

# Proyecto final

Nicolás Alejandro Núñez Hosco

Facundo Diaz

Grupo 17

Profesores: Martín Pedemonte, Pablo Ezzatti y Ernesto Dufrechou

Programación masivamente paralela en procesadores gráficos (GPUs)

Julio 2024

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Consideraciones</b>	<b>2</b>
<b>3. Implementación en CPU</b>	<b>2</b>
<b>4. Implementación en GPU</b>	<b>3</b>
4.1. Versión 1: Básica . . . . .	3
4.2. Versión 2: Uso de Shared Memory . . . . .	4
4.3. Versión 3: Radix Sort . . . . .	5
4.3.1. Radix Sort . . . . .	5
4.3.2. Median filter con Radix Sort . . . . .	5
4.4. Version 4: Quick Select . . . . .	6
4.4.1. Quick select . . . . .	6
4.4.2. Median filter con Quick Select . . . . .	6
4.5. Versión 5: Histogramas con ventana deslizante . . . . .	7
4.6. Versión 6: Algoritmo de Torben . . . . .	7
4.6.1. Algoritmo de Torben . . . . .	7
4.6.2. Median filter con algoritmo de Torben . . . . .	8
4.7. Algoritmos descartados . . . . .	8
<b>5. Resultados</b>	<b>9</b>
<b>6. Conclusiones</b>	<b>9</b>
<b>7. Bibliografía</b>	<b>10</b>
<b>8. Anexos</b>	<b>10</b>

## 1. Introducción

En el contexto del procesamiento de imágenes, el algoritmo median filter, es ampliamente utilizado como método para la disminución de ruido en una imagen.

En este informe se realizan distintas implementaciones de este algoritmo con el fin de que el mismo pueda ser ejecutado en GPU utilizando CUDA. Para esto, se toman en cuenta características específicas de la programación paralela en GPUs, como los accesos a memoria coalesced a memoria global, los conflictos de bancos en memoria compartida, divergencia de warps, nivel de paralelidad, entre otros.

Además, una parte crucial para el filtro es el cálculo de la mediana en sí, es por esto que se exploran diferentes algoritmos para computarla.

Para esto, se utilizarán diferentes imágenes de prueba que se encuentran en los anexos.

## 2. Consideraciones

- Durante la ejecución de los programas desarrollados, al utilizar el ClusterUy, se utilizan tarjetas Nvidia Tesla P100, las cuales utilizan compute capabilities 6.0. Es a considerar puesto que esta versión posee un máximo de 48KB de memoria compartida por bloque de hilos, lo cual limita el tamaño de bloques para algunos de los kernels implementados durante el desarrollo de este proyecto, dependiendo del tamaño de ventana a utilizar. Por esta razón, durante la ejecución, el tamaño de los bloques variará tomando en cuenta el tamaño de ventana, y la suposición de que se usa una tarjeta con esta limitante en el tamaño de memoria compartida.
- El programa es lanzado utilizando el comando *blur \* rutaAIImagen \* \*tamañoVentana\**. En caso de no indicar la ruta, se usará por defecto la imagen *por\_defecto.pgm* que debe estar en la misma carpeta que el archivo de ejecución, y, en caso de no indicar el tamaño de ventana, se utilizará 5.
- Por último, para simplificar los cálculos, solo se admiten tamaños de ventana impares. En caso de indicar un tamaño de ventana par en el momento de lanzar el programa, se le sumará 1 al valor ingresado.
- Con estas consideraciones hechas, el programa acepta tamaños de ventana de entre 1 a 27.

## 3. Implementación en CPU

Antes de implementar el filtro mediante GPU, se genera una implementación en CPU básica que nos servirá para comparar los resultados obtenidos.

Este algoritmo crea un vector de **unsigned char**, y recorre pixel por pixel de la imagen, donde, para cada uno de estos, agrega la ventana al vector creado anteriormente, luego, utiliza el algoritmo **Quicksort** (descrito más adelante) para encontrar la mediana de la ventana, y reemplaza el pixel por este valor.

