Programación masivamente paralela en procesadores gráficos (GPUs)

Informe - Práctico 4

Grupo 26

Nombre	**CI**
Agustín Matías Martínez Acuña	5.074.743-0
Natalie Valentina Alaniz Ferreira	5.209.018-4

Índice

1. Introducción

En este laboratorio se trabajará en la implementación de un filtro para disminuir el ruido en imágenes. El filtro utilizado intenta disminuir el ruido mediante la sustitución de cada píxel por el resultado de una operación sobre una ventana cuadrada de píxeles vecinos centrada en el píxel 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 consiste en sustituir el valor de brillo de cada píxel por la mediana de los valores dentro de la ventana de píxeles vecinos. Este filtro se caracteriza por ser efectivo en la eliminación de artefactos de ruido de un solo píxel, causando solo una reducción en la nitidez de la imagen a medida que aumenta el tamaño de la ventana [1].

1.1. Objetivos

El objetivo general del laboratorio es realizar una implementación en CUDA del filtro mediana [2]. Para ello el trabajo se dividirá en varias partes :

1. Desarrollo en CPU del filtro.
2. Desarrollo en CUDA que sirva como línea base.
3. Desarrollo de sucesivas versiones que aumenten la complejidad buscando obtener un mejor desempeño.
4. Evaluación experimental para imágenes de distintos tamaños con distintos tamaños de ventana.

El costo computacional del algoritmo depende de la eficiencia de la implementación del código que ordena la ventana de píxeles, la solución debe buscar aprovechar el paralelismo y reducir dentro de lo posible la cantidad de memoria auxiliar utilizada para el ordenamiento. Debido a lo anterior, el costo aumenta con el tamaño de la ventana. Las imágenes están en formato `.pgm` [3].

1.2. Especificación del entorno de prueba y del programa construido

El proyecto está desarrollado en el framework CUDA (C) y cuenta con un archivo `Makefile` para facilitar la compilación.

Las pruebas se realizan en la plataforma de supercomputación *Cluster.uy* [7], en particular, los procesadores gráficos utilizados son Tesla P40, con tamaño de memoria compartida de 48 KB por bloque. Se incluyen dos scripts para compilar y ejecutar en el Cluster.

1.3. Aspectos metodológicos del desarrollo

Versionado: Como se mencionó anteriormente, se realizan diferentes soluciones del filtro mediana, con distintos enfoques o herramientas. Para distinguir implementaciones conceptualmente distintas utilizamos la *major version* (el primer dígito en un número de versión). Aunque, por supuesto, pueden tener algún elemento en común. Luego entre *minor versions* se intenta experimentar o mejorar con algún ajuste o cambio, manteniendo la idea principal.

Pre-testing: Todas las versiones pasan por una pequeña etapa de verificación para decidir si aporta información relevante al proyecto (por ejemplo, resultados casi idénticos entre *minors*, o resultados muy malos pueden no ser relevantes). De no superar esta etapa, se comenta oportunamente y se descartan del análisis experimental.

Tiempo de ejecución: Para medir el rendimiento de cada solución se calcula la media y desviación estándar del tiempo de ejecución de diez muestras. Este enfoque permite discernir si los elementos agregados a la versión son (o no) beneficiosos.

2. Consideraciones generales

Hay decisiones de diseño que son comunes a todas las versiones, o que se engloban en un marco más general. Para no ser redundantes con esta información, se describen aquellos elementos de la solución que se usan en general:

Threshold: El texto de referencia [1] sugiere declarar un umbral tal que sí la distancia entre el píxel y la mediana es menor a ese umbral, entonces no se debe actualizar el valor. Después de experimentar con distintos umbrales se concluye (empíricamente) que las imágenes más limpias son aquellas con threshold=0, por lo que se descarta su uso.

Tipo de dato de entrada: Se cambia el tipo de dato de las imágenes de `float` a `unsigned char` (desde ahora `uchar`), esto se justifica porque el dominio de valores del formato `.pgm` es $[0, 255] \subset \mathbb{N}$, el tipo de dato `uchar` toma el mismo rango, haciéndolo conveniente para esta tarea. La ventaja principal de este cambio es reducir el tamaño de cada píxel de la imagen de 4 bytes a 1 byte, y aumentando la velocidad de acceso.

