Tarjetas gráficas y aceleradores

FILTRO SOBEL

Procesado de imágenes BMP en C y CUDA

Primavera 2018

Antonio J. Cabrera Valenzuela
Paul Gazel-Anthoine

Propuesta de trabajo.

El filtro Sobel es un filtro que se utiliza para detectar zonas de alto contraste de luz en una imagen, como pueden ser los bordes de los objetos. Su utilidad se centra en la detección de aristas, aplicable a multitud de ámbitos.

Su funcionamiento consiste en estudiar los píxeles vecinos de cada píxel de la imagen, detectar cambios en sus valores y calcular la magnitud de esos cambios.

Para ello, primero pasamos la imagen a blanco y negro y de ahí hacemos el estudio de píxeles vecinos.

En dicho estudio, se realizan dos pasadas, una vertical y otra horizontal. Combinadas mediante el teorema de Pitágoras, se obtiene un valor de contraste para el píxel estudiado.

Una vez realizado el estudio para todos los píxeles de la imagen, tenemos una matriz de valores obtenidos. Para tener una representación equilibrada, debemos conocer el valor máximo y mínimo de los elementos de la matriz, para así ponderar cada uno de los píxeles en función de la magnitud de su cambio respecto a sus vecinos: el valor mínimo se representará como el negro (RGB 0,0,0), el valor máximo se representará como blanco (RGB 255,255,255) y el resto de valores obtendrán un gris (RGB x,x,x) donde la x será el resultado de multiplicar el valor obtenido en el estudio por 255 y dividiendo el resultado entre la resta de máximo y mínimo. Este número que multiplica el valor obtenido del estudio lo llamamos factor de contraste.

De esta forma se obtiene el filtrado Sobel de una imagen.

Nosotros vamos a aplicar este método en imágenes BMP, utilizando las rutinas de lectura y escritura proporcionadas por el profesor.

Nuestra intención es realizar una implementación funcional en C, primero de un programa que pase la imagen a escala de grises, y posteriormente de un programa que obtenga la imagen con el filtro Sobel aplicado, aunque la imagen original sea a color.

A continuación, pasaríamos a implementar kernels de CUDA para paralelizar la computación de píxeles y estudiar qué mejorías obtenemos. Queremos ser capaces de implementar el kernel por filas y elemento a elemento.

Además, el filtro Sobel requiere el cálculo de máximo y mínimo de una matriz que potencialmente tiene millones de elementos. Por ello, haremos dos etapas del código.

En una, más simple, daremos al usuario la posibilidad de insertar como parámetro un factor de contraste que evite tener que calcular el mismo. En la implementación final, por defecto se deberá realizar el cálculo del factor de contraste a partir del máximo y mínimo que se obtengan de realizar una reducción; también se mantiene la opción de parametrizar el factor de contraste.

Como las matrices de imagen pueden ser muy grandes, hemos realizado el estudio de una implementación en la que la parte final de la reducción de cálculo de máximo y mínimo se realiza en CPU y otra en la que se realiza una segunda reducción de un solo bloque.

Referencias:

Diapositivas de la asignatura (Teoría y laboratorio)
en.wikipedia.org/wiki/Sobel_operator
www.tutorialspoint.com/dip/sobel_operator.htm

NVIDIA CUDA Reductions (PDF)
https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjc-_ismt7bAhWBFRQKHXRVDMMQFgg7MAA&url=https%3A%2F%2Fdeveloper.download.nvidia.com%2Fcompute%2Fcuda%2F1.1-Beta%2Fx86_website%2Fprojects%2Freduction%2Fdoc%2Freduction.pdf&usg=AOvVaw1exgpHZqgfH6jt4P14HOUN

Implementaciones realizadas y rendimiento obtenido

En primer lugar, para estructurar nuestro programa CUDA, hemos empezado pasando el código secuencial que pasa una imagen a blanco y negro (BYN.c) a CUDA. Es un código muy simple y lo hemos implementado por filas (BYNFila.cu) y elemento a elemento (BYNElem.cu).

Cuando hemos hecho una implementación por filas hemos dedicado 1024 threads en el eje Y de cada bloque. Cuando hemos tratado elemento a elemento, los bloques los implementamos de 32x32. Somos conscientes de que puede no ser lo más eficiente. Pero para hacer un análisis consistente de los pasos que hemos seguido, hemos mantenido el tamaño lo máximo posible.

Aquí tenemos algunos datos:

- El tiempo medio del código en C era de 76ms, mientras que en CUDA por filas era de 84,7ms, es decir, más lento. Esto es en buena medida porque la imagen debe pasar de memoria principal a la GPU.
- El código CUDA elemento a elemento ha sido 2,3 veces más rápido que el código en C, pues el kernel es casi 16 veces más rápido que por filas.
- Estimamos unos 30ms en mover la imagen hacia la GPU y escribirla de vuelta observando estos tiempos. Efectivamente, utilizando el profiler, vemos que la lectura y escritura en memoria se realiza en 30,3 ms.

A continuación, no sin bastante trabajo y búsqueda de errores, hemos realizado la implementación del filtro Sobel.

El código en C (FinalSobel.c) fue sencillo de implementar y pasarlo a CUDA no resultó especialmente difícil cuando pasabamos el factor de contraste parametrizado, aunque tuvimos algún contratiempo con la sincronización de los datos que pertenecían a diferentes bloques.

