

Dispositivos e Infraestructuras para Sistemas Multimedia 2018

Introducción a la Programación de GPUs usando CUDA

Práctica 1: Suma de Vectores

Alberto García García
Sergio Orts Escolano
José García Rodríguez

12 de noviembre de 2018

Proudly made with L^AT_EX

Copyright (C) 2017-2018 Alberto García García, Sergio Orts Escolano, José García Rodríguez

Índice general

1. Práctica 2: Filtro Mediana	5
1.1. El Algoritmo del Filtro Mediana	5
1.2. Flujo de Ejecución	7
1.3. Ejercicios	8
1.3.1. Ejercicio 0 (0 puntos) Creación y Ejecución del Proyecto .	9
1.3.2. Ejercicio 1 (1.5 puntos): Optimización de Transferencias entre GPU y CPU	9
1.3.3. Ejercicio 2 (2.5 puntos): Acceso Coalescente a Memoria .	10
1.3.4. Ejercicio 3 (2.5 puntos): Optimización de Ocupación de la GPU	11
1.3.5. Ejercicio 4 (1.5 puntos): Investigando el Tamaño de Malla y de Bloque	12
1.3.6. Ejercicio 5 (2 puntos): Optimización con Memoria Com- partida	12

Capítulo 1

Práctica 2: Filtro Mediana

En esta segunda práctica se plantea la implementación de un algoritmo de mayor complejidad que la suma de vectores, así como su optimización para obtener un rendimiento superior. En concreto, planteamos la implementación en CUDA de un filtro ampliamente utilizado en el campo del procesamiento de imágenes: el filtro mediana. El código fuente de la práctica se encuentra en el archivo `filtro_mediana.cu`, el cual compila y funciona inicialmente con una implementación ineficiente que utilizaremos como punto de partida. Los objetivos de esta práctica, principalmente centrados en el tratamiento de la memoria y optimización, son los siguientes:

- Profundizar en el flujo de ejecución de programas CUDA.
- Comprender el impacto de las transferencias de memoria y utilización de técnicas para minimizarlo.
- Explorar el concepto de acceso coalescente a memoria.
- Explorar el concepto de ocupación de recursos en la GPU.
- Comprender los efectos de los tamaños de bloque y de grid.

1.1. El Algoritmo del Filtro Mediana

En el campo de procesamiento de imágenes, una de las operaciones más necesarias y por lo tanto utilizadas es el filtrado de ruido. Este preprocesamiento se suele llevar a cabo antes de aplicar otros algoritmos para obtener información de la imagen como pueden ser filtros detectores de bordes, detectores de puntos característicos o descriptores. Entre las diversas técnicas existentes, el filtrado no lineal de mediana es una de las más utilizadas.

La idea en la que se fundamenta este filtro consiste en recorrer píxel a píxel toda la imagen, reemplazando cada valor de intensidad del píxel correspondiente por la mediana de los valores de intensidad de los píxeles de su vecindad (incluyéndose a sí mismo). Esta vecindad también se conoce como ventana, y su tamaño puede variar (siendo el más común 3×3 obteniendo un total de nueve elementos) para obtener diferentes resultados de suavizado o eliminación de ruido. La Figura 1.1 ilustra el proceso del cómputo del filtro mediana para un píxel concreto de una imagen.