Máximo tamaño de ventana: A no ser que se indique lo contrario, todas las versiones fueron probadas hasta ventanas de tamaño 15, incluso podrían soportar tamaños más grandes, dado que cuentan con programación defensiva para los tamaños de bloques y el uso de memoria compartida, aunque no fue verificado exhaustivamente.

Cálculo automático de bloque: Para evitar desbordamientos, cada versión calcula el tamaño de bloque máximo para el tamaño de ventana. Si el tamaño de bloque elegido es mayor al máximo permitido, entonces se elige este último.

3. Líneas base

Para tener referencias de rendimiento iniciales, se crean dos líneas base en CPU y GPU (también referidas como baseline). Estas permiten establecer un punto de comparación para evaluar la efectividad de futuras optimizaciones y cambios en el algoritmo. Los resultados experimentales pueden consultarse en el Anexo 1.

3.1. Baseline CPU

El algoritmo recorre cada píxel de la imagen de entrada y aplica el filtro mediana utilizando una ventana cuadrada de tamaño W centrada en el píxel actual.

Proceso de Filtrado: Para cada píxel de la imagen, se construye un vector `neighborhood` que almacena los valores de los píxeles en la ventana definida por el radio. Luego se ordena y se selecciona el valor medio utilizando la función `std::nth_element` [4], que permite encontrar la mediana sin necesidad de ordenar completamente el vector utilizando el algoritmo *Introselect* [5].

3.2. Baseline GPU

La versión base del filtro mediana en *GPU* se implementó utilizando *CUDA*, configurada para procesar cada píxel de la imagen de manera paralela: Inicialmente, se asignan y copian las imágenes de entrada y salida a la memoria de la *GPU*. Se asigna un hilo a cada píxel de la imagen, por lo que cada hilo ordenará una ventana y devolverá la mediana. Estos accesos se hacen de manera coalesced.

Cada hilo construye un vector `neighborhood` para almacenar los valores de los píxeles en la ventana alrededor del píxel procesado, basándose en el radio de la ventana `W/2`.

Para calcular la mediana, el vector se ordena utilizando el algoritmo *Bubble Sort*. Se eligió el algoritmo con mejor rendimiento entre: Quick, Insertion y Bubble Sort.

A pesar de ser una implementación básica, se puede observar una mejora sustancial en comparación a la implementación secuencial, lo que muestra una clara diferencia entre la eficiencia de una implementación en

GPU y en *CPU* para este problema específico.

4. Versiones

En esta sección se detallarán las distintas soluciones implementadas y sus resultados.

4.1. Uso de memoria compartida (v1.0)

Dado que la memoria compartida es sustantivamente más rápida que la memoria global, tanto en latencia como en rendimiento [6], una mejora inicial sobre la implementación base fue modificar los accesos a la imagen en el kernel para que se realicen en memoria compartida. Cada *thread* en un bloque carga su píxel correspondiente a memoria compartida.

Cuando se necesita acceder a un valor de la imagen, primero se verifica si está en memoria compartida. Si es así, se accede desde allí, si está dentro de los límites de la imagen pero no en memoria compartida, es accedida a través de memoria global. Y si el valor buscado está fuera de los límites de la imagen se le asigna cero.

Una vez cargados los datos, los hilos extraen los píxeles de su ventana correspondiente directamente desde la memoria compartida, reduciendo el tiempo necesario para acceder a estos datos. Las ventanas son variables locales dentro del kernel y el ordenamiento es el mismo que la línea base (*Bubble Sort*).

El uso de memoria compartida implica estudiar los conflictos de bancos, especialmente cuando se usan datos de tamaño `1 byte`: se recuerda que un conflicto de banco ocurre cuando dos o más hilos en un *warp* intentan acceder a diferentes direcciones dentro del mismo banco de memoria al mismo tiempo. Cuando esto ocurre, las accesos se serializan, reduciendo el rendimiento.

En CUDA, normalmente se utilizan 32 bancos. El criterio para seleccionar el banco de memoria donde debe guardarse una `dirección` es `direccion mod 32`.