Con este algoritmo, obtenemos los siguientes resultados:

CPU			
Figura	Tamaño de ventana (W)	Tiempo de ejecución ( $\mu$ s)	Desviación ( $\mu$ s)
1	3	33.617,8	366,553
2	3	123.117	1.936,91
3	3	1.182.110	5.717,36
1	7	215442	210.528
2	7	727744	642.523
3	7	9.478.100	4.153,09
1	11	432.763	827,817
2	11	1.474.920	3.036,36
3	11	18.733.447,82	13.879

Los imágenes resultantes con diferentes tamaños de ventana se encuentran en los anexos. Es importante notar que lo que busca este proyecto es mejorar el rendimiento de los tiempos obtenidos al aplicar el filtro, más no los resultados de estos en relación a la calidad final de la imagen, por lo que, a no ser que se mencione lo contrario, los resultados obtenidos por las siguientes implementaciones de las imágenes no variarán.

## 4. Implementación en GPU

### 4.1. Versión 1: Básica

En esta primera implementación en GPU se asigna el procesamiento de un píxel de la imagen a cada hilo. Luego, cada uno de estos copia en un arreglo los valores de píxeles de la imagen que se encuentren dentro de la ventana a procesar. Luego se utiliza un algoritmo de ordenamiento (En la versión inicial implementada, bubble sort) con el fin de facilitar la obtención del valor de la mediana dentro de la ventana.

Durante este copiado de la imagen cada hilo comenzará desde la esquina superior izquierda de su ventana correspondiente y recorrerá la misma por filas.

Dados dos píxeles adyacentes,  $A$  y  $B$  de forma que  $A$  se encuentre a la izquierda de  $B$ , se cumple que si la esquina superior izquierda de la ventana de  $A$  se encuentra en el índice  $V_a$ , entonces la correspondiente a  $B$  se encontrará en  $V_a + 1$ , luego, cuando los hilos asignados a cada uno de estos píxeles realicen la copia, se cumplirá que hilos contiguos dentro de un warp accedan a lugares de memoria contiguos, logrando acceso coalesced a los datos.

Notar que, para cada pixel, esta implementación requiere de  $O(n)$  accesos a memoria para el copiado de la ventana en el arreglo auxiliar y luego entre  $O(n^2)$  y  $O(n \cdot \log(n))$  accesos a memoria para ordenar el arreglo y finalmente  $O(n)$  accesos para escribir el resultado final.

Se obtienen los siguientes resultados:

Versión 1			
Figura	Tamaño de ventana (W)	Tiempo de ejecución ( $\mu$ s)	Desviación ( $\mu$ s)
1	3	3.271,1	53,7017
2	3	9.125,5	30,7905
3	3	51.326,6	26.163,4
1	7	122.264	15.401,6
2	7	373.318	1.139,24
3	7	3.961.060	51.326,6
1	11	1.033.940	1.105,2
2	11	26.729,8	75,7434
3	11	36.774.734,18	243.734,3808

## 4.2. Versión 2: Uso de Shared Memory

Esta implementación busca explotar el uso de memoria compartida para reducir los tiempos de acceso a memoria, dado que a través de la misma es posible realizar operaciones de lectura/escritura en con una latencia aproximadamente 100 veces menor que en el caso de memoria global no cacheada. Para esto, cada hilo copia primero su ventana a memoria compartida, para luego trabajar sobre esta, reduciendo así la cantidad de transacciones sobre memoria global.

En este caso, se identifican conflictos de banco cuando el tipo utilizado para representar la intensidad de cada pixel es **unsigned char**, y son resueltos cuando el tipo es **float**. Al utilizar estos últimos cada dato usa 4 bytes, es decir, una posición del banco, y elementos sucesivos usarán bancos diferentes. De esta forma, asumiendo que se utiliza ventana de  $3 \times 3$ , cuando todos los bancos acceden al primer elemento de su correspondiente ventana, el primer hilo del warp estará accediendo al banco 1, el segundo al banco 9 y así sucesivamente. La sucesión generada es la correspondiente a los múltiplos de 9 modulo 32. Esta sucesión no posee elementos repetidos dentro de un warp dado que 9 y 32 son números coprimos. En general, para todos los tamaños de ventana planteados se cumple que la cantidad de elementos total en cada ventana es coprima a 32, por lo que nunca ocurren conflictos de bancos.

