Procesadores Gráficos y Aplicaciones en Tiempo Real

Filtros de convolución en CUDA

Álvaro Muñoz Fernández
Iván Velasco González

Índice

1. Introducción

El objetivo de esta práctica consiste en la implementación de un filtrado de imágenes mediante filtros de convolución. La implementación de dichos filtros se realizará en CUDA, aprovechando así las capacidades de computación en paralelo de las GPU.

2. Directivas de compilador

Para facilitar el desarrollo y la realización de pruebas se ha optado por utilizar directivas de preprocesador. Estas están definidas al inicio del fichero con comentarios que definen su comportamiento. Mediante estas directivas puede cambiarse el filtro a emplear (y el tamaño del filtro en el caso de elegir un filtro gaussiano), activar o desactivar el uso de memoria compartida y alterar el tamaño de bloque a utilizar.

3. Estructura del código

Todo el código escrito para la práctica se encuentra en el fichero *func.cu* proporcionado como material de apoyo. En este fichero hay definidas varias funciones que han sido completadas para cumplir los requisitos de la entrega y que se exlican a continuación:

3.1. convolution()

Esta función es llamada automáticamente por el framework proporcionado para la realización de la práctica. En esta función se han realizado los cálculos de tamaño y número de bloques a utilizar, y se realizan las llamadas a los diferentes kernels de CUDA con los parámetros adecuados. Primero se llama al kernel de separación de canales de la imagen y posteriormente al kernel de filtrado con cada uno de los canales independientemente.

3.2. cleanup()

En esta función se realiza la liberación de memoria reservada durante la ejecución.

3.3. create_filter()

Según el filtro seleccionado, esta función reservará la memoria necesaria para el filtro de convolución e inicializará el filtro con los valores apropiados. Se han implementado los siguientes filtros:

1. **Laplacian5x5:** Filtro laplaciano de tamaño 5x5.
2. **Nitidez5x5:** Filtro de nitidez de tamaño 5x5.
3. **PasoAlto5x5:** Filtro de suavizado de tamaño 5x5.
4. **Media3x3:** Filtro de suavizado que aplica la media de todos los pixels circundantes en un área de 3x3 pixels.
5. **Blur3x3:** Filtro de suavizado que da mayor peso a los pixels próximos en un área de 3x3.
6. **Blur5x5:** Filtro de suavizado que da mayor peso a los pixels próximos en un área de 5x5.

7. **GaussianBlur:** Filtro de suavizado que aplica un desenfoque gaussiano, con un tamaño definido por la constante *GAUSSIAN_SZ*.

8. **SobelHori3x3:** Filtro de detección de bordes horizontal, tamaño 3x3.

9. **SobelVert3x3:** Filtro de detección de bordes vertical, tamaño 3x3.

3.4. allocateMemoryAndCopyToGPU()

Aquí se realizará la reserva de memoria en el dispositivo, subiéndose también los datos necesarios para la posterior ejecución de los kernels.

3.5. separateChannels()

Este kernel se encarga de recibir un buffer de elementos *char4* con la información de color de la imagen, separando cada uno de estos elementos en sus canales RGB y almacenándolos de forma independiente en los buffers previamente reservados para la información de cada canal.

3.6. box_filter()

En este kernel se aplican los filtros de convolución recorriendo, para cada thread, una región del tamaño del filtro de convolución centrada en el pixel correspondiente al thread actual, recogiendo los valores de color de dicha región de la imagen y multiplicándolos por los valores definidos en el filtro. Los resultados de estas multiplicaciones se sumarán en una variable auxiliar y una vez terminada la convolución se ajustará el valor final al rango [0-255], almacenándose este resultado en el buffer de color de salida.

3.7. box_filter_shared()

Se trata de una copia de la función anterior que ha sido modificada para hacer uso de memoria compartida en la que, durante una etapa inicial en la ejecución del kernel, los threads copiarán la región de la imagen a tratar (incluyendo los márgenes necesarios para poder aplicarse un filtro de tamaño variable) de forma que sólo se realice un número mínimo de accesos a memoria global. Durante la aplicación de las convoluciones los threads, a la hora de leer los valores de la imagen, harán las lecturas directamente desde el buffer almacenado en memoria compartida.

