PONTIFICIA UNIVERSIDAD
CATÓLICA DEL PERÚ

Escuela de Posgrado

# Optimal Vicinity 2D Median filter for fixed-point or floating-point values

Tesis para obtener el grado académico de Maestro en Procesamiento de Señales e Imágenes Digitales que presenta:

Javier Chang Fu

**Asesor:**

Ph.D. Cesar Carranza de la Cruz

Lima, 2024

## Informe de Similitud

Yo, Cesar Alberto Carranza De La Cruz, docente de la Escuela de Posgrado de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada “Optimal Vicinity 2D Median filter for fixed-point or floating-point values”, de el autor Javier Chang Fu, dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 11%. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 5/06/2024.
- He revisado con detalle dicho reporte y la Tesis o Trabajo de investigación, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Lugar y fecha:

Lima, 6 de Junio de 2024.

| | |
|---|---|
| Apellidos y nombres del asesor / de la asesora: Carranza De La Cruz, Cesar Alberto | |
| DNI: 09641576 | Firma [illegible] |
| ORCID: 0000-0003-1222-0118 | |

## Resumen

Los filtros medianos son una técnica digital no lineal normalmente usada para remover ruido blanco, 'sal y pimienta' de imágenes digitales. Consiste en reemplazar el valor de cada pixel por la mediana de los valores circundantes.

Las implementaciones en punto flotante usan ordenamientos con técnicas de comparación para encontrar la mediana. Un método trivial de ordenar $n$ elementos tiene una complejidad de $O(n^2)$, y los ordenamientos más rápidos tienen complejidad de $O(n \log n)$ al calcular la mediana de $n$ elementos. Sin embargo, éstos algoritmos suelen tener fuerte divergencia en su ejecución.

Otras implementaciones usan algoritmos basados en histogramas, y obtienen sus mejores desempeños cuando operan con filtros de ventanas grandes. Estos algoritmos pueden alcanzar tiempo constante al evaluar filtros medianos, es decir, presenta una complejidad de $O(1)$.

El presente trabajo propone un algoritmo de filtro mediano rápido y altamente paralelizable. Se basa en ordenamientos sin divergencia con ejecución $O(n \log^2 n)$ y mezclas $O(n)$ con los cuales se puede calcular grupos de pixeles en paralelo. Este método se beneficia de la redundancia de valores en pixeles próximos y encuentra la vecindad de procesamiento óptima que minimiza el número de operaciones promedio por pixel. El presente trabajo (i) puede procesar indiferentemente imágenes en punto fijo o flotante, (ii) aprovecha al máximo el paralelismo de múltiples arquitecturas, (iii) ha sido implementado en CPU y GPU, (iv) se logra una aceleración respecto al estado del arte.

***Palabras clave***— Filtro mediano, Mezcla de medianas, Procesamiento en paralelo, GPU, CUDA, Procesamiento de imágenes

## Abstract

Median filter is a non-linear digital technique often used to remove additive white, salt and pepper noise from images. It replaces each pixel value by the median of the surrounding pixels.

Floating point implementations use sorting and comparing techniques to find median. A common method for sorting $n$ elements has complexity $O(n^2)$, and the fastest sorting ones have complexity $O(n \log n)$ when computing the median of $n$ elements. However, such fastest algorithms have strong divergence in their execution.

Other implementations use histogram based algorithms and have their best performance for large size windows. These histogram based achieve constant time median filtering, exhibiting $O(1)$ complexity.

A fast and highly parallelizable median filter algorithm is proposed. It is based on sorting without divergence execution $O(n \log^2 n)$ and merge $O(n)$ that computes groups of pixels in parallel. The method benefits from redundancy values in neighboring pixels and finds the optimal vicinity that minimize the average operations per pixel. The present work (i) can process either fixed or floating point images, (ii) take full advantage of parallelism of multiple architectures, (iii) have been implemented on CPU and GPU, (iv) the results speed up state of the art implementations.

***Keywords***— Median filter, Median merge, Parallel Processing, GPU, CUDA, Image Processing

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Statement of the problem

Median filter is a basic operation used for noise reduction. Two dimensional median filter is used in digital imaging processing where each pixel value is substitute with the median of the neighboring pixels. This process typically uses a square odd window, sort the pixels values and replace the center pixel with the median of the sorted list. Sorting is a non-linear technique and the computational complexity of a naive algorithm like bubble sort is $O(n^2)$.

Digital images values are susceptible to corruption through various stages such as acquisition, transmission, processing, storage or reproduction. For example, image sensors may exhibit damaged pixels, transmission media might introduce spikes in the data and storage bits could become corrupted. An especially disruptive type of noise is impulsive noise, also known as salt and pepper noise, which saturate pixel values to either the maximum or the minimum value. Median filter is specially useful for this kind of heavy tailed distribution noise. It is capable of noise removal while preserving edges of an image.

Median filtering is used in a wide variety of fields such as medical imaging, where it enhances the quality of MRI images, documents processing by reducing noise in scanned documents to improve OCR algorithms performance, and robotics, by filtering noise in distance sensors and inertial measurement units.

Median filter used in digital imaging processing has been extensively studied and a wide variety of efficient algorithms have been developed. The appearance of new technologies prompts the creation and driven forth the development of new methods on median filter calculation, most of them for parallel architectures.

The present work evaluates new methods to improve and optimize median filter using actual technologies.

# Chapter 2

# State of the Art

There have been several works on median filter with different approaches like histogram based algorithms for fixed point or parallelizable schemes using GPUs.

The Branch-less Vectorized Median Filter (BVMF) [1] avoids divergence produced by code branching in parallel computing schemes like CUDA, speeding up previous implementations although it is $O(n^2)$, but the performance is significantly reduced as the size of the window increases.

For large windows there are techniques based on histograms such as Constant Time Median Filter (CTMF) [2] which finds the median in $O(1)$ but have the disadvantage of requiring large amounts of memory for the histograms and not being very suitable for parallel architectures.

Parallel CCDF-based Median Filter (PCMF) [3] a Complementary Cumulative Distribution Function (CCDF) [4] based sorting algorithm propose a highly parallelable structure with a trade-off between $O(1)$ and $O(n)$ complexity order in order to improve throughput.

These histogram-based median filtering algorithms have their best performance in mid-size windows and up. Some implementations such as [5] can be found in libraries like OpenCV, but for small windows sorted and comparison based algorithms outperform histogram methods.