Se observa entonces que un conflicto de banco se genera si dos o más threads acceden a direcciones d_1 y d_2 tal que $d_1 \equiv d_2 (mod\ 32)$.

El uso de uchar (1 byte), puede ser propenso a tener conflictos de banco, se sospecha que esa fue la razón por la que el programa de ejemplo brindado por el equipo docente utiliza floats (4 bytes), porque con ese tipo de dato no se generan tantos. Sin embargo, en este contexto, los conflictos no son muy frecuentes, y tras efectuar pruebas, la reducción en el rendimiento es mínimo, por lo que se decide seguir trabajando con este tipo de dato.

4.1.1. Aumentando el uso de memoria compartida (v1.1)

En esta versión se mueven las ventanas de memoria global a memoria compartida. Se espera que a pesar de existir redundancia (un pixel dado puede estar cargado en varias ventanas), el rendimiento mejore con ventanas pequeñas, no se espera que escale muy bien con tamaños grandes de ventana, pues la cantidad de memoria compartida crecería.

4.1.2. Ajuste sobre la v1.1 (v1.2)

Esta *minor* es un pequeño ajuste de la versión 1.1, donde se elimina la copia de la matriz a memoria compartida, dejando solamente las ventanas. La justificación del ajuste es que la reducción de la cantidad de memoria auxiliar puede tener un impacto positivo en el rendimiento general.

Los resultados previos de esta versión fueron casi idénticos a los de la versión 1.1 (a menos de ruido estadístico), por esta razón se decide que esta versión no supera la etapa de pre-testing.

4.1.3. Evaluación experimental (v1.x)

La Figura 1 muestra el Tiempo de ejecución de las versiones 1.x.

En un principio, se confirma que estas versiones no escalan bien con ventanas grandes, pero tienen un rendmiento muy bueno en ventanas pequeñas ($W < 9$).

También se observa que el pasaje de ventanas a shared tiene un impacto favorable en el rendimiento.

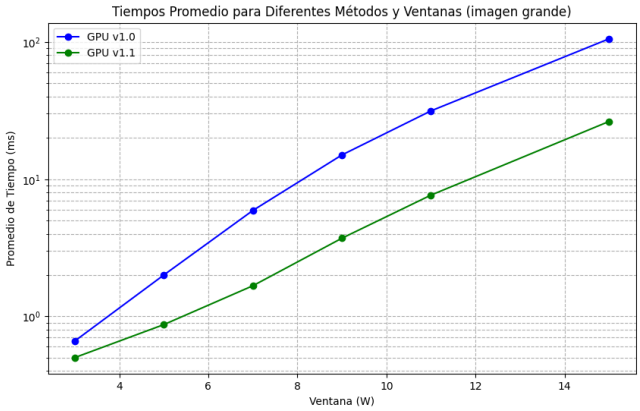

Figura 1: Comparación de promedio de tiempos por ventana para la versión 1.

4.2. Solucionando el problema con bibliotecas de paralelización general (v2.0)

Se desarrolla una implementación básica utilizando `Thrust`, cuya lógica consiste en ejecutar un `thrust::for_each` para generar las ventanas de forma similar a versiones anteriores. Se crea un arreglo de ventanas donde cada posición corresponde a un píxel, cada ventana es ordenada utilizando *Insertion Sort*, cambiando el algoritmo de ordenación porque realiza menos intercambios que *Bubble Sort* [8] y, al reducir los accesos a memoria, disminuye el tiempo de ejecución. Estos accesos a memoria muestran un mayor impacto al utilizar `Thrust` en comparación a los *kenels* implementados con `CUDA` puro. Finalmente, la ventana se reemplaza por su mediana y se copia el resultado a host.

4.2.1. Ordenamiento paralelo de ventanas utilizando pseudo-claves en un arreglo unificado (v2.1)

Para mejorar los resultados anteriores, se propone un nuevo enfoque, consistente en generar un solo (y gran) arreglo de ventanas, para ordenarlo en paralelo.