Por otro lado, al usar **unsigned char**, estos usan 1 byte por elemento por lo que, los primeros 4 elementos estarán en un mismo banco, los siguientes 4 en otro banco, y así sucesivamente. En este contexto ocurre que existirán pares de hilos para los cuales, al acceder a determinados elementos de sus respectivas ventanas, el banco correspondiente coincide. Esto implica la existencia de conflictos de bancos, si bien los mismos son relativamente poco comunes para los tamaños de ventana utilizados, por lo que el impacto en el rendimiento es marginal.

Al experimentar utilizando floats en lugar de unsigned chars para este kernel en particular, notamos un empeoramiento en los tiempos de alrededor de un 30 %, una posible explicación de esto es que, al transferir un elemento del arreglo, se estarán accediendo 3 bytes que, pese al ocupar ancho de banda, son inútiles para la ejecución del programa, por lo que la mitigación de los conflictos de banco no representan una mejora sustancial cuando se tiene esto último en cuenta.

Por otro lado, fue planteado utilizar **quicksort** para el ordenamiento de la ventana luego de ser copiada a memoria compartida, sin embargo notamos un aumento en los tiempos de ejecución. Es posible que esto sea debido a que el algoritmo debe dividir al arreglo original en 2 partes de forma recursiva y, para distintas ventanas, estos sub-arreglos no necesariamente tendrán la misma cantidad de elementos lo cual causa que, distintos hilos del mismo warp deban ejecutar instrucciones distintas, esto es, que exista divergencia a nivel de warp. Bajo estas condiciones, se espera que el algoritmo anteriormente usado (bubble sort) de mejores resultados, aun teniendo una complejidad de  $O(n^2)$ , puesto que, aun teniendo una condición que causa divergencia, esta es "resuelta" inmediatamente después de efectuar una operación "swap" entre dos valores.

Este análisis es análogo para el resto de versiones presentes en este informe.

Se obtienen los siguientes resultados:

Versión 2

Figura	Tamaño de ventana (W)	Tiempo de ejecución ( $\mu s$ )	Desviación ( $\mu s$ )
1	3	853,5	50,443
2	3	2.258,6	1.500,48
3	3	8.800,8	494,295
1	7	1.967,1	154,535
2	7	5.556,1	227,732
3	7	48.208,5	18,1735
1	11	8.168,1	71,7332
2	11	26.729,8	75,7434
3	11	252.356,4545	1.809,628214

### 4.3. Versión 3: Radix Sort

Esta versión busca mejorar el tiempo de ejecución mediante el uso de un algoritmo de ordenamiento paralelo, reduciendo así el tiempo requerido para el cálculo de la mediana de cada ventana, que es la operación mas costosa del proceso de aplicado del filtro.

#### 4.3.1. Radix Sort

El algoritmo Radix sort es un algoritmo de ordenamiento que, además de ser considerablemente eficiente, es altamente paralelizable. El mismo consiste en separar progresivamente al arreglo original basándose en el valor del bit menos significativo de cada entrada.

Este algoritmo, en su versión secuencial tiene una complejidad temporal de  $O(w \cdot n)$ , donde  $n$  corresponde al numero de elementos y  $w$  a la cantidad de bits utilizada para representar los elementos de la lista.

#### 4.3.2. Median filter con Radix Sort

Para esta implementación, debido a que se busca utilizar un algoritmo de ordenamiento paralelo, cada píxel será procesado por un bloque de CUDA en vez de por un único hilo. Esto es así, además, porque este algoritmo requiere sincronizaciones entre hilos dentro de un mismo bloque usando `__syncthreads()`, lo cual, en caso de utilizar más hilos por bloques, de forma que un mismo bloque procese varios pixeles, sincronizaría hilos los cuales no están trabajando en conjunto, perjudicando el tiempo de ejecución.