Parallel Register-only Median filter (PRMF) [6] is a fine-tuned high-speed implementation that uses forgetful selection algorithm and exploit GPUs capabilities processing two pixels per thread.

Neighboring redundancy is expanded to 4 pixels in ForgetSel4pix [7] and works with fixed point or floating point data, unlike histogram based methods.

This work proposes and evaluates a novel 2D median filter algorithm that exploits the redundancy data of neighboring pixels and is suitable for parallelization. The main contributions are:

(i) Minimum number of comparison per pixel with an optimal neighborhood selection.

(ii) Uses a novel fast median merge algorithm which merges two sorted lists.

(iii) Highly parallelizable scheme with function, instruction and core acceleration.

(iv) Suitable for either fixed or floating point values which are used in medical imaging applications [8].

(v) Highly optimized implementations on CPU and GPU.

# Chapter 3

# Objectives

## 3.1. Main objective

Propose a new parallel and optimal 2D median filter algorithm.

## 3.2. Specific objectives

(i) An algorithm highly parallelizable that maximizes the utilization of parallel processing across diverse CPU and GPU architectures.

(ii) Highly optimized implementations on CPU and GPU.

(iii) Suitable for either fixed or floating point values.

(iv) Evaluate the new algorithm performance in terms of computational complexity and running time. Compare it with other solutions.

# Chapter 4

# Methodology

## 4.1. Notation and definitions

Symbols notation:
(i) vectors: starts with one capital letter in bold.
(ii) matrix: starts with two capital letters in bold.
(iii) set: Greek letters in bold.

A general median filter runs through each pixel of the image and replace it with the median of the neighboring pixels, this neighborhood is called the **kernel**. The median filter have its domain in the input image and its range in the filtered output image. A general median filter kernel $\boldsymbol{\kappa_i}$ with arbitrary shape generates exactly one output pixel. The **vicinity** are the set of adjacent output pixels in the range. The set of common input pixels of the vicinity are called **common pixels** $\boldsymbol{\zeta}$. **Fast median merge** (FMM) is the algorithm to obtain the median of two sorted lists.

## 4.2. Sorting Network

A sorting network of $n$ elements is an oblivious fixed sequence of comparison-exchange operations (comparators) that sorts any $n$ elements sequence. Optimization of these sorting networks points towards minimizing the length and the depth of the sequence, where the length of the sequence is the number of comparators needed, and the depth is the number of steps (parallel comparators) required to complete the sort. Sorting networks optimizations is a very hard challenging open problem and it have been shown that sorting network verification problem remains $CoNP$ complete [9].

Even though Ajtai, Komlós and Szemerédi (AKS) [10] network has asymptotically length of $O(n \log n)$ large constant hidden in the big O bound make it not practical. In practice, algorithms like Batcher's Odd-even Merging Sort are often used despite its non-optimality $O(n \log^2 n)$ complexity.

Batcher's odd-even Merging Sort [11] is a recursive algorithm that builds larger networks merging two smaller sorted list, which have been Batcher's odd-even sorted. While originally designed only for sorting power of 2 elements, variations of this method can sort list of arbitrary length.

Sorting two elements networks requires just one comparator. Merging two lists of one element each requires also one comparator. Let $N(n)$ denote the number of comparators for sorting $n$ elements and $M(n)$ denote the number of comparators for merging two $n/2$ sorted lists. Batcher's odd-even merging sort is described by the following recursive equations:

$$N(n) = 2 \cdot N(n/2) + M(n) \tag{4.1}$$
$$M(n) = 2 \cdot M(n/2) + n/2 - 1 \tag{4.2}$$
$$N(2) = M(2) = 1 \tag{4.3}$$

## 4.3. Parallel Architectures

Modern CPU architectures uses Single Instruction Multiple Data (SIMD) that group array of data in a single register, and can be simultaneously operated with a single instruction. Actual SIMD architectures can operate from array of bits to array of double precision floats (64-bits). The instructions used in this work were: gather, compare, max, min and several logic ones. Gather instructions were used to collect data, max and min instructions were used in sorting arrays, compare and logic instructions were used in fast median merge. In addition, actual CPUs have a multicore architecture that allows the use of Multiple Instruction Multiple Data (MIMD) scheme that can run different threads in parallel.

Gaming leading technology push the development of graphics processing units (GPU). Modern GPUs are not used only for graphics, but for scientific purposes. Actual GPUs are very complex units with a manycore architecture, that have thousands of cores coordinated by hardware schedulers and organized in streaming multiprocessors. GPU also include memory organized in different levels of cache, with complex sharing and access techniques. All this hardware is finely synchronized and tunned so user do not have to deal with all the details involved. However, knowing the intrinsics of the architecture allows to optimize the implementation of the algorithm. This Single Instruction Multiple Threads (SIMT) architecture is a combination of SIMD, MIMD and multithreading [12].

## 4.4. Proposed Vicinity 2D Median Filter algorithm

The proposed algorithm computes a set of medians defined by a vicinity $\boldsymbol{\nu} = \{s_0, s_1, \ldots, s_{n-1}\}$. This vicinity have its domain in a set of kernels $\boldsymbol{\kappa} = \{\boldsymbol{\kappa_0}, \boldsymbol{\kappa_1}, \ldots, \boldsymbol{\kappa_{n-1}}\}$ such as $\boldsymbol{\zeta} = \boldsymbol{\kappa_0} \cap \boldsymbol{\kappa_1} \cap \cdots \cap \boldsymbol{\kappa_{n-1}}$ is a non empty set. For each kernel $\boldsymbol{\kappa_i}$ there is a two subset partition $\boldsymbol{\kappa_i} = \boldsymbol{\{\zeta, \alpha_i\}}$. The common set $\boldsymbol{\zeta}$ is sorted once for the vicinity. Then each $\boldsymbol{\alpha_i}$ is sorted and it is fast median merged with sorted $\boldsymbol{\zeta}$ where the median $s_i$ of $\boldsymbol{\kappa_i}$ is extracted.

For the algorithm implementation the following considerations without loss of generality are set:

- Fixed square kernels $k \times k$. $k \in \mathbb{N}$, $k$ *odd*, $k \geq 3$
- Fixed square vicinity $s \times s$. $s \in \mathbb{N}$, $1 \leq s \leq k$
- Use of Batcher's odd-even sorting algorithm.

