UNIVERSIDAD DE ALICANTE

CUDA WORKSHOP 2019

PRÁCTICAS

Sergio Orts Escolano (sorts@ua.es)

Albert García García (agarcia@dtic.ua.es)

José García Rodríguez (jgarcia@dtic.ua.es)

PRÁCTICA 3: FILTRO MEDIANA

Este ejercicio tiene como objetivo implementar en CUDA un filtro típico en procesamiento de imágenes. Se trata del filtro mediana. Para llevar a cabo la práctica se os proporciona una versión inicial del algoritmo funcionando, esta versión es una implementación sobre la GPU pero sin llevar a cabo ninguna optimización. Vuestra tarea es aplicar sobre el código proporcionado una serie de optimizaciones para mejorar el rendimiento.

EL ALGORITMO

En procesamiento de la señal, es a menudo deseable llevar a cabo procesamiento para reducir el nivel de ruido en la imagen o señal. El filtro mediana es una técnica de filtrado digital no lineal muy utilizada para eliminar ruido. Este tipo de preprocesamiento para reducir el ruido es un paso típico a llevar a cabo antes de aplicar otros algoritmos más complejos en visión por computador o procesamiento de la imagen: Filtros detectores de bordes, reconocimiento de formas, extracción de puntos característicos, etcétera.

La idea principal del filtro mediana es recorrer pixel a pixel toda la imagen, remplazando cada valor del pixel por la mediana de los valores de su vecindad, incluyéndose a sí mismo. Este patrón de vecindad se conoce como ventana en el área de procesamiento de la imagen. En nuestro caso aplicaremos una ventana de tamaño 3x3 sobre el pixel que estamos procesando. De esta forma nuestra vecindad estará compuesta por 9 elementos, siendo el 5º elemento el valor mediana tras su previa ordenación.

Este filtro se puede aplicar iterativamente, de forma que cuanto mayor número de veces aplicamos esta operación sobre la imagen original mayor ‘distorsión’ obtendremos sobre la imagen de entrada. Se ha de tener en cuenta que aplicar este filtro demasiadas veces puede conllevar otro efecto sobre la imagen de entrada. El efecto del filtro es el emborronamiento de la imagen de entrada, por lo que se si aplica

repetidamente el resultado obtenido, más allá de reducir ruido, es obtener una imagen emborronada. A continuación podemos ver el efecto del filtro sobre una imagen con cierto ruido, y el resultado tras aplicar repetidamente tras 5 iteraciones el filtro mediana.

Figura 1. Izquierda: Imagen con ruido. Derecha: Imagen tras aplicar filtro mediana.

OPTIMIZANDO EL ALGORITMO EN CUDA

Para llevar a cabo la práctica se os da una implementación en CUDA funcionando, el código se encuentra en el archivo *median.cu*. Dentro de este encontrarás 3 kernels distintos. El primero de ellos (medianFilter1D_col) está completo y funcionando; los otros dos se tendrán que completar a lo largo de los distintos apartados de la práctica. También se proporciona en el código funciones auxiliares para la entrada y salida de imágenes BMP así como medir los tiempos de ejecución. Finalmente, para comprobar que la solución en CUDA es correcta también se ha implementado una versión del filtro mediana para la CPU.

Para llevar a cabo este filtro con una ventana de 3x3 píxeles, es necesario añadir una región adicional sobre los bordes de la imagen, facilitando así el cálculo y evitando la comprobación de límites de tamaño durante el procesamiento. A esta región la llamaremos 'Halo' y estará compuesta por píxeles cuyo valor es 0. Para el código de la parte GPU, las imágenes se tratarán como arrays unidimensionales, esto cambiará la forma de direccionar el acceso a cada uno de los píxeles pero el formato de la imagen sigue siendo el mismo como veremos a lo largo de la práctica.

En la siguiente imagen se puede ver de forma visual el halo que se introduce en la imagen para facilitar el procesamiento de filtros sobre imágenes.