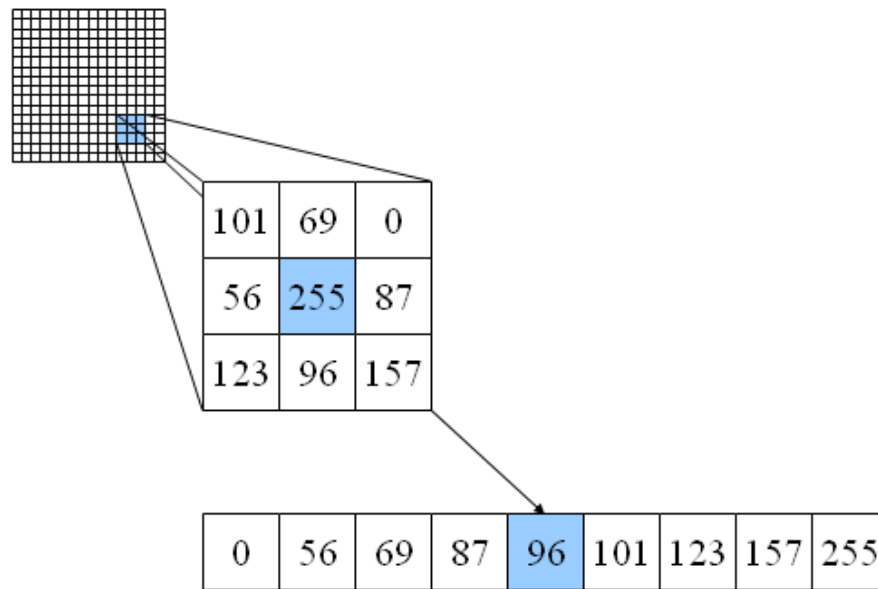


Figura 1.1: Ejemplo de procesamiento de píxel de una imagen mediante filtro mediana. Se toma una vecindad de 3×3 píxeles sobre el que está siendo procesado y se ordenan los valores de intensidad para reemplazar el valor del píxel central por la mediana de los valores de intensidad de la vecindad.

Generalmente, este filtro se aplica de forma iterativa ya que una única ejecución puede no ser suficiente para eliminar el ruido presente en la imagen. La contrapartida de esta operación es la distorsión producida en la imagen original en forma de emborronamiento, por lo que un número excesivo de iteraciones puede conllevar este efecto indeseado en la imagen en lugar de reducir el ruido. En la Figura 1.2 se muestra el efecto del filtro sobre una imagen con ruido característico del tipo sal y pimienta.



Figura 1.2: Ejemplo de resultado de aplicación de un filtro mediana 3x3 tras 10 iteraciones (derecha) sobre la popular imagen de Lena con ruido de tipo sal y pimienta (izquierda).

1.2. Flujo de Ejecución

Para llevar a cabo la práctica se proporciona una implementación CUDA funcional del filtro mediana sin optimizar. El código principal de la práctica se encuentra en el archivo `median.cu`. Además, se proporcionan los archivos de la librería EasyBMP para la manipulación de mapas de bits: las cabeceras `EasyBMP.h`, `EasyBMP_BMP.h`, `EasyBMP_DataStructures.h` y `EasyBMP_VariousBMPutilities.h` junto con las implementaciones de las funciones en `EasyBMP.cpp`.

Dentro del archivo `median.cu` se encuentran tres kernels distintos, siendo el primero de ellos la implementación funcional que se proporciona: `medianFilter1D_col`. Los otros dos kernels `medianFilter1D_row` y `medianFilter2D` tendrán que completarse a lo largo de la práctica. El código proporcionado también utiliza funciones para medir el tiempo de ejecución tanto en la GPU como en la CPU. Además, la versión de filtro mediana implementada para la CPU permite testear la salida de la solución CUDA para comprobar que es correcta y emitir mensajes de error en caso de que no lo sea.

En el código proporcionado, la imagen se almacena como un array bidimensional en la parte host. Para el cálculo del filtro utilizaremos una ventana o vecindad de 3x3. Dada esta ventana, es necesario añadir una región adicional sobre los bordes de la imagen para facilitar el cálculo y evitar la comprobación de los límites de tamaño durante el procesamiento (imaginad cuál es la vecindad de un píxel esquina o borde en la imagen, ciertos píxeles quedan fuera por lo que para evitar esta situación ampliamos la imagen con un halo de valores

0). Para la parte device o GPU, la imagen será tratada como un vector unidimensional (fila a fila de la imagen), por lo que cambiará la forma de direccionar el acceso a los píxeles y mapear los hilos. En la Figura 3 se muestra una representación visual del halo introducido en una imagen de 256x256 píxeles y las posiciones correspondientes linearizadas.