The algorithm is shown in figure 4.1. It processes a matrix Functional Region of interest $\boldsymbol{FR}(i,j)$ of $(k+s-1) \times (k+s-1)$ generated by the set $(\boldsymbol{\kappa_0} \cup \boldsymbol{\kappa_1} \cup \cdots \cup \boldsymbol{\kappa_{s^2-1}})$ and get $s \times s$ output pixels in a vicinity $\boldsymbol{VI}(k,l)$. The process begin by sorting once the common input pixels (line 3). Then, the additional input pixels (line 7) that complete the kernel $k \times k$, are sorted. Finally, FMM is applied to common part with additional part to find the output median pixel $\boldsymbol{VI}(x,y)$ (line 8).

```
 1: function OV2DMF ($k$, $s$, $\boldsymbol{FR}$)
    ▷ Optimal Vicinity 2D Median Filter of $s \times s$ output pixels from $\boldsymbol{FR}$ input pixels.
    ▷ Inputs:
    ▷       $k$: Kernel size $k \times k$ for median filter.
    ▷       $s$: Vicinity output size $s \times s$ for OV2DMF.
    ▷       $\boldsymbol{FR}(i,j)$: Functional Region of interest.
    ▷             $0 \leq i,j \leq (k+s-2)$
    ▷ Output:
    ▷       $\boldsymbol{VI}(k,l)$: Median filtered $s \times s$ output pixels.
    ▷             $0 \leq k,l \leq (s-1)$
 2:    $\boldsymbol{\zeta} = \{\boldsymbol{FR}(i,j) \setminus (s-1) \leq i,j \leq (k-1)\}$
 3:    $\boldsymbol{C}(m) \leftarrow Sort(\boldsymbol{\zeta}), 0 \leq m < (k-s+1)^2$
 4:    for $x \leftarrow 0$ to $s-1$ do
 5:       for $y \leftarrow 0$ to $s-1$ do
 6:          $\boldsymbol{\kappa} = \{\boldsymbol{FR}(i,j) \setminus x \leq i < (x+k),$
                        $y \leq j < (y+k)\}$
 7:          $\boldsymbol{\alpha} = \boldsymbol{\kappa} - \boldsymbol{\zeta}$
 8:          $\boldsymbol{A}(n) \leftarrow Sort(\boldsymbol{\alpha}), 0 \leq n < (s-1)(2k-s+1)$
 9:          $\boldsymbol{VI}(x,y) \leftarrow FMM(\boldsymbol{C}(m), \boldsymbol{A}(n))$
10:       end for
11:    end for
12: end function
```

Figure 4.1: Vicinity 2D Median Filter algorithm.

For the FMM algorithm, given $k$ odd, each kernel $\boldsymbol{\kappa_i}$ has odd cardinality. In the partition of a kernel one subset has odd cardinality and the other one has even cardinality. The proposed FMM uses a three element comparison starting with the median of the odd array and two middle elements of even array. This first comparison sets the displacement direction right/left or left/right of the odd/even arrays. After each one-position

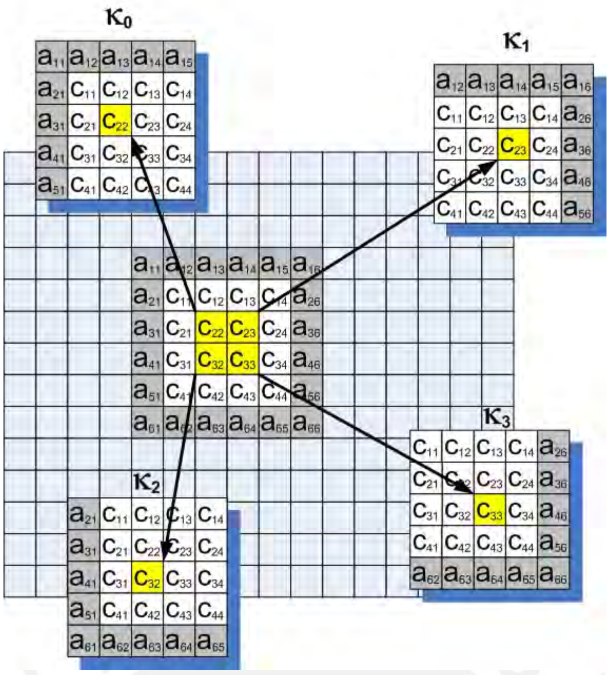


Figure 4.2: Kernels in Region of interest for Function acceleration. $\{\boldsymbol{\kappa_0}, \boldsymbol{\kappa_1}, \boldsymbol{\kappa_2}, \boldsymbol{\kappa_3}\}$ are the set of $5 \times 5$ kernels in a $2 \times 2$ vicinity shown in yellow background. These kernels have $c_{ij}$ as common pixels.

displacement a new three element comparison is done until the median is found. In the best scenario the median is found in the first three element comparison where only 2 comparisons are needed, and the median is the odd array median. In worst case scenario the displacement will reach the end of the shortest array to find the median and the number of comparison will be $2 + (min(|\boldsymbol{\zeta}|, |\boldsymbol{\alpha_i}|) + 1)/2$. In highly correlated images the median is expected to be found in the very first comparisons.

For example, let's set $k = 5$ and a vicinity $s = 2$ as depicted in figure 4.2. The vicinity in yellow gives the set of kernels $\boldsymbol{\kappa} = \{\boldsymbol{\kappa_0}, \boldsymbol{\kappa_1}, \boldsymbol{\kappa_2}, \boldsymbol{\kappa_3}\}$ that share 16 common pixels $\boldsymbol{\zeta} = \{c_{11}, c_{12}, c_{13}, c_{14}, c_{21}, c_{22}, c_{23}, c_{24}, c_{31}, \ldots, c_{43}, c_{44}\}$.

First, $\boldsymbol{\zeta}$ is sorted once. Then, elements of $\boldsymbol{\alpha_0} = \{a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{21}, a_{31}, a_{41}, a_{51}\}$ are sorted. And next, this two sorted arrays of $\boldsymbol{\kappa_0} = \{\boldsymbol{\zeta}, \boldsymbol{\alpha_0}\}$ are fast median merged to get the median as depicted in figure 4.3. This procedure is repeated for the rest of kernels $\{\boldsymbol{\kappa_1}, \boldsymbol{\kappa_2}, \boldsymbol{\kappa_3}\}$ to get the four medians of the output pixels in the vicinity.