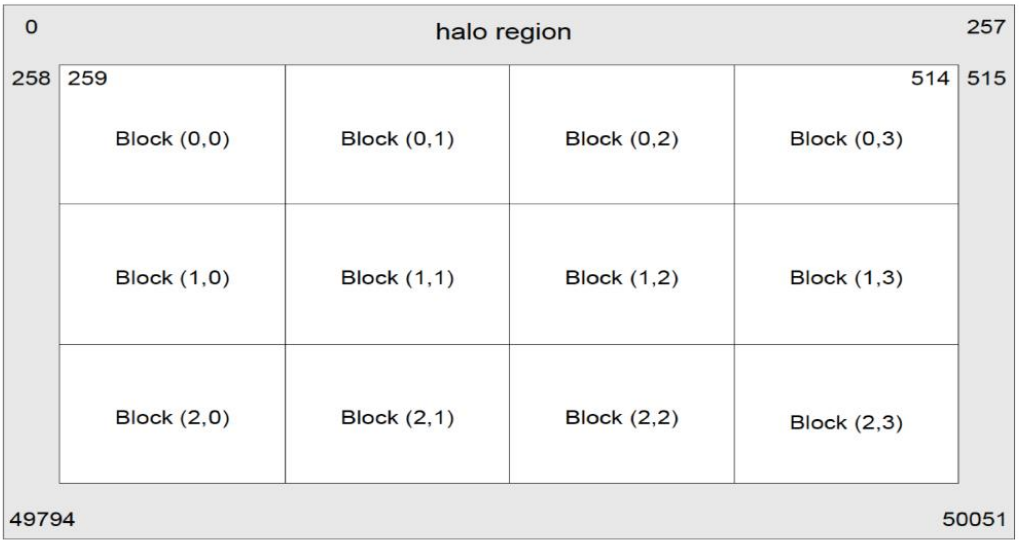

Figura 2. Halo introducido para una imagen de 256x256 píxeles. como se puede observar se introduce una fila/columna de ceros en los bordes de la imagen.

El kernel que se os proporciona ya solucionado (medianFilter1D_col), asigna cada fila de la imagen a un hilo en CUDA. El kernel itera sobre las columnas de la imagen, calculando los índices de la vecindad correspondiente y posteriormente ordenando estos para obtener la mediana. Una vez obtenida, la mediana se almacena en el vector que representa la imagen procesada.

Debido a que el algoritmo se puede aplicar iterativamente sobre la imagen de entrada, hay un bucle en el programa que ejecuta el kernel repetidamente. Entre ejecuciones del kernel se copia el resultado del procesamiento al buffer de entrada para la siguiente iteración. Esto conlleva la copia de datos desde memoria GPU a memoria CPU y con ello una latencia de tiempo importante.

EJECUTANDO EL CÓDIGO

Esta primera versión del kernel está preparada para ejecutarse y una vez ejecutada comparar los resultados obtenidos con la versión calculada en la CPU. Si todo ha ido bien, se mostrará por pantalla un mensaje de verificación y en el directorio del proyecto aparecerá la imagen procesada con el siguiente nombre: *lena_1024_noise.bmp*.

Antes de ejecutar, comprueba que en los parámetros de compilación del proyecto (CMakeLists) para código CUDA se ha especificado como arquitectura objetivo la versión necesaria para las tarjetas sobre las cuales se va a depurar o desplegar.

TRANSFERENCIAS DE DATOS ENTRE CPU-GPU

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. Estas suelen ser muy lentas, incluso puede ocurrir que toda la aceleración conseguida por el paralelismo 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 como en el código proporcionado, al final de cada iteración del bucle, los datos son copiados desde memoria GPU hacia memoria CPU. Esto 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.

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 puntos *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 su visualización.

Una vez realizados estos cambios, ejecuta el código de nuevo y toma 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 requiere un número de iteraciones elevado: ~250-500.

VERSION CON ACCESO COALESCENTE A MEMORIA (2 PUNTOS)

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.

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 los datos que el propio compilador nos facilita: *blockId.x, threadIdx.x, blockDim.x, …*
- *Calcular los índices de la vecindad del píxel que se está procesando. Nota: esto se lleva a cabo de igual manera que en la primera versión del kernel. Consultar código y entender acceso a vecindad.*
- Iterar sobre todas las filas de la imagen en lugar de las columnas

Prueba a ejecutar de nuevo el código invocando al nuevo kernel. ¿Mejora el rendimiento respecto a versiones anteriores?

MEJORANDO LA 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 descomponen el problema en un mapa de 2 dimensiones donde cada hilo de ejecución procesará un pixel de la imagen.

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

- Calcular el índice columna (col) utilizando las variables proporcionadas por CUDA: información del hilo y bloque de ejecución. De esta forma accediendo a la columna global de la imagen de entrada.
- Hacer lo mismo 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 posee dos dimensiones, una para especificar el número de filas y otra el número de columnas.

En vuestra nueva versión del kernel ya no hay ningún bucle que itere sobre columnas o filas. Cada hilo se encarga de procesar un único pixel de la imagen.