Por lo tanto, en la implementación paralela realizada del algoritmo Radix Sort, a cada hilo se le asigna una posición del arreglo de entrada, que queda determinada por la variable `id_local = threadIdx.y * blockDim.x + threadIdx.x`, observar por lo tanto que, debido a que el largo del arreglo a ordenar se ve determinado por el tamaño de ventana, el tamaño de bloque también será determinado a partir de este parámetro, de forma que todos los hilos de cada bloque tengan un elemento asignado.

Luego, se realiza un bucle con tantas iteraciones como bits sean utilizados en la representación de los valores a ordenar, en este caso, 8. En cada iteración se ejecutará de forma paralela un split del arreglo para el bit correspondiente.

La ejecución del split requiere de 3 partes:

- En primer lugar se obtienen los valores negados de los bits en la posición correspondiente según la iteración, que son almacenados en el arreglo `f`.
- Luego, una vez todos los hilos del bloque hayan terminado con la operación anterior (para asegurar que esto sea así se hace uso de la primitiva `__syncthreads()`), se realiza un exclusive scan sobre el arreglo `f`. Durante el desarrollo, se determinó que realizar este scan de forma paralela empeoraba los tiempos obtenidos, posiblemente debido al tamaño pequeño de los arreglos sobre los que se aplica.
- Finalmente se sincronizan nuevamente los hilos, se realiza el calculo del total de falsos, sumando el valor final del escaneo y el valor negado del bit actual y se determina el índice de destino `d` para cada entrada del arreglo.

Finalmente, una vez completada la ejecución de radix sort, el valor final del píxel corresponde al valor de la mediana del arreglo que se encuentra en el índice  $W * W / 2$  del arreglo ordenado, donde  $W$  es el tamaño de ventana.

Notar que la cantidad de hilos utilizados por esta implementación crece de forma muy rápida al aumentar el tamaño de la ventana y la resolución de la imagen.

Se obtienen los siguientes resultados:

**Versión 3**

<b>Figura</b>	<b>Tamaño de ventana (W)</b>	<b>Tiempo de ejecución (<math>\mu</math>s)</b>	<b>Desviación (<math>\mu</math>s)</b>
1	3	3.744,5	1.826,58
2	3	9.783,3	1.062,17
3	3	71.749,3	164,069
1	7	6.067,7	95,5895
2	7	19.818,2	44,9291
3	7	183.265	29,4824
1	11	13.026,3	66,433
2	11	43.393,2	27,2348
3	11	403.859,2727	310,5276448

#### 4.4. Version 4: Quick Select

En esta versión se toma un acercamiento diferente. En lugar de intentar optimizar el tiempo de ejecución de ordenamiento del arreglo, se intenta aplicar un algoritmo específico para la búsqueda de la mediana.

##### 4.4.1. Quick select

El algoritmo Quick Select permite encontrar el k-ésimo elemento más pequeño de una lista. Este algoritmo tiene una complejidad promedio de  $O(n)$ , por lo que, si bien no es paraleizable, resulta más eficiente que utilizar un algoritmo de ordenamiento tradicional. Este algoritmo funciona con un acercamiento parecido al algoritmo Quicksort para ordenamiento de arreglos. Realiza varias iteraciones, tomando un pivot random del arreglo (aunque en nuestro caso, el pivot sera el último elemento), contando la cantidad de elementos menores a este y colocándolos en una posición anterior, y, en caso de que hayan  $\frac{\text{tamañoDelarreglo}}{2} - 1$  elementos menores, lo devuelve como el valor de la mediana, mientras que en el caso contrario, repite el mismo proceso, para el subarreglo compuesto de los elementos menores o mayores, dependiendo de la cantidad de estos. Sin embargo este algoritmo no es paraleizable.

##### 4.4.2. Median filter con Quick Select

La implementación de esta versión sigue la línea propuesta para la versión 2, en la cual se asigna un hilo a cada pixel de la imagen a procesar y se emplea memoria compartida para reducir el costo de los accesos a memoria. Cada hilo copia entonces de forma similar a la versión 2 la ventana a memoria compartida, para luego aplicar el algoritmo Quickselect sobre esta.