Extending the example of the FMM procedure depicted in figure 4.3: (i) In comparisons 1 and 2 the median $a_{11}$ of the odd array is compared with $c_{24}$ and $c_{31}$ that are the two middle elements of the even array. This first two comparisons sets the direction of displacement, otherwise the final median will be $a_{11}$. (ii) With the direction set, another three elements $\{a_{21}, c_{31}, c_{32}\}$ set is selected, comparison 3 ($c_{32} < a_{21}$) indicate if the displacement continues, (iii) otherwise the median will be the maximum between $a_{21}$ and $c_{31}$. (iv) This last two steps (ii) (iii) are repeated until the median is found or the end of

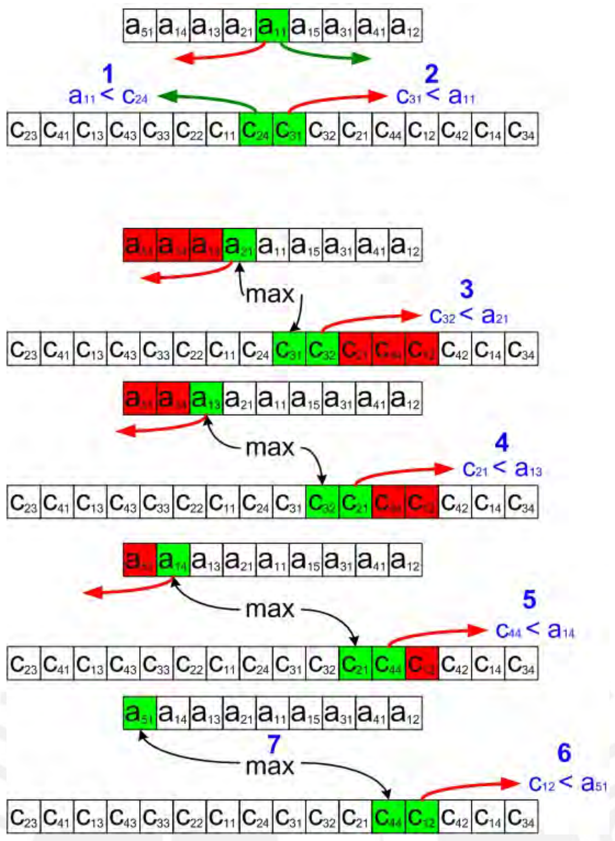


Figure 4.3: Median Merge algorithm sequence example. Common pixels list $\boldsymbol{\zeta}$ and non common pixels $\boldsymbol{\alpha_i}$ are sorted. Comparisons 1,2 set the displacement direction. Comparisons 3,4,5,6 exhaust possible positions on arrays. In worst case scenario, median is found in comparison 7.

the shortest array is reach.

In this example, if the last displacement comparison 6 is true, the median will be $c_{12}$. In worst case scenario this last displacement comparison is false and one last comparison 7 is needed to get de median that will be the maximum between $a_{51}$ and $c_{44}$.

The described FMM algorithm can be presented in different forms, such as: (i) generic size non-recursive function, (ii) recursive function that resemble a binary search, and (iii) loop unroll form. In this work loop unroll form is chosen because it provides the fastest implementation.

FMM algorithm was implemented as a generic size non-recursive function. Then it was compare with its recursive form witch resemble a binary search method and could be used in large kernels. Finally, figure 4.4 shows a loop unroll form of the FMM algorithm that was the fastest implementation for the kernel sizes that were tested in this work.

```
function FastMedianMerge ($\boldsymbol{A}$, $\boldsymbol{B}$, $m$, $n$)
    ▷ Finds the median of two sorted arrays.
    ▷ Inputs:
    ▷       $\boldsymbol{A}(i)$:  Sorted odd array.  $0 \leq i < m$
    ▷       $\boldsymbol{B}(j)$:  Sorted even array.  $0 \leq i < n$
    ▷ Output:
    ▷       med: Median($\boldsymbol{A}$,$\boldsymbol{B}$).
    $k \leftarrow min(m, n)/2 - 1$
    $i \leftarrow (m - 1)/2$
    $j \leftarrow (n/2) - 1$
    if $\boldsymbol{A}(i) < \boldsymbol{B}(j)$ then
        if $\boldsymbol{A}(i+1) < \boldsymbol{B}(j-1)$ then
            ...
            if $\boldsymbol{A}(i+k) < \boldsymbol{B}(j-k)$ then
                $med \leftarrow \boldsymbol{B}(j-k)$
            else
                $med \leftarrow min(\boldsymbol{A}(i+k+1), \boldsymbol{B}(j-k))$
            end if
        else
            $med \leftarrow min(\boldsymbol{A}(i+1), \boldsymbol{B}(j-0))$
        end if
    else if $\boldsymbol{A}(i) > \boldsymbol{B}(j+1)$ then
        if $\boldsymbol{A}(i-1) > \boldsymbol{B}(j+2)$ then
            ...
            if $\boldsymbol{A}(1-k) > \boldsymbol{B}(j+K)$ then
                $med \leftarrow \boldsymbol{B}(j+k)$
            else
                $med \leftarrow max(\boldsymbol{A}(1), \boldsymbol{B}(j+k))$
            end if
        else
            $med \leftarrow max(\boldsymbol{A}(i-1), \boldsymbol{B}(j+1))$
        end if
    else
        $med \leftarrow \boldsymbol{A}(i)$
    end if
end function
```

Figure 4.4: Speed optimized algorithm to find median of two sorted arrays.

## 4.5. Optimal Vicinity computation

To compute the optimal vicinity, first the number of operations $L(k, s)$ for a given kernel size $k$ and vicinity size $s$ need to be derived. Next, for each $k$, there is an $\hat{s}$ that results in minimum number of comparisons, that optimal vicinity size $\hat{s} = S(k)$ need to be found. Finally, the speed up $S_p(k)$ is calculated with the ratio of $L(k, s)$ between naive case (vicinity of $s$=1) and the optimal vicinity $L(k, \hat{s})$.