Tenemos que para toda ventana `i`, a cada píxel de ella se le aplica la función $F(p) = p + 256 \cdot i$, esto genera una pseudo-clave tal que si se ordenan todas las ventanas juntas, en el espacio [0 .. W*W-1] del arreglo unificado estará la `ventana[0]` ordenada, en `[W*W ... 2*W*W-1]` estará la segunda, y así sucesivamente...

Entonces, esta lógica permite ordenar el arreglo de ventanas un utilizando la primitiva `sort` de `Thrust`, donde las ventanas se ordenan internamente sin cambiar de posición. Al calcular la mediana, es necesario aplicar $F^{-1}(p) = p - 256i$ para obtener el valor original de la mediana.

La decisión de usar pseudo-claves fue para experimentar con la ordenación paralela de un gran arreglo, en lugar de ejecutar en paralelo la ordenación secuencial de varios arreglos.

Es importante tener en cuenta el tamaño de las imágenes utilizadas, ya que la mayor suma realizada en este enfoque en una ventana es `ancho * alto * 256`. Este valor puede ser muy grande para imágenes con alta resolución, por lo que se debe utilizar un tipo de datos que soporte esta operación. Se utilizó `uint` (4 bytes), que fue suficiente para las imágenes de prueba. La observación anterior implica que a medida que aumenta la resolución de la imagen se debe recurrir a tipos de datos más grandes para almacenar las pseudo-claves, lo que empeora el rendimiento.

4.2.2. Evaluación experimental (v2.x)

La Figura 2 muestra el tiempo promedio de ambas versiones. Entre las versiones, se observa que el ordenamiento paralelo de un solo arreglo escala mejor que el ordenamiento de varios arreglos.

Sin embargo, estas implementaciones tienen un rendimiento muy inferior a la versión 1.x, pero para grades tamaños de ventana la versión 2.1 tiende a cerrar la brecha.

La ventaja de `Thrust` es la sencillez y velocidad con la que se puede implementar un algoritmo paralelo con un *speedup* sustantivo con respecto a su contraparte secuencial. También abstrayéndose en gran medida de los conceptos fundamentales de la programación paralela. Los códigos de esta versión pueden ser comprendidos por un programador familiarizado con C y con el uso de bibliotecas generales.

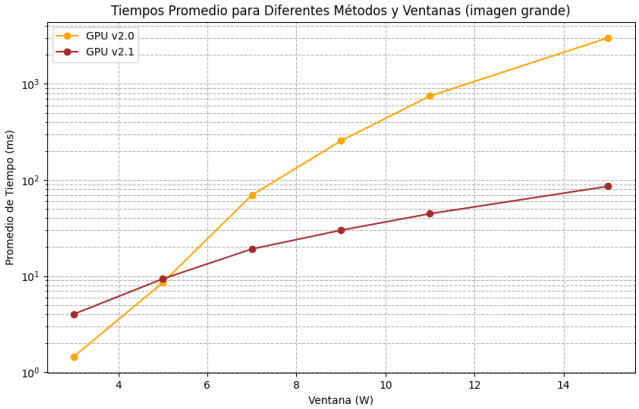

Figura 2: Comparación de promedio de tiempos por ventana para la versión 2.

4.3. Incorporación de un algoritmo de ordenación paralelo (v3.0)

En esta versión, se vuelve a una implementación de `CUDA` pura. En este caso se plantea ordenar las ventanas utilizando el algoritmo *Radix Sort*.

Para la implementación se toma como base lo realizado en versiones anteriores, enfocándose en el uso eficiente de la memoria ya que el algoritmo de ordenamiento, requiere la utilización de memoria auxiliar. La memoria compartida se organiza para contener tanto la porción de la imagen que cada bloque procesará, como las ventanas a ordenar y la memoria auxiliar que requiere el algoritmo de ordenamiento.

Figura 3: Organización de la memoria compartida

De esta forma cada hilo opera independientemente con su segmento de datos. El uso de memoria compartida se justifica porque el volumen de memoria requerido por cada hilo es relativamente bajo, correspondiendo a `W * W * 2 bytes` de memoria, siendo W un valor pequeño.