Se obtienen los siguientes resultados:

**Versión 4**

<b>Figura</b>	<b>Tamaño de ventana (W)</b>	<b>Tiempo de ejecución (<math>\mu</math>s)</b>	<b>Desviación (<math>\mu</math>s)</b>
1	3	1.366,2	1.726,98
2	3	2.113,7	1.642,03
3	3	8.680	402,607
1	7	1.097,1	98,4992
2	7	2.459,8	213,734
3	7	17.824,3	29,1092
1	11	1.703	122,237
2	11	5.326,6	176,328
3	11	40.341,81818	293,4756611

#### 4.5. Versión 5: Histogramas con ventana deslizante

En el paper '*A fast two-dimensional median filtering algorithm*' [3] se detalla una posible implementación del algoritmo que busca utilizar histogramas para realizar el cálculo de la mediana y, además, explotar la geometría del problema a resolver y las propiedades de los histogramas para realizar esta operación con la menor cantidad de cálculos posible. El algoritmo consiste en calcular, para cada píxel, el histograma correspondiente a la ventana centrada en el mismo y obtener de la mediana con esta información. Luego, se desplaza la ventana un píxel hacia la derecha y se repite el cálculo. Sin embargo, la obtención del nuevo histograma puede realizarse de forma eficiente al restar del histograma anterior los valores correspondientes a la primera columna de la ventana anterior, y sumando los valores de la columna agregada. Como consecuencia, el cálculo de la nueva mediana es de orden  $O(v)$  para una ventana de  $v \times v$ , en vez de ser  $O(v^2)$  como ocurre con quickselect.

El objetivo de esta versión es adaptar la implementación detallada en el documento mencionado a CUDA.

Para esto se realizan algunos cambios. Por un lado, se utilizará una ventana con desplazamiento vertical en vez de horizontal, con el fin de que los accesos a memoria sean coalesced, además cada hilo procesará por lo tanto una porción de columna de pixeles.

Notar que el grado de paralelismo de esta implementación depende en gran medida de la cantidad de columnas existentes en la imagen y la cantidad de segmentos en las que se divide cada una de ellas. En el caso borde en el cual cada columna se divide en tantas secciones como pixeles tenga la misma, el orden del cálculo de la mediana será  $O(v^2)$  puesto que no se está explotando la geometría del problema (no habría desplazamiento de ventana). Se observa como el grado de paralelismo del cual se beneficia esta implementación es, por lo tanto, limitado.

Se obtienen los siguientes resultados:

Versión 5			
Figura	Tamaño de ventana (W)	Tiempo de ejecución ( $\mu s$ )	Desviación ( $\mu s$ )
1	3	4.222,8	127,455
2	3	11.005,4	109,374
3	3	97.743,6	87,5471
1	7	3.989,5	84,5646
2	7	10.118,2	175,19
3	7	95.327,4	28,7526
1	11	4.301,5	81,5724
2	11	10.436,7	136,839
3	11	96.835,72727	299,2393994

Se observa que el aumento en el tiempo de ejecución al modificar el tamaño de ventana es notablemente bajo.

#### 4.6. Versión 6: Algoritmo de Torben

Esta versión utiliza un algoritmo conocido como "Algoritmo de Torben", desarrollado por Torben Mogensen el cual permite calcular la mediana de un arreglo utilizando únicamente lecturas sobre el mismo, eliminando la necesidad de aislar los arreglos utilizados para las ventanas que procesará cada hilo, permitiendo que todos los hilos de un mismo bloque trabajen sobre los mismos datos sin alterar el resultado de los demás.

##### 4.6.1. Algoritmo de Torben

El algoritmo de Torben comienza haciendo una estimación del valor de la mediana utilizando el valor máximo y mínimo del arreglo:  $guess = \frac{min+max}{2}$ , luego recorre el arreglo múltiples veces,

contando la cantidad de ocurrencias de elementos menores, mayores e iguales al valor estimado, y dependiendo de estos valores, genera una nueva estimación, hasta terminar con el valor exacto.