Given kernel $k \times k$ and a vicinity $s \times s$ , the number of elements of each partition of $\boldsymbol{\kappa_i} = \{\boldsymbol{\zeta}, \boldsymbol{\alpha_i}\}$ is:

$$|\boldsymbol{\zeta}| = (k - s + 1)^2 \tag{4.4}$$

$$|\boldsymbol{\alpha_i}| = (s - 1) \cdot (2k - s + 1) \tag{4.5}$$

The dominant operation in the present method is comparison. The implemented sorting network is Batcher odd-even mergesort that is a recursive algorithm where $N(n)$ is the number of comparison for sorting $n$ elements and $M(n)$ is the number of comparisons for merging two $n/2$ as depicted in equations 4.1, 4.2 and 4.3. Solving these recursive equations, $N(n)$ is expressed as an explicit function of $n$:

$$N(n) = n - 1 + \frac{n}{4} \cdot lg(n) \cdot (lg(n) - 1) \tag{4.6}$$

The proposed FMM require a maximum of $G(m, n)$ comparisons to find the median of two sorted arrays of $m$ and $n$ elements as depicted in the following equation:

$$G(m, n) = \frac{min(m, n) + 1}{2} + 2 \tag{4.7}$$

The proposed median filter requires a maximum of $L(k, s)$ comparisons per pixel. This value represents the sum of three steps: (1) average comparisons for the common elements $\boldsymbol{\zeta}$ in the vicinity $s \times s$, (2) comparisons for the non-common elements, and (3) FMM maximum comparisons to determine the median:

$$L(k, s) = \frac{N(|\boldsymbol{\zeta}|)}{s^2} + N(|\boldsymbol{\alpha_i}|) + G(|\boldsymbol{\zeta}|, |\boldsymbol{\alpha_i}|) \tag{4.8}$$

The optimal vicinity $S(k)$ is the one with the minimum number of operations per pixel:

$$S(k) = \underset{1 \leq s < k, s \in \mathbb{N}}{\arg\min}\ L(k, s) \tag{4.9}$$

Figure 4.5 shows the surface $L(k, s)$. Quasi-newton algorithm was used to find the optimal $s$ for each $k$ as shown in magenta dots over the surface.

Figure 4.6 shows slices of the surface $L(k, s)$ for different values of $k$. In these overlay plots the highlighted points indicates the optimal $s$ with the minimum number of comparisons for a given $k$.

The number of comparisons is inversely proportional to the processing speed of the pixels, so this optimal vicinity speed up the median filter. Setting batcher's odd-even sorting network of 1 pixel vicinity ($s = 1$) as reference, the theoretical speed up $S_p(k)$ can be computed as:

$$S_p(k) = \frac{N(k * k)}{L(k, S(k))} \tag{4.10}$$

Figure 4.7 shows the plot of this theoretical speed up.

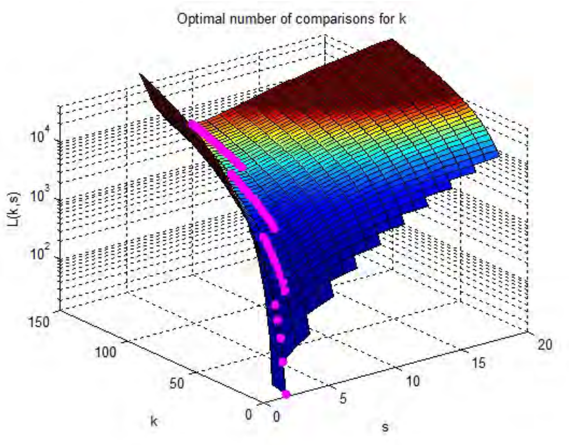


Figure 4.5: Optimal vicinity. Minimum number of comparisons $L(k, s)$ per pixel of proposed median filter, where $k$ is the size of the kernel and $s$ is the size of the vicinity. Magenta dots show the optimal $s$ vicinity for each $k$ on the $L(k, s)$ surface.

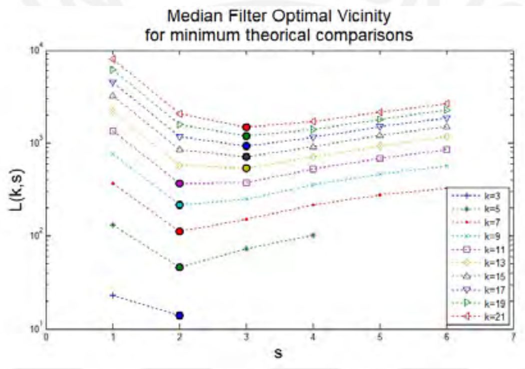


Figure 4.6: Optimal vicinity $s$. Minimum number of comparisons $L(k, s)$ per pixel of proposed median filter, where $k$ is the size of the kernel and $s$ is the size of the vicinity. Highlight dots show the optimal $s$ vicinity given fixed $k$ value.

## 4.6. Parallel architectures

The implementation of the proposed method uses multiple layers of parallelism that can be combined based on the function to be accelerated and the available hardware architecture. In the present work four acceleration layers are defined: (i) **function acceleration**, (ii) **instruction acceleration**, (iii) **multicore acceleration** and (iv) **manycore acceleration**. Function acceleration depends on the target function to accelerate, median filter for this work. Instruction acceleration uses SIMD to manipulate data arrays in parallel. Multicore acceleration assigns a partition of the input image to each core. Manycore acceleration process threads in parallel.

In this work the target function is Median Filter and the Region of Interest ***RI*** for a median filter is a square of $k \times k$ pixels that will generate 1 output pixel $O$ as shown in figure 4.8(a).

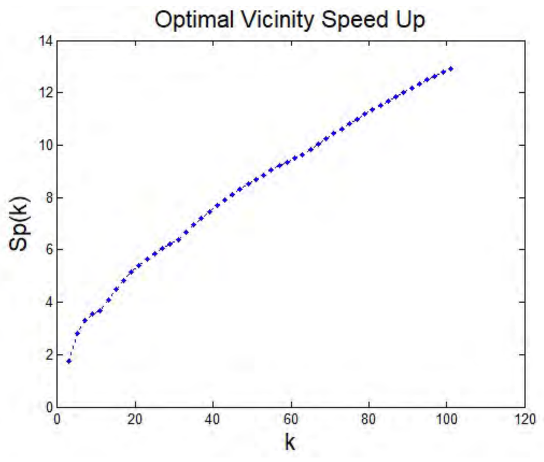


Figure 4.7: Theoretical proposed median filter speed Up $S_p(k)$ with optimal vicinity in relation to vicinity of 1 element

### Function acceleration

Parallel Layer 0 uses **function acceleration** that benefits of redundancy to reassemble a target function. In this work Median Filter is the target function and OV2DMF is the reassemble algorithm. The redundancy that is used are the common pixels of a vicinity, which are sorted once. This results in an acceleration speed up of $S_p(k)$ with respect to single pixel vicinity as shown in equation 4.10. Figure 4.8(b) shows $\boldsymbol{FR}$ that denotes the Functional Region of interest of $(k + s - 1) \times (k + s - 1)$ in the case of OV2DMF and the output pixels processed in the vicinity $\boldsymbol{VI}$ of $s \times s$ corresponding to $s^2$ output pixels.