En la figura 4 se puede observar el tiempo promedio de esta versión. Para tener alguna noción comparativa, se agrega la versión 1.1 en ella, porque estas dos tienen un enfoque muy similar en el uso de memoria compartida y su principal diferencia es el algoritmo de ordenación utilizado. Para la v1.1 se utiliza *Bubble Sort* mientras que para la v3 se utiliza *Radix Sort*. El rendimiento de las dos implementaciones es muy similar, teniendo un poco de ventaja esta versión en ventanas de tamaño más grande.

Observación: Este algoritmo no soporta tamaños de ventana mayor que 11 para la imagen grande.

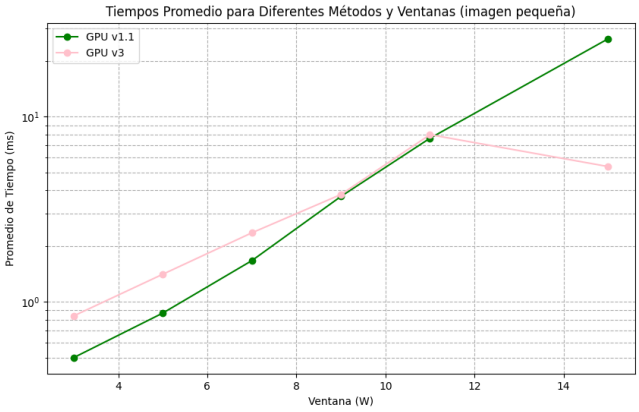

Figura 4: Comparación de promedio de tiempos por ventana: v3 vs v1.1

4.4. Un enfoque de ordenación con buckets (v4.0)

Dado que conocemos los elementos del dominio del arreglo a ordenar $[0, ..., 255] \subset \mathbb{N}$, se cumplen las precondiciones para utilizar un algoritmo de la familia Bucket Sort para hacer la ordenación en $O(n)$.

Para cada ventana, se reserva un arreglo con 256 entradas inicializadas en cero, y la i-ésima entrada de este corresponde a un contador de ocurrencias del elemento i en el arreglo a ordenar. En una sola recorrida del arreglo objetivo, se puede llenar el arreglo auxiliar con la información necesaria para ordenarlo.

Sin embargo, no es necesario ejecutar la ordenación, basta con recorrer condicionalmente el arreglo auxiliar de izquierda a derecha, acumulando la suma total del contenido de los *buckets*. Se termina de iterar cuando el contador llegue o supere la mitad del arreglo , y de ahí se conoce cuál fue el bucket que desbordó al acumulador, entonces se conoce el elemento en el medio del arreglo.

Dado que para cada píxel se reserva una cantidad de memoria constante correspondiente a los *buckets* (256 bytes) de cada ventana, comienza a compensar cuando el tamaño de ventana es cercano o mayor que la cantidad de *buckets* (por ejemplo a partir de `W = 11`). Entonces, en esta versión no se espera que tenga un buen rendimiento para `W` pequeños, pero sí se espera que sea competitivo para ventanas grandes.

4.4.1. Eliminando redundancia (v4.1)

En esta *minor* se mejora el proceso para obtener la mediana, en lugar de cargar el arreglo de ventanas y acumular el contenido de los *buckets* hasta la mitad, se cargan los datos directamente en los *buckets* y luego se acumula. Este cambio ahorra una iteración entera sobre el arreglo de ventanas, es decir, se elimina una operación de orden $O(W^2)$.

4.4.2. Unión de las dos anteriores (v4.2)

Esta versión es un pequeño experimento donde se vuelven a alojar los *buckets* en memoria global, pero con la técnica de cálculo de mediana de la versión 4.1. Esta versión está hecha para comprobar si el gran uso de memoria para ventanas pequeñas se contrarresta.

En la etapa de pre-testing los resultados fueron insatisfactorios, siendo incapaz de superar a cualquiera de sus versiones hermanas en ninguna de las pruebas. Debido a esto se toma la decisión de descartarla.

4.4.3. Evaluación experimental (v4.x)

La figura 5 muestra el rendimiento de la versión 4.0 y 4.1, donde se observa que la primera tiene mejor rendimiento para ventanas pequeñas. Pero a medida que aumenta el tamaño de ventana se destaca una estabilidad muy alta por parte del algoritmo 4.1, que termina superando ampliamente al primero.