Una vez salvados los problemas, teníamos la implementación sin cálculo de factor de contraste en CUDA por filas (ByNSobelFilasSinFactor.cu) y elemento a elemento (ByNSobelElemSinFactor.cu).

Estas fueron las conclusiones:

- El kernel elemento a elemento se ejecuta más de 76 veces más rápido que por filas, ya que para una imagen de aproximadamente 4000 píxeles de ancho, en el segundo caso cada

thread debe encargarse de 4000 píxeles, mientras que en el primero cada thread se encarga de de un píxel. Muchísimo más paralelismo nos da mucho más rendimiento.

- El cómputo total se realiza 4,5 veces más rápido.

Por útlimo, implementamos las dos versiones de CUDA donde se calcula el factor de contraste.

En una (ByNSobelReducCPU.cu) se realiza la parte final de la reducción recorriendo el vector de máximos y mínimos de cada bloque en CPU y en la otra (SobelFinal.cu) se realiza otra reducción de un solo bloque para obtener el resultado de máximo y mínimo en el thread con id = 0.

Estos han sido los datos extraídos:

- Respecto al código en C, el código que finaliza la reducción en la CPU es 9,7 veces más rápido; el código que realiza otra reducción final es 12,3 veces más rápido.
- En cálculos de kernel más finalización de la reducción, el código final es 1,35 veces más rápido que el que finaliza la reducción en la CPU.
- Aprovechando el profiler, hemos visto que el 90% del tiempo total se emplea en transferencias de memoria (host hacia device y device hacia host). Podemos intuir que el potencial de mejora ya es muy pequeño, pues estas imágenes inevitablemente deben pasar por el proceso.

Como ya se ha comentado, hemos utilizado el profiler de NVIDIA (nvprof) para sacar algunas conclusiones.

Aunque no son perfectamente comparables, hemos cogido un caso que es una clara muestra de ineficiencia (ByNSobelFilasSinFactor.cu) y nuestro código más eficiente (SobelFinal.cu), a ver qué podemos deducir de sus métricas.

Algunos datos llamativos, por ejemplo, son los de instrucciones por warp. En el caso ineficiente encontramos que cada warp debe ejecutar aproximadamente 270.000 instrucciones, mientras en el caso eficiente cada warp se encarga de unas 480 pese a tener que calcular, también, el factor de contraste (con dos kernels de reducciones). Esto se debe a que ocupamos mucho mejor la gráfica en el segundo caso y dividimos la tarea mejor entre los warps disponibles (mayor paralelismo).

También hemos notado un gran cambio en la métrica ‘sm_efficiency’. En el caso por filas obtenemos para los dos kernels un 14,7% y 14,5%. Una clara señal de que hay que ocupar mejor los

recursos. En el caso final, todos los kernels superan el 98,7% excepto el de la reducción final, que solo utiliza un bloque y evidentemente no obtiene una buena métrica (4,4%) pero no nos preocupa porque nos pareció la mejor forma de resolver el final de la reducción.

Para tomar los datos de los tiempos de ejecución hemos llamado 3 ejecuciones de programa con 4 imágenes diferentes por cada versión de código, en total 12 ejecuciones por código. Todas las imágenes son del mismo tamaño, BMPs de 3840x2160 píxeles (subaru, isla, carretera y NightKing).

Evidentemente, todas las pruebas realizadas se han hecho en el servidor Boada mediante el sistema de colas.

En definitiva, estamos contentos con el trabajo realizado. Nos hemos dado cuenta de lo que realmente hemos aprendido durante el curso y donde está nuestro nivel. Han sido muchas horas, aunque parezca mentira, dedicadas a estas versiones de código. Aun sabiendo que podíamos utilizar código ajeno para el proyecto, hemos partido de un documento en blanco para ver de lo que éramos capaces, guiándonos con la estructura de los códigos de las prácticas y consultando las dudas en los mismos y acudiendo al profesor en algún instante de desesperación. Esto nos ha permitido notar como ha evolucionado nuestra agilidad con el lenguaje CUDA en los días de dedicación al proyecto, que no es poca.

Además, los resultados de análisis han sido muy coherentes con las expectativas que cabría tener. La salida de imagen que nos da es perfecta, a nuestro juicio, y los tiempos logrados son un éxito. Si en lugar de procesar una sola imagen tuviéramos que procesar 10.000, estaríamos hablando de una diferencia notable, 6 minutos con SobelFinal.cu y 75 minutos para FinalSobel.c .

Somos conscientes de que no es perfecto, pero sí ha sido muy útil. Por ejemplo, la próxima vez que trabajemos con CUDA tenemos que intentar controlar más los posibles errores durante ejecución, deberíamos jugar con diferentes tamaños de bloque cambiando el SIZE definido arriba (que lo hemos mantenido a 32) y tratar de entender las consecuencias en ciertos códigos.

---

*Al ejecutar los códigos se necesita importar la imagen a la carpeta de trabajo, ya que las hemos retirado para la entrega.

**Los archivos job.sh contienen por defecto una ejecución con la imagen subaru.bmp de las que están en el Racó.

***Se adjunta una tabla con los tiempos obtenidos en las ejecuciones.