### Instruction acceleration

Parallel Layer 1 uses **instruction acceleration** that can manipulate multiple data simultaneously. **SIMD** is used for this layer, specifically AVX2 that was supported on the test machine. This parallel layer can process $\boldsymbol{r}$ regions $\boldsymbol{FR}$ simultaneously to generate a vicinity vector $\boldsymbol{Y}$, that is, the array of $\boldsymbol{r}$ contiguous output vicinities $\{\boldsymbol{V_0}, \boldsymbol{V_1}, \ldots, \boldsymbol{V_{r-1}}\}$ corresponding to $r \times s^2$ output pixels. $\boldsymbol{IR}$ denotes de Instruction Region of interest of $(k+s-1) \times (k+s \times r-1)$ input pixels, corresponding to $\boldsymbol{r}$ overlapped $\boldsymbol{FR}$ with horizontal stride of $s$, as seen in figure 4.8(c).

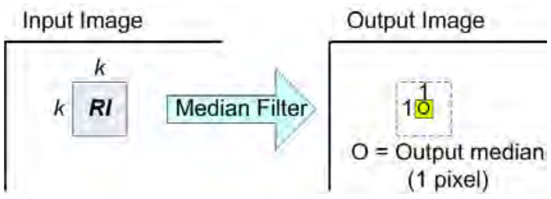
### Multicore acceleration

Parallel Layer 2 uses **multicore acceleration** that benefits of multicore architectures that can process $\boldsymbol{p}$ partitions of the output image in parallel. Given a multicore architecture with $\boldsymbol{p}$ cores, the input image is divided in $\boldsymbol{p}$ blocks, one block per core. With this acceleration layer, $\boldsymbol{p}$ vicinity vectors $\boldsymbol{Y}$ can be processed simultaneously corresponding to $p \times r \times s^2$ output pixels, as seen in figure 4.8(d).

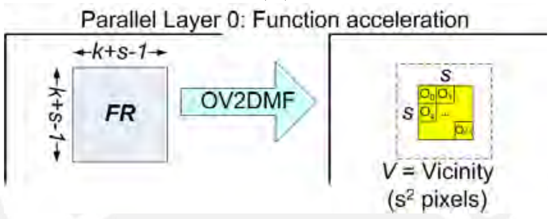
### Manycore acceleration

Parallel Layer 3 uses **manycore acceleration** that benefits of manycore architectures which can process $\boldsymbol{q}$ threads in parallel. This architecture can process $\boldsymbol{q}$ contiguous output vicinities $\boldsymbol{V}$ simultaneously, corresponding to $q \times s^2$ output pixels. The input region of interest is $(k + s - 1) \times (k + s \times q - 1)$ corresponding to $\boldsymbol{q}$ overlapped $\boldsymbol{FR}$ with stride of $s$, as seen in figure 4.8(e).

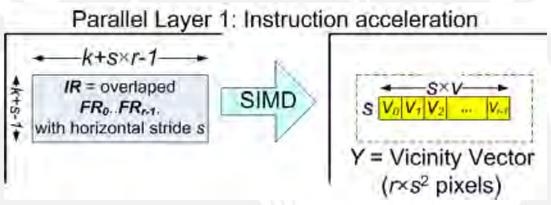
In the CPU implementation layers 0, 1 and 2 are combined. In the GPU case layers 0 and 3 are used. The processing in layer 3 on the GPU is equivalent to layer 1 and 2 combined with MIMD [12]



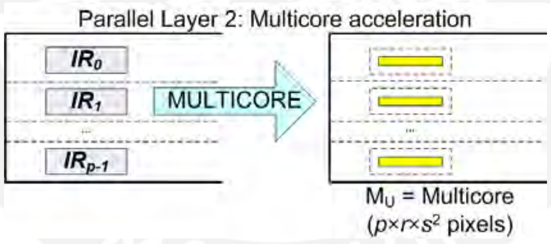
(a)



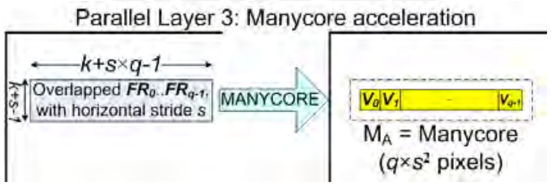
(b)



(c)



(d)



(e)

Figure 4.8: (a),(b),(c),(d),(e)

# Chapter 5

# Results

The presented implementation was tested using a standard desktop, Intel core i5-6500 @ 3.2 GHz with 8 GB of RAM running under 64-bit Windows 10 Pro OS, with a graphics card NVIDIA GeForce GTX1050, compute capability 6.1, 640 cores, 1354 MHz base clock, 2 GB of GDDR5 RAM, connected to the host through PCI express x16 Gen3, using CUDA toolkit version 9.0, Microsoft Visual Studio Community 2014 version 14. An image of size 2560 x 2560 8-bit grayscale was used, but was processed as a float image instead of a fixed point image.

Reference [6] uses a Tesla C2070 GPU (compute capability 2.0, 448 cores, 1, 1.15 GHz of GPU clock speed) card hosted by a system with one Xeon E5620 2.40GHz processor running a linux kernel 2.6.18 x86\64 and CUDA toolkit v4.0. Test image was a 2048 x 2048 8-bit grayscale image.

For CPU implementation explicit loop unroll optimization technique was used, but speed results where not as fast as other implementations. Then in addition SIMD AVX2 instructions were used. These instructions operate on 256-bit registers, allowing simultaneous processing of eight 32-bits floating points numbers. The SIMD instructions used in the implementation were gather, compare, min/max and logical ones. Proposed Fast median merge uses conditionals in its algorithm, this approach had to be changed to a deterministic form to take advantage of SIMD instructions. Gathering was implemented with available SIMD, but storing back filtered pixels was done without SIMD. Finally, the tests were done in two flavors, with and without parallel threads. Figure 5.1 shows single threads results, highlighted points indicate the optimal vicinity $s$. When kernel size increases, execution exception is generated due to SIMD resources limits. When parallel threads were used, each CPU core process one block of consecutive data in memory. In this work, four cores were used. Figure 5.2 shows results using 4 threads. The results with pthreads on CPU for $k = 3$ shows that memory bandwidth was reached. This was tested with a dummy filter using only addition instead of median filter, revealing a transfer instruction limit of 1027 Mpix/s.

