels. Sin embargo, puesto que el tiempo de estos solo supone un 30 % del tiempo total, quizá sería más interesante intentar reducir los tiempos de I/O. Una posible solución a este problema podría ser llamar a los kernel de forma sucesiva. Es decir, según la implementación actual para la ejecución de cada kernel se reserva memoria en el *device*, se copia del *host* al *device* y viceversa y ser liberan recursos. Este proceso se repite tres veces. Se podría combinar en un único proceso, de forma que solo se necesitaría mover datos dos vedes del *host* al *device* y otra del *device* al *host*.

En conclusión, este trabajo me ha servido no solo para aprender CUDA, sino también para entender el funcionamiento de los sistemas SIMD y como obtener todo su potencial.

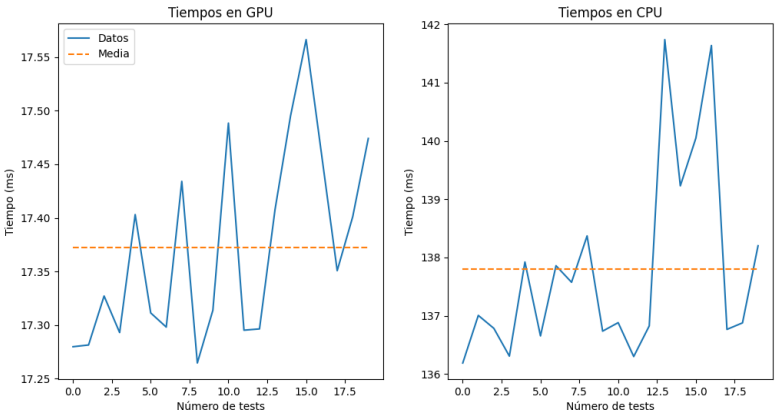


Figura 3: Imagen FHD

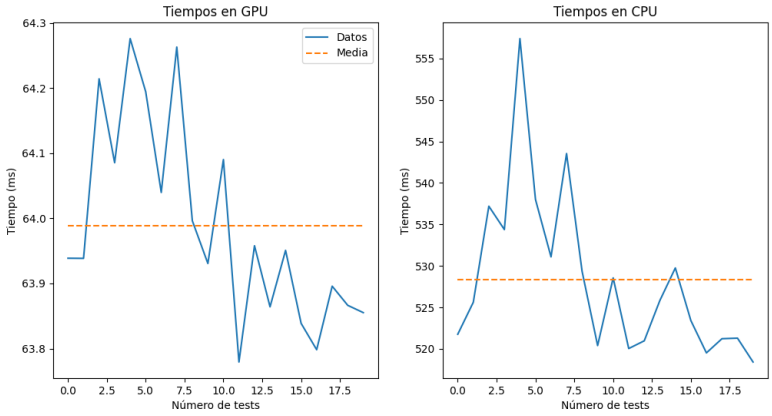


Figura 4: Imagen 4K

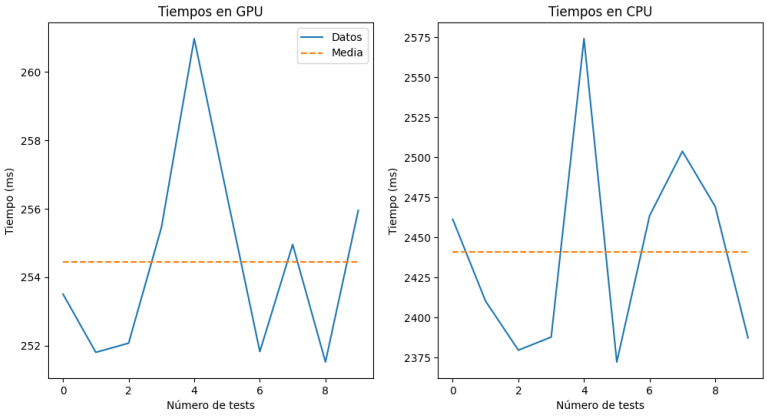


Figura 5: Imagen 8K