High Performance Median Filtering Algorithm Based on NVIDIA GPU Computing

Placido Salvatore Battiato
University of Catania, Italy
battiatoplacido@gmail.com

***Abstract*—Median filtering technique is often used to remove additive white, salt and pepper noise from a signal or a source image. This filtering method is essential for the processing of digital data representing analog signals in real time. The median filter considers each pixel in the image in turn and looks at its nearby neighbors to determine whether or not it is representative of its surroundings. It replaces the pixel value with the median of neighboring pixel values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. We have used graphics processing units (GPUs) to implement the post-processing, performed by NVIDIA Compute Unified Device Architecture (CUDA). Such a system is faster than the CPU version, or other traditional computing, for processing medical applications such as echography or Doppler. This paper shows the effect of the Median Filtering and a comparison of the performance of the CPU and GPU in terms of response time.**

***Index Terms*—GPU programming, performances, filters, images**

I. Introduction

The noise and distortion are the main factors that limit the capacity of data transmission in telecommunications affecting the accuracy of the results in the signal measurement systems, in communications and signal processing [1], [2], [3]. The noise reduction and distortion removal are major problems in applications such as: cellular mobile communication, speech recognition, image processing, medical signal processing, radar, sonar, and any other application where the desired signals cannot be isolated from noise and distortion. Noise is defined as an unwanted signal that interferes with the communication or measurement of another signal. Noise itself is an information-bearing signal that conveys information regarding the sources of the noise and the environment in which it propagates. Impulse noise is caused by malfunctioning pixels in camera sensors, faulty memory locations in hardware, or transmission in a noisy channel. The noise can be classified by its spectral characteristics, in discrete sense, the white noise signal constitutes a series of samples that are independent and generated from the same probability distribution *Gaussian Noise*, given a Gaussian distribution, it is concentrated around a mean frequency μ and about 70% of noise is in the interval $[\mu-\sigma, \mu+\sigma]$ and about 95% in the interval $[\mu-2*\sigma, \mu+2*\sigma]$ where σ is the standard deviation. The white noise is defined as an uncorrelated random noise process. Random noise has the same power at all frequencies and it would necessarily need to have infinite power, and it is therefore a theoretical concept only. It cannot be removed by a traditional low-pass filter or band-pass filter.

The *salt and pepper noise* or impulsive noise is visible in images and is similar to many black and white spots and is generated during image acquisition. It is characterized by positive or negative peaks that saturate the pixels of the image. An example of salt and pepper noise in images is shown in figure 6b. An example of white noise audio signal is shown in figure 7a. A white noise is also a sound that contains every frequency within the range of human hearing and it can be generated on a sound synthesizer. Sound designers can use this sound, with some processing and filtering, to create a multitude of effects such as wind, surf, space whooshes, and rumbles. The analog voltage signal produced by the phototransistors is converted into a digital or numeric value, the ADC adds broad spectrum noise or white noise. The Median filter is a nonlinear digital filtering technique often used to remove salt and pepper noise.

Computing the neighborhood requires more computation time in a post processing step. More recently, the opened perspectives have been offered by modern GPUs that allow us to develop CUDA-based filters, which will be shown as the fastest median filter implementation known. The surgical applications such as echography or Doppler to biomedical research and clinical medicine for *real-time* image request image brightness and exceptional image quality. In this paper, the first section explains the filtering problem and mathematical model. In CUDA section, we have presented the algorithm implemented using CUDA of median filter comparing the performance of CPU implementations of filtering effects, analysing one dimensional (1D) audio signal.

II. Analog to Digital conversion

In *Signal Theory, Sampling* is a technique that allows the conversion of a continuous signal (for example audio) to a discrete-time signal evaluating the amplitude of that signal at regular time interval (sample time, t_s) [4], [5] and translate it to a binary word. Each of these amplitude is converted to the respective binary code through an ADC (Analog to digital converter). The ADC has a voltage reference that is used to understand the right binary value of each sample. An example is shown in figure 1 where a continuous signal is sampled and then converted. The output string of figure 1 is {000

100 101 001 010 101 111 111 001 000 001 011 101 110 101}, that is a bipolar 3 bits codification of 16 samples. If the codifications uses 8 bits, the digitalized output signal is more accurate and closer to the original. Another important parameter is the sampling frequency $f_s = 1/t_s$.

Shannon Theorem establishes the minimum sampling frequency f_s so that the continuous time signal can be reconstructed from the discrete time samples without distortion and error [4], [5]:

$$f_s > 2 * f_m \tag{1}$$

where f_m is the maximum frequency in the spectrum of the analog signal to sampling. Generally for a good result it is imposed $f_s = 5{*}f_m$ or $f_s = 10{*}f_m$. For example, the standard audio coding format (.wav) is sampled at $f_s = 44.1khz$ and 16/24/32 bits per sample.