For GPU implementation explicit loop unroll for sorting common and additional data was used. Better results were achieved by first reading all region of interest to registers, then move processing data to working buffer. Cuda C compiler took a lot of time in

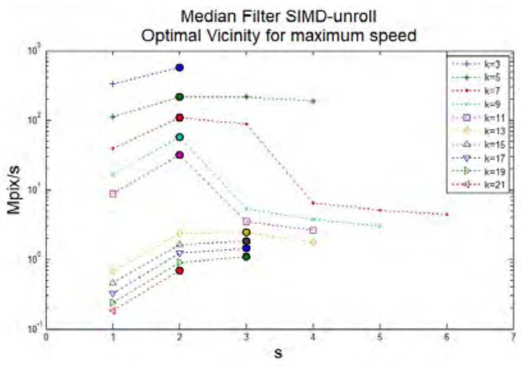


Figure 5.1: Proposed median filter results on CPU. Implementation using SIMD and unroll techniques. Highlight dots mark optimal $s$ for maximum speed.

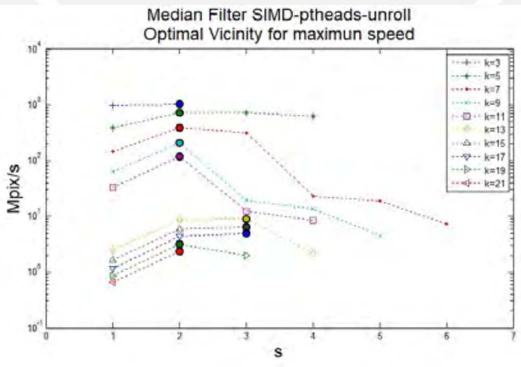


Figure 5.2: Proposed median filter results on CPU. Implementation using SIMD, unroll and threads techniques. Highlight dots mark optimal $s$ for maximum speed.

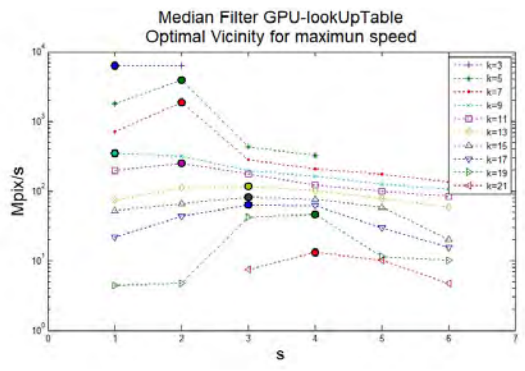


Figure 5.3: Proposed median filter results on GPU. Implementation using look-up table techniques. Highlight dots mark optimal $s$ for maximum speed.

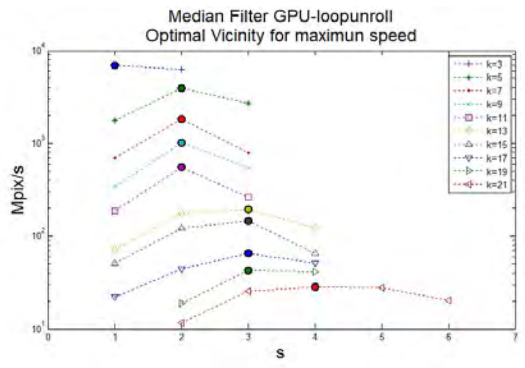


Figure 5.4: Proposed median filter results on GPU. Implementation using unroll techniques. Highlight dots mark optimal $s$ for maximum speed.

the optimization of large kernels, due explicit loop unroll optimization. Compile took up to 22 hours for k=21 s=6. Figure 5.4 show GPU results using loop unroll technique, highlight points indicate optimal vicinity $s$. Data was plot until the point where the peak performance points were detected. At this point a new approach with lookup tables for gathering common and additional data was used. Compile time when down to 5 minutes instead of 22 hours, but speed performance decreased. Figure 5.3 show GPU results using lookup tables.

Source codes for CPU and GPU implementation could be handwritten for small kernels. For example for $k = 3$ with optimal vicinity the kernel source code for CPU have 164 lines. However, due to loop unroll technique, as $k$ increases so does the source code lines in a $O(k^2 \log^2 k)$. For example with $k = 21$ with optimal vicinity $s = 3$ the CPU source code have 9200 lines. In such long source codes with lots of index due to Batcher's sorting network and Fast Median Merge algorithm, a numerical mistake is very probable. To solve this problem, a source code generator program for CPU and GPU was implemented. This program writes all possible source codes that need to be compiled after being generated. For this work, almost six million source codes lines where generated for exploration but once the optimal vicinity $s$ has been checked and the hardware limitation have been found; this source code lines were reduced dramatically.

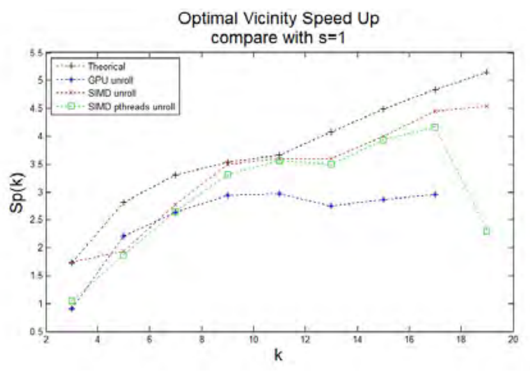


Figure 5.5: Proposed Median filter speed-up results in comparison with theoretical plot.

Figure 5.5 show speed up comparisons between theoretical plot, CPU and GPU results. Theoretical plot only consider comparison as relevant instructions. In GPU lookup table results, control instructions, indexed table access, and conditionals are not irrelevant with respect to comparison instructions, so speed up results does not follow theoretical plot. Other results does follow theoretical plot.

Table 5.1 shows speed comparisons in Mpixels/second of the proposed algorithm for CPU and GPU, with respect to the best know published and commercial algorithms up to date.

For GPU, the floating point results are the fastest than any other implementation up to date. Also, the closest competition ForgetSel4pix [7] only have implementation up to $k = 7$ and Mathlab R2015a up to $k = 11$.

Only in the case of $k = 3$ for GPU implementation, a special Functional Region of interest was used. Such a small kernel does not have comparison as the predominant instruction. Memory access, control and conditional instruction are not negligible. In addition, cuda compiler took a lot of time with internal optimization techniques that are hidden from the programmer. The optimal vicinity for this case was a $10 \times 1$ instead of the theoretical $2 \times 2$.