4. Mejoras implementadas

4.1. Selección de filtros y configuración

Como se menconaba al inicio del documento, para facilitar las pruebas se han incluído directivas de preprocesador que permiten seleccionar el filtro a aplicar, así como activar o desactivar el uso de memoria compartida o alterar el tamaño de bloque a utilizar. Además de los filtros propuestos se han implementado algunos filtros extra enumerados en el apartado anterior.

4.2. Adecuación de bloques

Se han realizado pruebas modificando el tamaño de bloque, pero las diferentes dimensiones no han alterado los tiempos de ejecución de manera significativa. Aunque ha sido posible ver una muy ligera mejora de rendimiento al ajustar las dimensiones de bloque para lanzar un número de threads igual al número de pixels de la imagen, la diferencia

en el tiempo de ejecución es mínima. Esto puede deberse a que, aunque en el peor de los casos haya bloques con un alto número de threads desocupados, salvo que se trate de una imagen de unas dimensiones muy peculiares (por ejemplo de Nx1 pixels, causando una desocupación casi completa en los threads de cada bloque) las dimensiones de bloque no causan una diferencia de rendimiento perceptible bajo las condiciones de la práctica. También hay que notar que un tamaño de bloque de 16x16 mostraba mejores resultados que bloques de 8x8 o 32x32, aunque tampoco han sido diferencias notables.

Se ha podido observar que el límite de tamaño de bloque para la GPU empleada para las pruebas (Geforce GTX1080) es de 32x32, no habiendo podido ejecutarse correctamente el programa cuando las dimensiones de bloque eran mayores.

4.3. Memoria de constantes

Dado que el filtro únicamente será leído durante la ejecución de los kernels, que tiene un tamaño reducido y que todos los threads de un mismo warp accederán a las mismas posiciones del filtro en paralelo, se ha decidido almacenar el filtro en memoria de constantes. El filtro es subido antes de iniciar la ejecución, siendo accedido después por los diferentes bloques. Al hacerse un recorrido en bucle en cada thread, estando sincronizados todos los threads de un mismo warp y siendo todos los accesos al buffer del filtro por parte de un mismo warp serán al mismo elemento, puede aprovecharse la cualidad de la memoria de constantes para servir a todos los threads en un broadcast.

Al realizar pruebas de rendimiento no se ha observado la mejora prevista, pero esto puede deberse a que al tratarse de filtros de muy pequeño tamaño, la GPU puede mantener el filtro en caché sin que se generen fallos de caché, haciendo irrelevante el uso de memoria de constantes ya que los accesos a memoria global seguirán siendo los mínimos necesarios.

4.4. Memoria compartida

Para la implementación de memoria compartida se ha creado una nueva función derivada de *box_filter()* que primero realiza una etapa de pre-procesado en el que cada thread del bloque contribuye a copiar la región de la imagen necesaria para la ejecución en un bloque de memoria compartida. Una vez terminada la copia a memoria compartida, los threads se sincronizan y realizan el proceso de aplicación del filtro leyendo los valores del filtro de la memoria de constantes y los valores de la imagen de la memoria compartida.

En este caso sí se han observado diferencias de rendimiento, aunque estas sólo han sido notables cuando se ha aumentado el tamaño del filtro de convolución. Mientra que en filtros de pequeño tamaño los tiempos de ejecución han sido similares o incluso ligeramente superiores a los de la implementación sin memoria compartida (probablemente por el overhead causado por las instrucciones de la lógica de copia y por la sincronización forzada de threads), cuando se ha aumentado el tamaño de filtro a 9x9 aplicando un filtro gaussiano a la imagen *sunrise_sunset.jpg*, se han observado tiempos de ejecución de unos 550ms al emplear memoria compartida, en contraste con los 590ms al no utilizar memoria compartida.

La causa más probable es que, al no estar optimizados los accesos a memoria, el aumento del número de operaciones realizadas y de la distancia en memoria de los distintos accesos está provocando fallos de caché, haciendo que los accesos a memoria global sean mucho más costosos que los accesos a memoria compartida.

4.5. Pruebas locales

Para asegurar el correcto funcionamiento de los filtros se han realizado ejecuciones sobre las imágenes proporcionadas y se han comparado con los resultados de ejemplo. Se han obtenido coincidencias exactas tanto al aplicar el filtro laplaciano 5x5 como al aplicar el gaussian blur 9x9.

Las directivas de preprocesador necesarias para estas ejecuciones son **FILTER = 1** para el filtro **Laplacian5x5**; y **FILTER = 7**, **GAUSSIAN_SZ = 9** para el filtro **Gaussian9x9**.