Ejecutar el nuevo kernel y observar la mejora en el tiempo de ejecución obtenida.

INVESTIGANDO EL TAMAÑO DEL GRID Y DEL BLOQUE

Una vez que tenemos funcionando correctamente la versión en 2D, podemos probar a modificar ciertos parámetros de invocación del kernel y observar que efectos tiene en el rendimiento de la aplicación. En particular puedes investigar los diferentes efectos en 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.

Block width	Block height	Threads per Block	Grid width	Grid height	Time
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	

Consideraciones:

- Para asegurar el correcto número de píxeles que se procesan, el grid y el bloque tienen que tener tamaños múltiplos de las dimensiones de la imagen. En este caso 1024x1024 píxeles.
- Para la arquitectura Fermi (antiguas GTX480), el tamaño total del bloque no debe exceder de 1024 hilos. En el caso de las GTX1050Ti podéis comprobarlo utilizando la utilizad DeviceQuery. Aunque puedes comprobar por ti mismo qué pasa si aumentas el número de hilos indefinidamente…

Trata de explicar brevemente cómo afecta cambiar el tamaño del grid y del bloque y cuál es la configuración que mejor rendimiento ofrece (y por qué).

UTILIZACIÓN DE LA MEMORIA COMPARTIDA

Dado que la actualización de cada pixel se basa en los valores de los píxeles vecinos, cada valor de un pixel es utilizado más de una vez en cada iteración. Esto significa que dentro de un bloque de hilos, dos hilos distintos están accediendo al valor de un pixel vecino de forma repetida, ocasionando dos accesos a memoria global y la latencia de tiempo que ello conlleva. Para evitar estos múltiples accesos a memoria por hilos de un mismo bloque, es más eficiente utilizar la memoria compartida por todos los hilos de un mismo bloque para almacenar los valores de los píxeles a procesar posteriormente. De esta forma el acceso repetido se llevará a cabo sobre esta memoria cache en lugar de sobre la memoria global y por tanto disminuirá considerablemente el tiempo de acceso.

Para la implementación en 2D, cada hilo de un bloque operará sobre su propia copia de memoria compartida, por ello es necesario previamente copiar estos datos a la memoria compartida, incluyendo el halo de la imagen. Operaremos con estos datos y finalmente el valor final del pixel se almacenará de nuevo en memoria global.

Para llevar a cabo esta implementación tendrás que hacer los siguientes cambios:

- Invocar al kernel con suficiente número de hilos, para no solo procesar la imagen, sino para poder copiar primero a memoria compartida todos los valores de la imagen y su halo.
- Calcular los índices globales y los índices locales a memoria compartida para cada hilo de ejecución CUDA.
- Sincronizar la copia de estos valores con el comando *__syncthreads(),* el cual sincroniza todos los hilos dentro de un mismo bloque.
- Utilizar los índices calculados previamente para acceder a los valores en memoria compartida y calcular la mediana de la vecindad.

Una vez realizados los cambios, compila y ejecuta de nuevo. Verás que no se produce una gran mejora en el tiempo de ejecución, incluso en alguno casos puede empeorar. Esta técnica es muy eficiente para algoritmos que reutilizan mucha información entre hilos. En nuestro caso solo estamos reutilizando 3 píxeles entre hilos adyacentes. Imagina que en lugar de una ventana de 3x3, trabajásemos con una ventana de 32x32, 64x64, 128x128, etcétera, el número de accesos a memoria realizados por dos píxeles adyacentes se incrementa considerablemente y por tanto el rendimiento al utilizar esta técnica también incrementa.

A Continuación se lista una serie de documentos y enlaces sobre programación CUDA.

Uno de los primeros cursos sobre programación en CUDA impartido por la universidad de Illinois:

The first course talking about CUDA programming in the world:

http://courses.ce.illinois.edu/ece498/al/

El manual official de programación CUDA de NVIDIA, el cual describe el modelo de programación, la sintaxis, una serie de consejos básicos para obtener más rendimiento, especificaciones técnicas, etcétera.

NVIDIA CUDA Programming Guide Version 9.1

http://developer.nvidia.com/nvidia-gpu-computing-documentation

Página official de NVIDIA:

CUDA Getting Started Guide (Windows)

Getting Started With CUDA SDK Samples

CUDA Toolkit 9.1 Release Notes & Erratas

CUDA C Programming Guide

Documento de buenas prácticas

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

Cursos de programación en CUDA online

https://eu.udacity.com/course/intro-to-parallel-programming--cs344

https://www.coursera.org/course/hetero