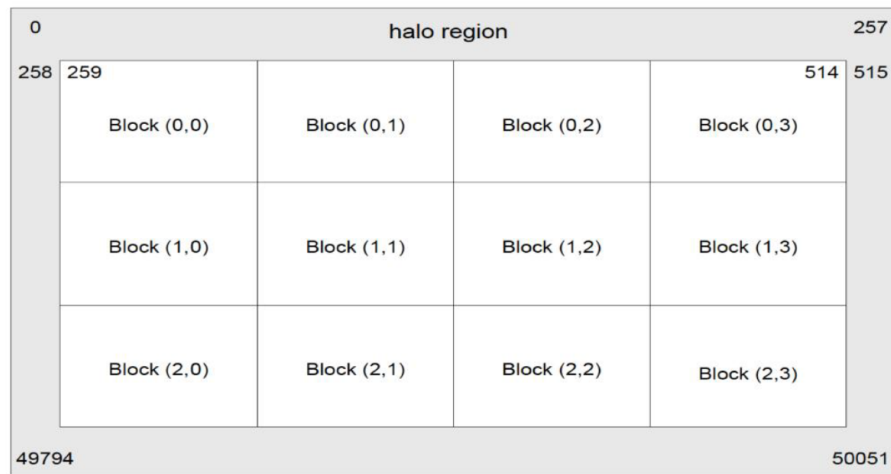


Figura 1.3: Halo introducido para una imagen de 256x256 píxeles para utilizar una ventana de 3x3 (como se puede observar, se introduce una nueva fila y columna de ceros en los bordes de la imagen).

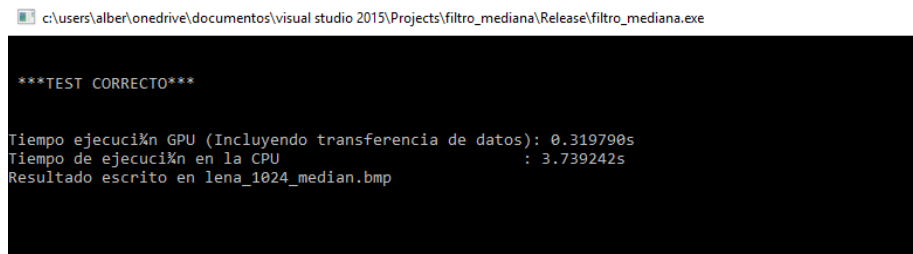
El kernel proporcionado asigna cada fila de la imagen a un hilo en CUDA. Cada hilo itera sobre las columnas de la imagen, calculando los índices de la vecindad 3x3 correspondiente a cada píxel y posteriormente ordenando los valores de dicha vecindad para encontrar la mediana. Una vez obtenida, la mediana se almacena en la posición correspondiente de la imagen de salida. Debido a que el algoritmo se suele ejecutar de forma iterativa, el programa host ejecuta el kernel en un bucle repetidamente. Por ello, entre ejecuciones del kernel se copia el resultado de procesamiento de la imagen de salida a la imagen de entrada para la siguiente iteración, con la consecuente copia de datos y latencia asociada.

1.3. Ejercicios

A continuación se presentan los ejercicios evaluables de esta práctica. Como norma de entrega, la práctica debe explicarse y mostrarse funcionando durante las horas de prácticas en el laboratorio siendo la fecha límite la última sesión de prácticas. Los ejercicios a realizar son descritos a continuación:

1.3.1. Ejercicio 0 (0 puntos) Creación y Ejecución del Proyecto

Esta primera versión del kernel está preparada para ejecutarse y una vez ejecutada comparar los resultados con la versión CPU. Si todo ha funcionado correctamente se mostrará por pantalla un mensaje de verificación y en el directorio del proyecto aparecerá la imagen procesada con el siguiente nombre: lena_1024_median.bmp.



```
c:\users\alber\onedrive\documentos\visual studio 2015\Projects\filtro_mediana\Release\filtro_mediana.exe

***TEST CORRECTO***

Tiempo ejecución GPU (Incluyendo transferencia de datos): 0.319790s
Tiempo de ejecución en la CPU : 3.739242s
Resultado escrito en lena_1024_median.bmp
```

Figura 1.4: Resultado de ejecución del programa básico.

1.3.2. Ejercicio 1 (1.5 puntos): Optimización de Transferencias entre GPU y CPU

Un problema común de las GPUs, y otros dispositivos aceleradores de cómputo, es la transferencia de datos entre memoria de la CPU y memoria del dispositivo. Dada su elevada latencia, puede ocurrir que toda la aceleración conseguida por la aceleración del cómputo en el dispositivo se vea ocultada por la alta latencia causada por la transferencia de datos entre memorias. Por lo tanto, esto se convierte en un aspecto importante a tener en cuenta para optimizar el tiempo de ejecución de una implementación GPU.