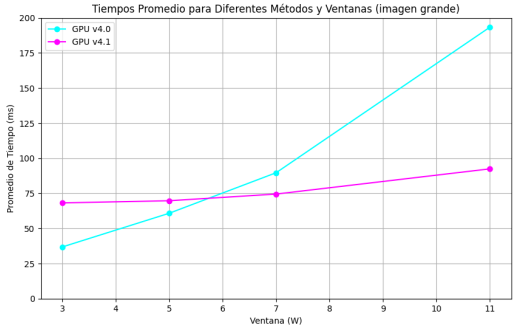

Figura 5: Comparación de promedio de tiempos por ventana para la versión 4.

5. Resultados finales

Los resultados finales se encuentran en los Anexos 1 y 2, el primero es un resumen que muestra la mejor ejecución de cada algoritmo para cada ventana, variando el tamaño de bloque. El segundo es una tabla con los resultados de todas las ejecuciones realizadas.

En esta sección nos interesa reflexionar sobre el desempeño de las distintas versiones mayores, ayudándonos con las figuras 6, 7 y 8, que muestran la comparación en escala logarítmica del tiempo de ejecución para las distintas imágenes de prueba.

Se observa una mejora significativa en el rendimiento al pasar de la versión baseline a la v1.x, la cual hace un uso básico de la memoria compartida. Esto destaca la importancia de aprovechar esta memoria de manera eficiente. Al pasar del uso básico de memoria compartida, al de la v1.1 que la aprovecha para almacenar las ventanas que posteriormente serán ordenadas, se logra otro salto en rendimiento.

Visto de manera holística, la familia 2.x no tuvo buenos resultados a pesar de utilizar una librería performante.

Las versiones v4.x muestran estabilidad a medida que aumenta el tamaño de la ventana, mientras que otras versiones como la v2.0, muestran un crecimiento acelerado en el tiempo de ejecución con el incremento en el tamaño de ventana.

Dado lo comentado en el párrafo anterior, y que estas implementaciones son más rápidas que el resto en términos de rendimiento con ventanas de mayor tamaño, son las opciones preferidas para trabajar con este tipo de ventanas. De manera similar, las versiones v1.1 y v3 muestran ser más efectivas para ventanas de tamaño pequeño, siendo las mejores soluciones en estos casos.

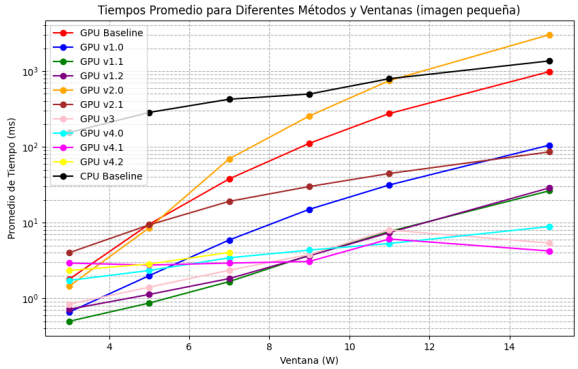

Figura 6: Comparación de promedio de tiempos por ventana por versión en imagen pequeña.

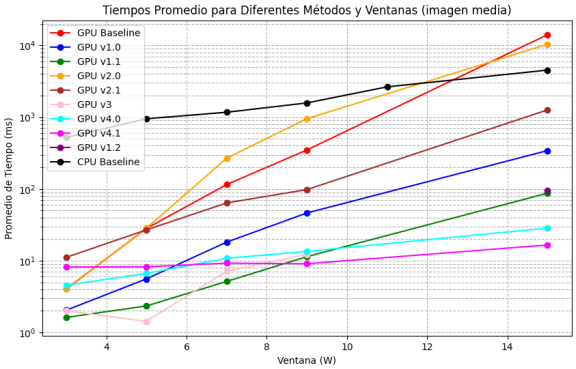

Figura 7: Comparación de promedio de tiempos por ventana por versión en imagen mediana.

