Proyecto final del curso GPGPU:
Filtro mediana

Introducción

La eliminación o disminución del ruido en una señal (por ejemplo una imagen) es crucial en muchos campos de aplicación. Muchos filtros de imágenes intentan disminuir el ruido mediante la sustitución de cada pixel por el resultado de una operación sobre una ventana cuadrada de pixels vecinos centrada en el pixel a modificar. De esta operación dependen las propiedades del filtro, como su capacidad para preservar los bordes de las figuras mientras se suavizan las zonas más homogéneas de la imagen.

El filtro mediana (*median filter*) consiste en sustituir el valor de brillo de cada pixel por la mediana de los valores dentro de la ventana de pixels vecinos. Este filtro se caracteriza por ser efectivo en la eliminación de artefactos de ruido de un solo pixel, causando solo una pequeña reducción en la nitidez de la imagen [1].

Objetivo

El objetivo general del laboratorio es realizar una implementación en CUDA del filtro mediana. La implementación debe buscar ser lo más eficiente posible, aplicando los principales conceptos vistos a lo largo del curso (maximizar el paralelismo, acceso coalesced a memoria global, uso de memoria compartida, etc.).

Para lograr el objetivo general deben cumplirse los siguientes objetivos específicos:

1. Desarrollar una implementación en CPU del filtro.
2. Desarrollar una implementación en CUDA que sirva como línea base. Esta implementación puede ser sencilla pero debe ser razonable, es decir, seguir las buenas prácticas de la programación en CUDA.
3. Desarrollar sucesivas versiones del filtro que aumenten la complejidad de la implementación buscando obtener un mejor desempeño. Las propuestas de optimización debe ser bien fundamentadas. Un ejemplo puede ser la mejora de la versión base introduciendo el uso de la memoria compartida.
4. Evaluar experimentalmente las distintas versiones sobre imágenes y ventanas de distinto tamaño.

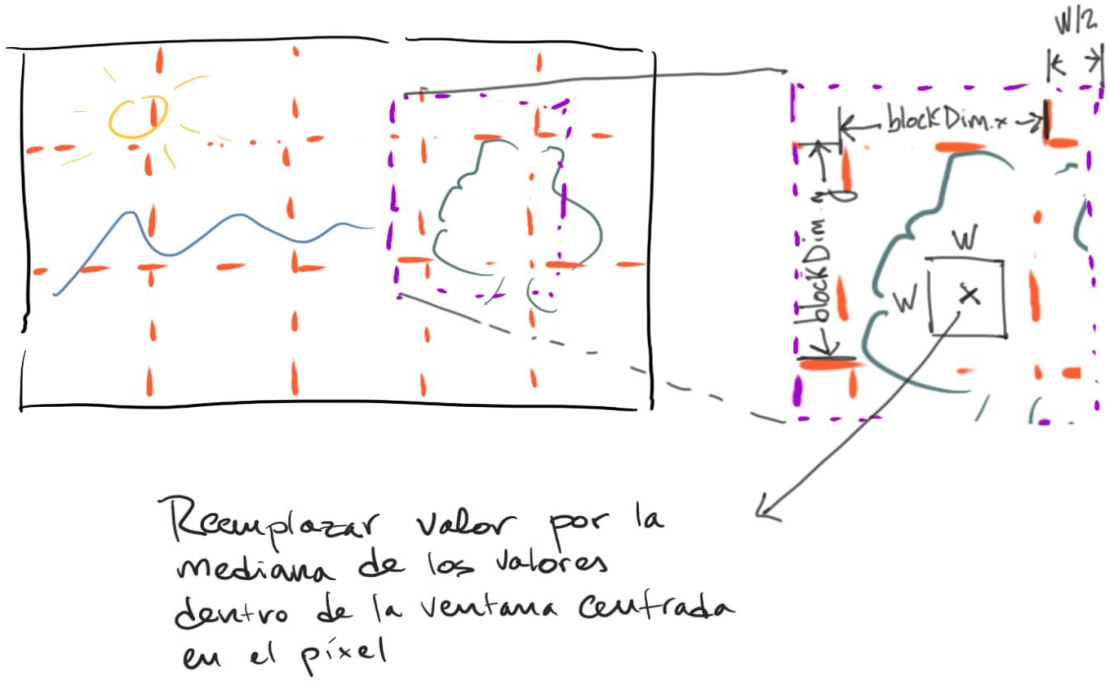
Consideraciones

La parte que concentra el costo computacional del algoritmo es el cómputo de la mediana para cada píxel. Esto implica ordenar los valores de la ventana centrada en el pixel y seleccionar el que queda en el centro. Para ordenar los valores, además es necesaria memoria auxiliar. Debido a esto, el costo computacional y de memoria aumenta considerablemente con el tamaño de la ventana.