Observa que en el código proporcionado, al final de cada iteración del bucle, los datos son copiados desde memoria GPU hacia memoria CPU. Este paso puede ser evitado, con la excepción de la copia tras la última iteración del bucle. Podemos evitar esta transferencia de datos simplemente apuntando la dirección de memoria de entrada para nuestro kernel a la dirección de memoria de salida, donde se encuentran los resultados de procesar la imagen. Esta retroalimentación consigue que la imagen procesada sirva a su vez como entrada para la siguiente iteración, sin necesidad de copiar datos entre memorias.

Para conseguir esto vamos a realizar los siguientes cambios en el código:

- Eliminar todas las llamadas al comando `cudaMemcpy` de dentro del bucle.
- Reemplaza estas copias por un intercambio de direcciones en los punteros `d_input` y `d_output`.

- Añade una nueva llamada a `cudaMemcpy` justo al finalizar el bucle, de forma que tras la última iteración, los datos son copiados a la CPU para comprobación.

Una vez realizados estos cambios, ejecutad el código de nuevo y tomad nota de los tiempos obtenidos. ¿Se ha producido una mejora en el tiempo de ejecución? (Nota: para conseguir una mejora considerable en el tiempo de ejecución se necesita un número de iteraciones elevado: 250-500).

1.3.3. Ejercicio 2 (2.5 puntos): Acceso Coalescente a Memoria

Existe otro cuello de botella en nuestro código. La GPU puede procesar datos de forma más rápida cuando los hilos de ejecución de un kernel leen posiciones de memoria adyacentes, permitiendo que los accesos a memoria sean compartidos; esto se conoce como acceso coalescente a memoria. Sin embargo, en nuestro actual código, los hilos no leen direcciones de memoria consecutivas sino diferentes filas de memoria, por lo que no se da un acceso coalescente a memoria. La Figura 5 ilustra el efecto beneficioso de la coalescencia.

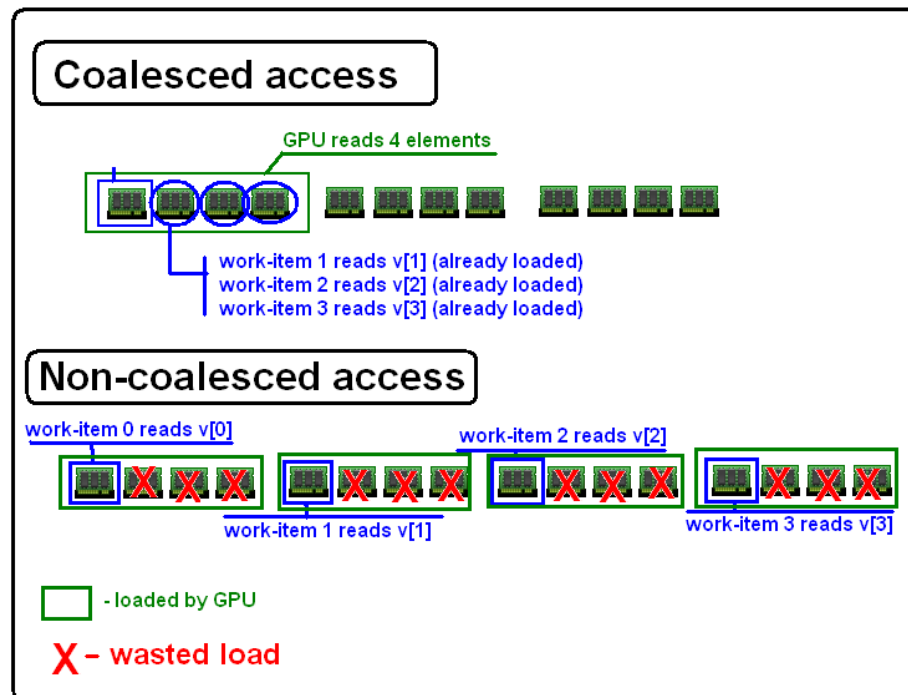


Figura 1.5: Comparación de acceso coalescente contra no coalescente.