#### 4.6.2. Median filter con algoritmo de Torben

Se comienza copiando los pixeles alcanzados por todos los hilos del bloque a memoria compartida, moviendo el bloque de hilos con un desplazamiento de **blockDim.x** y **blockDim.y**, de forma que cada elemento se copie de memoria global a memoria compartida una única vez, consiguiendo además una lectura coalesced. Cada hilo aplica entonces el algoritmo de Torben sobre su ventana, comenzando con la estimación del valor. Luego, en cada recorrida, una vez contada la cantidad de elementos menores, mayores e iguales, en caso de que  $less \leq \frac{tamano}{2}$  y  $greater \geq \frac{tamano}{2}$ , entonces se halló el valor de la mediana, en caso contrario, se hace una nueva estimación y se repite el proceso.

Aún teniendo una complejidad de **O(n)**, esta algoritmo es más lento que quickselect, sin embargo, el uso de este algoritmo reduce la cantidad de accesos a memoria global, necesitando únicamente una lectura por elemento en la matriz.

Se obtienen los siguientes resultados:

Versión 7

Figura	Tamaño de ventana (W)	Tiempo de ejecución ( $\mu s$ )	Desviación ( $\mu s$ )
1	3	892,9	50,6852
2	3	1.968,2	227,788
3	3	11.007,5	390,492
1	7	1.543,2	129,302
2	7	4.154,8	181,673
3	7	31.523,2	194,939
1	11	2.694,9	105,097
2	11	8.166,3	150,91
3	11	68.789,7	14,4457

#### 4.7. Algoritmos descartados

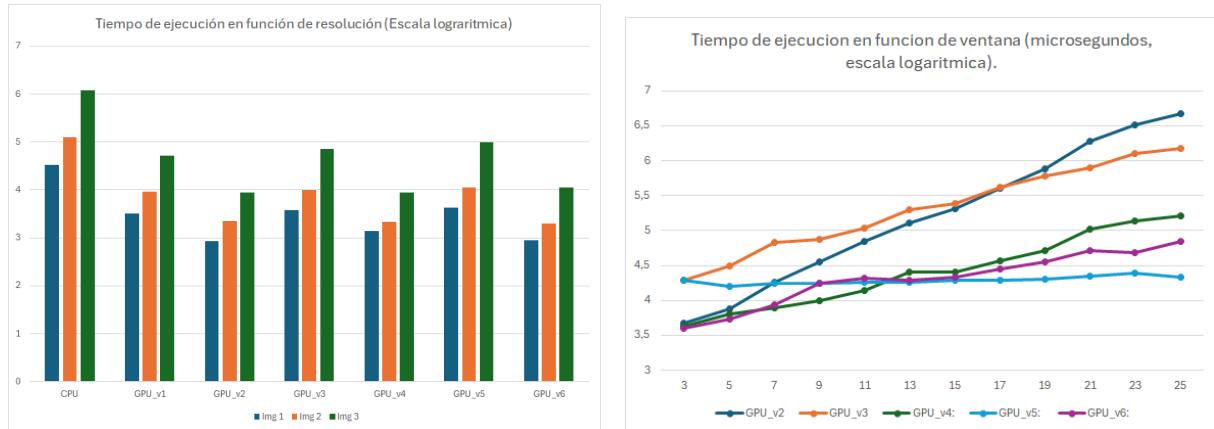
Por un lado, se descartó una aproximación utilizando paralelismo dinámico puesto que el uso de **cudaDeviceSynchronize()** dentro de kernels fue deprecado en la versión 11.6 de CUDA, lo cual nos imposibilitaba la sincronización de un kernel con un kernel hijo, agregando pasos extras a esta aproximación. Por ejemplo, una posible idea sería que un hilo genere la ventana, y esta sea pasada a un kernel hijo que utilice varios hilos para ordenarla, sin embargo, el problema anteriormente dicho junto con la imposibilidad de utilizar memoria compartida como argumento (lo cual es posible con funciones de tipo **\_\_device\_\_**) terminaban resultando en tiempos mayores.

Por otro lado, se descartaron algunas implementaciones que hacían uso de bibliotecas como **Thrust** y **CUB**, puesto que, aún tomando en cuenta las optimizaciones que estas ofrecen, el tiempo añadido al agregar pasos adicionales a su uso era considerablemente mayor que las ganancias obtenidas.