Una forma directa de implementar el filtro puede ser asignar un hilo a cada pixel de la imagen, y por cada hilo disponer de un arreglo del tamaño de la ventana (WxW). El hilo deberá copiar los valores al arreglo, ordenarlos y seleccionar el valor central (la mediana).

Para ordenar el arreglo, puede ser interesante explorar el uso de las bibliotecas Thrust o CUB. También es interesante explorar la utilización de más de un hilo para ordenar. En el anexo se describe la implementación paralela del algoritmo Radix Sort.

Debido a que las ventanas correspondientes a pixels cercanos se superponen considerablemente, es deseable utilizar (en la versión optimizada) la memoria compartida con el fin de evitar cargar múltiples veces los mismos valores desde la memoria global. Notar que, como se aprecia en la imagen, un bloque de hilos que procesa una zona de la imagen debe acceder a pixels fuera de la zona a procesar. Por simplicidad, los elementos de la ventana que caen fuera de la imagen pueden considerarse con valor 0.



Para que el filtro sea efectivo, las ventanas deben tener un tamaño pequeño (W=3,5,7...). Cuanto más grande sea W, más ruido se elimina, pero también más detalles de la imagen se perderán. Para lograr el mejor desempeño, una idea es contar con implementaciones distintas según el tamaño de ventana. Queda a criterio del estudiante qué tamaños de ventana son soportados en su implementación (se valorará el soporte de múltiples tamaños de ventana).

Las imágenes tendrán un único canal (blanco y negro), y cada pixel tendrá un valor de brillo entre 0 y 255.

Material

Puede utilizarse cualquier biblioteca para manipular imágenes en blanco y negro. Para simplificar la tarea se proporciona código para manipular imágenes en formato pgm utilizando la biblioteca Cimg. También se proporcionan algunas imágenes de ejemplo.

Entrega

La entrega consiste en todo el código correspondiente a la solución de los ejercicios, un makefile, y un informe de hasta 8 páginas en pdf que contenga la descripción y justificación de las implementaciones, su evaluación experimental, y discusión de los resultados.

[1] RON BRINKMANN, CHAPTER FOUR - Basic Image Manipulation, The Morgan Kaufmann Series in Computer Graphics, The Art and Science of Digital Compositing (Second Edition), Morgan Kaufmann, 2008, 93-148, ISBN 9780123706386, https://doi.org/10.1016/B978-0-12-370638-6.00004-3.

Anexo A: Radix Sort

Uno de los algoritmos de ordenamiento más eficientes para ordenar claves cortas en procesadores paralelos es el radix sort. El algoritmo comienza considerando el primer bit de cada clave, empezando por el bit menos significativo. Utilizando este bit se particiona el conjunto de claves de forma que todas las claves que tengan un 0 en ese bit se ubiquen antes que las claves que tienen el bit en 1, manteniendo el orden relativo de las claves con mismo valor de bit. Una vez completado este paso se hace lo mismo para cada uno de los bits de la clave hasta completar todos sus bits.

Definimos la primitiva *split(input, n)* como la operación que ordena el arreglo **input** de acuerdo al valor **b** del bit **n** de cada elemento. Para implementar en GPU dicha primitiva se procederá de la siguiente manera:

- En un arreglo temporal **e** almacenar el valor de **not b** para cada posición **i** de **input**.
- Computar la suma prefija (exclusive scan) del arreglo. Ahora cada posición del arreglo contiene la cantidad de **f** de elementos de **input** con **b**=0 que hay antes que esa posición. Para los elementos con **b**=0, esta cantidad determina la posición en el arreglo de salida. El último elemento del arreglo de salida del scan contiene el total de posiciones con **b**=0 (hay que sumar 1 a este valor si la última posición tiene **b**=0), denominada **totalFalses**.
- Ahora se computa el índice de las posiciones con **b**=1 en el arreglo de salida. Para cada posición **i**, este índice será **t = i - f + totalFalses**.
- Una vez obtenidos los índices anteriores se graba cada elemento de **input** en el arreglo de salida en la posición **t** o **f** dependiendo de si **b** es 1 o 0.

Para implementar el algoritmo de radix sort utilizando la primitiva *split* simplemente debe inicializarse una máscara binaria para aislar el bit menos significativo, realizar el *split* del arreglo según ese bit, comprobar si el arreglo ya está ordenado y, si no lo está, hacer un shift a la izquierda de la máscara y volver a iterar. El procedimiento anterior se ejemplifica en la figura.