Comparing the GPU 32-bit floating point or 32-bit fixed point results with the best GPU 8-bit fixed point results, this work is still faster than any other implementation. PRMF [6] have implementation up to $k = 7$ and Matlab 2015a up to $k = 13$.

For CPU, the floating point results are faster that any other implementation. In the CPU tests with $k = 3$ memory bandwidth limit was reached. OpenCV only have implementation up to $k = 7$. Matlab implementation is only optimized for $k = 3$ and drops dramatically thereafter.

Comparing the preset work CPU 32-bit floating results with the best CPU 8-bit fixed point, both implementations have reached memory bandwidth limit for $k = 3$. For $k = 5$ OpenCV have twice the performance in Mpix/s, but with a data width that is four times smaller. For $k = 7$ up to $k = 11$ the present implementations have better performance than OpenCV and Matlab, despite the comparison are not base on the same data widths. For $k >= 13$ the performance drops dramatically due to exhausted SIMD CPU resources.

| | Mpix/s | | | | | | | | | | Type | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GPU Method | $k = 3$ | $k = 5$ | $k = 7$ | $k = 9$ | $k = 11$ | $k = 13$ | $k = 15$ | $k = 17$ | $k = 19$ | $k = 21$ | bpp | Fix/Flo |
| **OV2DMF** | 11552 | 3904 | 1841 | 1007 | 559 | 195 | 146 | 65.4 | 42.4 | 28.2 | 32 | Fix/Flo |
| ForgetSel4pix [7] | 6994 | 3036 | 1046 | | | | | | | | 32/16/8 | Fix/Flo |
| Matlab R2015a | 1180 | 176 | 39 | 10 | 3.6 | | | | | | 32 | Flo |
| PRMF [6] | 5309 | 1422 | 440 | | | | | | | | 8 | Fix |
| Matlab R2015a | 1160 | 160 | 42 | 16 | 7.2 | 3.7 | | | | | 8 | Fix |
| OpenCV [5] | 44.6 | 40.9 | 42.7 | 43.4 | 43.4 | 42.7 | 43.7 | 43.9 | 44.3 | 43.9 | 8 | Fix |
| CPU Method | | | | | | | | | | | | |
| **OV2DMF** | 1027 | 733 | 383 | 208 | 118 | 8.9 | 6.6 | 4.89 | 3.19 | 2.38 | 32 | Fix/Flo |
| OpenCV [2] | 1081 | 332 | 3.03 | | | | | | | | 32 | Flo |
| Matlab R2015a | 215 | 10.52 | 6.78 | 4.74 | 3.6 | 2.73 | 2.18 | 1.79 | 1.48 | 1.25 | 32 | Flo |
| OpenCV [2] [5] | 4641 | 1438 | 99.8 | 91.4 | 114.4 | 116.6 | 118.7 | 120.7 | 121.4 | 122.3 | 8 | Fix |
| Matlab R2015a | 614 | 475 | 53.66 | 51.05 | 49.38 | 47.25 | 45.89 | 44.13 | 43.74 | 25.81 | 8 | Fix |

Table 5.1: Speed comparison table for CPU and GPU (output pixels/second). Test were done under the same conditions, except PRMF. Nomenclature: (i) BPP means bits per pixel, (ii) FLO means floating point data, (iii) FIX means fixed point data. Blank tabs denotes not supported size for that method. Dashed line separate floating point and fixed point results. Proposed method in bold.

# Conclusions

The work presented has led to the development of a new, fast, scalable, and highly parallelizable algorithm for the computation of a 2D median filter that operates on both fixed and floating-point images. This includes the introduction of a new fast median merge algorithm for merging two sorted lists. The highly parallelizable scheme can achieve up to one thread per vicinity, and highly optimized implementations have been developed for both CPU and GPU.

The results of the implementation on CPU and GPU show faster speeds compared to commercial solutions and previous works to date for practical kernels (small and medium sizes).

# Future work

(i) Optimize implementation for CPU and GPU for 8/16 bits fix point.

(ii) Optimize the algorithm to remove unnecessary comparisons.

(iii) Optimize the merge algorithm through binary search.

(iv) Enhance algorithm efficiency through the utilization of partial orderings retrieved from prior vicinities.

(v) FPGA implementation.

# Bibliography

[1] Wei Chen, Marcel Beister, Yiannis Kyriakou, and Marc Kachelrieß, "High performance median filtering using commodity graphics hardware," *IEEE Nuclear Science Symposium Conference Record*, pp. 4142–4147, 2009.

[2] Simon Perreault and Patrick Hebert, "Median Filtering in Constant Time," *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2389–2394, 2007.

[3] Ricardo M. Sánchez and Paul A. Rodríguez, "Highly parallelable bidimensional median filter for modern parallel programming models," *Journal of Signal Processing Systems*, vol. 71, no. 3, pp. 221–235, 2013.

[4] Paul A. Rodríguez Ricardo M. Sánchez, "Bidimensional Median Filter for Parallel Computing Architectures," *Icassp 2012*, pp. 1549–1552, 2012.

[5] Oded Green, "Efficient scalable median filtering using histogram-based operations," *IEEE Transactions on Image Processing*, vol. 27, no. 5, pp. 2217–2228, 2018.

[6] Gilles Perrot, Stéphane Domas, and Raphaël Couturier, "Fine-tuned high-speed implementation of a GPU-based median filter," *Journal of Signal Processing Systems*, vol. 75, no. 3, pp. 185–190, jun 2014.

[7] Gabriel Salvador, Juan M Chau, Jorge Quesada, and Cesar Carranza, "Efficient GPU-based implementation of the median filter based on a multi-pixel-per-thread framework," pp. 5–8.

[8] Cesar Carranza, Victor Murray, Marios Pattichis, and E. Simon Barriga, "Multiscale AM-FM decompositions with GPU acceleration for diabetic retinopathy screening," *Proceedings of the IEEE Southwest Symposium on Image Analysis and Interpretation*, pp. 121–124, 2012.

[9] Aviad Rubinstein, "On the Computational Complexity of Optimal Simple Mechanisms," 2015.

[10] Miklós Ajtai, János Komlós, and Endre Szemerédi, "An 0 (n log n) sorting network," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing.* ACM, 1983, pp. 1–9.

[11] K. E. Batcher, "Sorting networks and their applications," *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, p. 307, 1968.

[12] John L. Hennessy, David A. Patterson, and Krste. Asanovic, K.., *Computer architecture : a quantitative approach*, Morgan Kaufmann/Elsevier, 2012.