Por último, se descartó una versión que generaba un histograma por pixel para mantener la brevedad del informe, y dado que no presentaba mejoras considerables respecto a otras versiones ya implementadas, manteniéndose en la linea de los resultados ya obtenidos. Aun así, se agrega a los anexos los resultados obtenidos por esta implementación.

## 5. Resultados

A continuación, se muestran los datos sintetizados en 2 gráficas para tamaños de ventana desde 3 a 25.



Estos resultados demuestran que el uso de GPU para la aplicación de este filtro, como es de esperar, logra mejoras notables en tiempo de ejecución, logrando una reducción del 99.7847 % en el tiempo de ejecución para imágenes y ventanas grandes respecto a la versión en CPU.

Se observa que, para imágenes pequeñas, la versión 2 consigue tiempos de ejecución menores, sin embargo, sus tiempos de ejecución aumentan rápidamente al incrementar la resolución de la imagen procesada o el tamaño de ventana. Por otro lado, la versión 4 resulta preferible para tamaños de imagen grandes y ventanas pequeñas o medianas.

Las versiones 4 y 6 obtienen resultados comparables. Entre ellas, la versión 4 parece tener resultados marginalmente superiores para los tamaños de imagen y ventana analizados.

Debido a las características antes mencionadas del algoritmo de Torben, es posible que este algoritmo (versión 6) sea preferible en dispositivos cuya velocidad de acceso a memoria global sea menor. La versión 5 es más lenta que las anteriormente mencionadas para tamaños de ventana pequeños y medianos, pero su tiempo de ejecución se mantiene más estable frente a cambios en este parámetro, esto es consistente con el análisis teórico desarrollado anteriormente.

Se puede notar como al aumentar el tamaño de ventana por encima de 15, esta última versión logra superar los tiempos de ejecución de las demás versiones, siendo esta la versión óptima si se pretende aplicar el filtro mediana para grandes tamaños de ventana.

## 6. Conclusiones

Se concluye que existen una variedad de algoritmos paralelos que mejoran sustancialmente el tiempo de ejecución del algoritmo *Median Filter*.

El algoritmo indicado a utilizar dependerá de los valores de ventana y resolución sobre los cuales se quiera trabajar, como se detalla en el análisis de resultados.

Las características específicas del hardware utilizado tienen un impacto directo en el diseño y rendimiento de los algoritmos implementados. En este caso, los programas hacen uso extensivo de la memoria compartida y, por lo tanto, esta característica causa que deba variarse el tamaño de bloque a utilizar y limite el tamaño de ventana máximo posible para algunos algoritmos.

## 7. Bibliografía

(1) Quickselect alghorithm.

<https://www.geeksforgeeks.org/quickselect-algorithm/>

(2) Median filter

[https://eva.fing.edu.uy/pluginfile.php/514757/mod\\_resource/content/1/median\\_filter](https://eva.fing.edu.uy/pluginfile.php/514757/mod_resource/content/1/median_filter)

(3) T. Huang, G. Yang and G. Tang, "A fast two-dimensional median filtering algorithm" in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 27, no. 1, pp. 13-18, February 1979, doi: 10.1109/TASSP.1979.1163188.