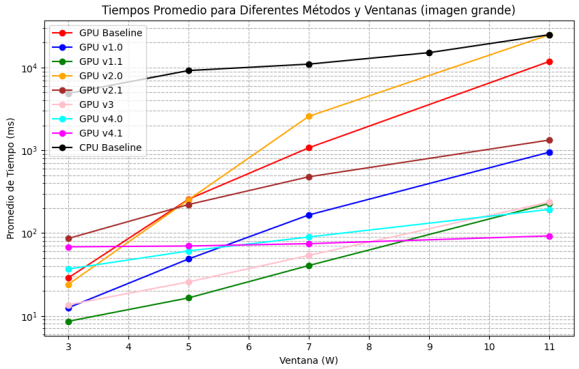

Figura 8: Tiempos por versión en imagen grande.

6. Conclusiones

En general, se logra el objetivo de implementar el filtro mediana en `CUDA`, experimentando con varias versiones, logrando un tiempo de ejecución entre 200 y 400 veces más rápido que la *baseline* en CPU.

Del cumplimiento del objetivo general se desprenden una serie de conclusiones particulares, presentadas a continuación:

A mayor tamaño de ventana, menor tamaño de bloque: En la mayoría de las versiones, cuanto mayor es el tamaño de ventana, mayor cantidad de memoria compartida debe reservarse. La memoria compartida por bloque está acotada por una constante, por lo que conviene que cada bloque trabaje con menos píxeles, para poder alocar las estructuras auxiliares necesarias sin sobrepasar ese límite.

Las bibliotecas generales de paralelización son sencillas de utilizar, pero hay que tener especial cuidado al diseñar una solución: En nuestro caso, las implementaciones con `Thrust` son de las que peores escalan de todas las realizadas. Se lograron realizar implementaciones puras que superan el rendimiento de ellas para todas las ventanas. Una propuesta a futuro es realizar una nueva versión con `Thrust`, intentando aprovechar mejor el abanico de operaciones que brinda la biblioteca.

En general, los esfuerzos por aumentar la complejidad en busca de un mejor desempeño son contraproducentes para los casos de uso realistas del filtro mediana ($W \in \{3, 5, 7\}$), los usos de ventanas más grandes se vuelven artificiales ya que las imágenes pierden sus detalles, por lo que se enfatiza en la siguiente conclusión final:

A veces, menos es más: la versión con mejor rendimiento para ventanas pequeñas es uno de los *kernels* más simples implementados en esta tarea, los esfuerzos realizados por aprovechar el paralelismo a través de estructuras más complejas implican mayor reserva de memoria que no se compensan para estos casos de uso realistas.

En el caso de ventanas más grandes, la v1.1 pierde competitividad frente a la v4.1, que utiliza *buckets* en memoria compartida y cuya reserva de memoria es grande, pero constante. A medida que aumenta el tamaño de la ventana la constante pasa a ser menos significativa, hasta que a partir de cierto tamaño, comienza a superar al resto de las implementaciones.

7. Referencias

Referencias

[1] R. Brinkman | Basic Image Manipulation, The Morgan Kaufmann Series in Computer Graphics | The Art and Science of Digital Compositing (Second Edition). <eva.fing.edu.uy/pluginfile.php/514757/mod_resource/content/1/median_filter>.

[2] P. Ezzatti, M. Pedemonte, E. Dufrechou | Letra del Laboratorio final | GPGPU 2024 | Fing | Udelar. <eva.fing.edu.uy/pluginfile.php/513982/mod_resource/content/7/letra2024.pdf>.

[3] J. Poskanzer | PGM Format Specification. <netpbm.sourceforge.net/doc/pgm.html

[4] nth_element | STD | C. <en.cppreference.com/w/cpp/algorithm/nth_element>.

[5] Introselect. <https://en.cppreference.com/w/cpp/algorithm/nth_element>

[6] E. Dufrechou , P. Ezzatti y M. Pedemonte | Programación CUDA II | GPGPU 2024 | Fing | Udelar. <eva.fing.edu.uy/pluginfile.php/504457/mod_label/intro/Clase%206%20-%20ProgramacionCUDA2_2024.pdf>

[7] Página web del ClusterUY. <cluster.uy>

[8] Insertion Sort vs Bubble Sort. <https://www.baeldung.com/cs/insertion-vs-bubble-sort>