Esto se puede solucionar descomponiendo el problema de otra forma. En lugar de tener hilos que trabajen sobre filas e iterar sobre los píxeles columna

dentro de una fila, podemos hacer que cada hilo trabaje sobre una columna e itere sobre las distintas filas que componen la imagen. Esto significa que los hilos leerán siempre posiciones consecutivas de memoria, permitiendo lecturas coalescentes.

Esta implementación debe llevarse a cabo sobre el kernel del código denominado `medianFilter1D_row`, el cual necesita los siguientes cambios:

- Calcular el índice columna (`col`) para cada hilo CUDA a partir de las variables especiales de CUDA.
- Calcular los índices de la vecindad del píxel que se está procesando (de igual manera que en la primera versión del kernel).
- Iterar sobre todas las filas de la imagen por cada hilo en lugar de iterar sobre las columnas.

Una vez realizados estos cambios, ejecutad el código de nuevo y tomad nota de los tiempos obtenidos. ¿Se ha producido una mejora en el tiempo de ejecución?

1.3.4. Ejercicio 3 (2.5 puntos): Optimización de Ocupación de la GPU

Pese a que hemos mejorado el tiempo de ejecución desde la primera versión que os proporcionamos, todavía nos encontramos con una versión subóptima. Esto se debe a que no creamos suficientes hilos para utilizar todos los SM disponibles en la GPU. Nuestra implementación actual se podría decir que tiene una baja ocupación de los recursos disponibles.

Los códigos implementados sobre la GPU de forma de general obtienen mejor tiempo de ejecución cuando hay un número mayor de hilos ejecutándose en paralelo, cada uno procesando una pequeña parte del algoritmo. Para conseguir esto en algoritmos de procesamiento de la imagen, lo lógico es que utilicemos un hilo por cada píxel en lugar de hilos que recorren las columnas o las filas de nuestra imagen. CUDA soporta la descomposición de un problema en 1, 2 y 3 dimensiones, para este caso lo más intuitivo y natural sería descomponer el problema en un mapa de 2 dimensiones donde cada hilo de ejecución procesará un píxel de la imagen.

En el código proporcionado encontrareis un kernel para llevar a cabo esta modificación, ver `medianFilter2D`. Para que funcione correctamente tendréis que hacer los siguientes cambios:

- Calcular el índice columna (`col`) utilizando las variables proporcionadas por CUDA, accediendo de esta forma a la columna global de la imagen.
- Hacer lo propio con la fila.

- Modificar los parámetros de invocación del kernel, teniendo en cuenta el nuevo tamaño en dos dimensiones que se le pasa a la función. Ahora el tamaño del grid y el tamaño del bloque poseen dos dimensiones, una para especificar el número de filas y otra el número de columnas.

Cabe destacar que en nuestra nueva versión del kernel ya no hay ningún bucle que itere sobre columnas o filas en cada hilo. Ahora cada hilo se encarga de procesar un único píxel de la imagen. Ejecutad de nuevo el kernel y observad la mejora en el tiempo de ejecución obtenida.

1.3.5. Ejercicio 4 (1.5 puntos): Investigando el Tamaño de Malla y de Bloque

Una vez que tenemos funcionando correctamente la versión en 2D, podemos probar a modificar ciertos parámetros de invocación del kernel y observar qué efectos tienen en el rendimiento de la aplicación. En particular investigad el efecto que produce alterar los tamaños del bloque y del grid. Estos están definidos como constantes al principio del código proporcionado. Algunas de las posibles configuraciones sugeridas, se muestran a continuación, también puedes probar a experimentar con otras.

Anchura Bloque	Altura Bloque	Hilos/Bloque	Anchura Malla	Altura Malla	Tiempo
1	1	1	1024	1024	
2	2	4	512	512	
4	4	16	256	256	
8	8	64	128	128	
16	16	256	64	64	
32	32	1024	32	32	

Explica brevemente como afecta cambiar el tamaño del grid y del bloque y cual es la configuración que mejor rendimiento ofrece.

1.3.6. Ejercicio 5 (2 puntos): Optimización con Memoria Compartida

El último día de prácticas se planteará este último apartado para aquél alumno que quiera optar a la máxima calificación en la práctica. Tendrá que desarrollarse durante el turno de prácticas correspondiente y consistirá en aplicar los conceptos explicados en teoría sobre la jerarquía de memorias en la GPU para optimizar la implementación 2D existente.