## 8. Anexos



Figura 1: Imagen sin filtro de tamaño 1140x320

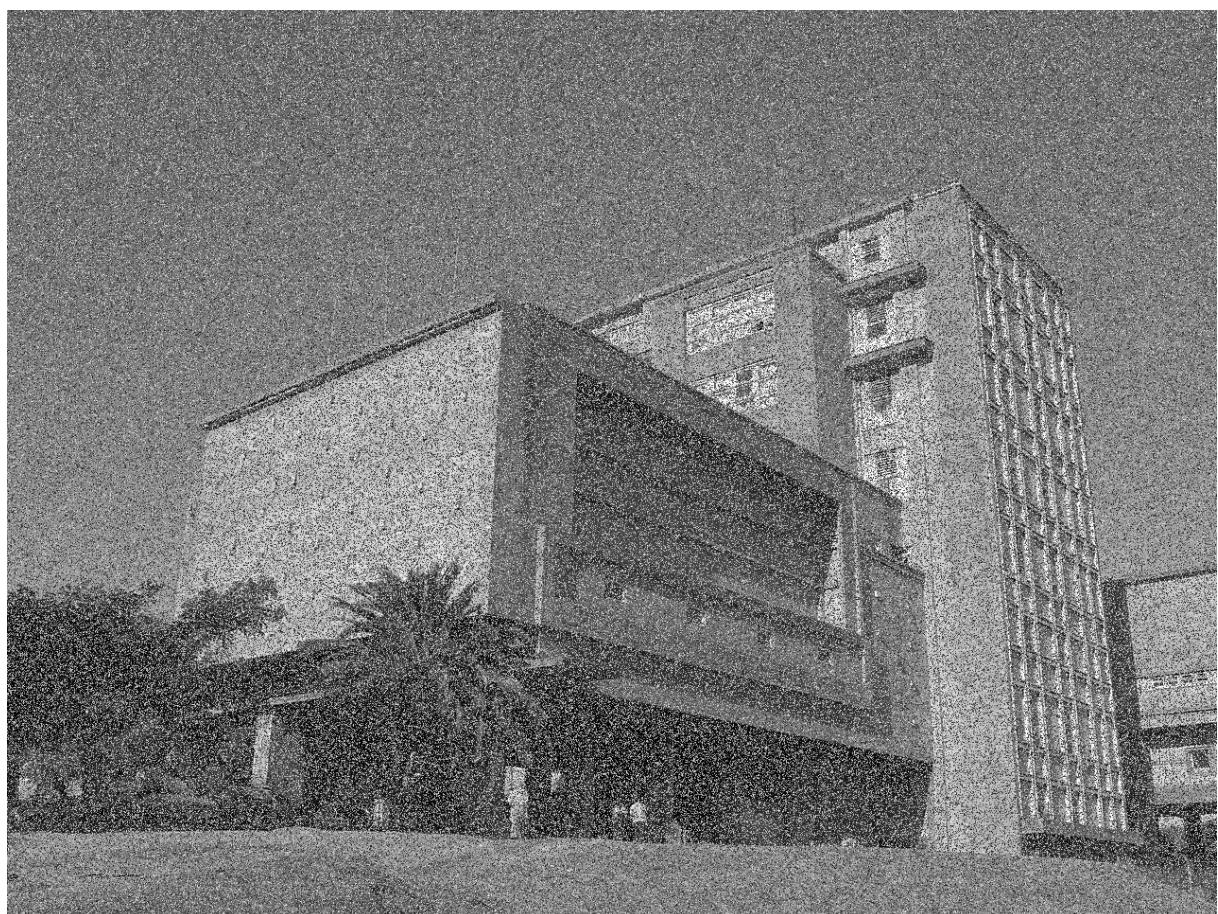


Figura 2: Imagen sin filtro de tamaño 1280x960



Figura 3: Imagen sin filtro de tamaño 4410x2609



Figura 4: Figura 1 con filtro mediana, W=3



Figura 5: Figura 1 con filtro mediana, W=7



Figura 6: Figura 1 con filtro mediana, W=11



Figura 7: Figura 2 con filtro mediana, W=3



Figura 8: Figura 2 con filtro mediana,  $W=7$



Figura 9: Figura 2 con filtro mediana, W=11



Figura 10: Figura 3 con filtro mediana, W=3



Figura 11: Figura 3 con filtro mediana, W=7



Figura 12: Figura 3 con filtro mediana, W=11

**Versión descartada con histogramas**

<b>Figura</b>	<b>Tamaño de ventana (W)</b>	<b>Tiempo de ejecución (<math>\mu</math>s)</b>	<b>Desviación (<math>\mu</math>s)</b>
1	3	1.301,2	101,592
2	3	2.975,8	253,599
3	3	23.591,8	312,524
1	7	1.447,5	94,6458
2	7	3.670,3	255,716
3	7	31.034,3	21,7871
1	11	1.959,3	119,476
2	11	5.414,3	150,91
3	11	47.768,2	68.789,7