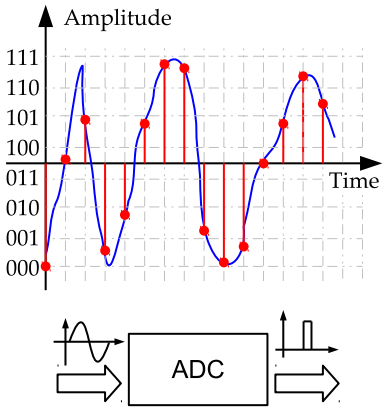


Fig. 1: Example of an analog-to-digital conversion of continuous audio signal to a discrete binary value

III. Filtering

Digital filters [4], [5] are specific causal discrete-time LTI systems. They can be implemented and simulated on digital microprocessors or specialized processors such as DSPs—Digital Signal Processors. The advantage over analog filters is twofold:

- they can be reprogrammed by software on the same hardware;
- the filter coefficients can be changed in real time, thereby obtaining "adaptive" filters.

The main types of digital filters are: Infinite Impulse Response (IIR) filters, and Finite Impulse Response (FIR) filters. The first class of digital filters is distinguished by having an impulse response which continues indefinitely. A FIR filter is a causal LTI system whose impulse response is of finite duration; the transfer function of a FIR filter turns out to be a polynomial in z^{-1}. Essentially, the filtering process is a function in which the value that assumes the output sample is determined by $h(n)$ that operates on neighbors too. The filter operates on a window w of N elements with a width smaller than the signal duration. For this reason it is called *local operator*. Two classes of filter are distinguished: *Linear and Non Linear*. The linear filtering operation returns an output that is a linear combination of the input. A non linear filter is a filter whose output is not a linear function of its input and it is based on *rank function* for operation of erosion and dilation. The Fourier transform is a reversible, linear transform used only for linear filters.

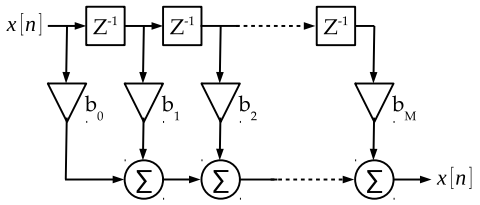


Fig. 2: A discrete-time FIR filter of order N

In an invariant-time system, in order to filter a signal we need to implement the convolution of the input signal $x(t)$ with the filter transfer function $h[t]$

$$y(t) = x(t) * h(t) \tag{2}$$

A discrete-time linear system transforms the input succession $x[n] \in \mathbb{Z}$ to the output succession $y[n] \in \mathbb{Z}$ across the discrete convolution with the discrete transfer function $h[n]$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k] * h[n-k] = \sum_{k=-\infty}^{\infty} x[n-k] * h[k] \tag{3}$$

so that for a discrete-time filter the output is a weighted sum of the values taken from the input at the current time and at earlier times. This is described by the following operation:

$$y[n] = h_0 x[n] + h_1 x[n-1] + \cdots + h_N x[n-N] = \sum_{i=0}^{N} h_i x[n-i] \tag{4}$$

where h_i are the filter coefficients that determine the impulsive response and N is the filter order.

A. Example: mobile mean filter

A simple example of FIR filter is the *mobile mean filter*, the coefficients $b_0, \cdots, b_N$ with N=2 , are determined by equation (5)

$$b_i = \frac{1}{(N+1)} \tag{5}$$

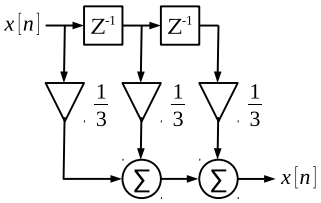


Fig. 3: Example of mobile mean filter with N=2

B. Median filtering

Median filtering [6] is a non-linear, low-pass filtering method that can remove white or salt and pepper noise from an image or an audio signal. It operates directly on samples of acquired signals or images and it has the tendency of edge-preserving smoothing. Median filtering means that for each input sample it is considered a window of N samples about the pivot sample, with N an odd number $\in \mathbb{N}$, where the current pivot sample is in the middle. The window of samples are sorted in ascending or descending order and then the median sample is taken. The median is always positioned in the middle of the window. Figure 4 shows an example of median operation and figure 5 shows 6 possible masks for the 1D and 2D domain. In our case, a ROW mask has been chosen for the window.

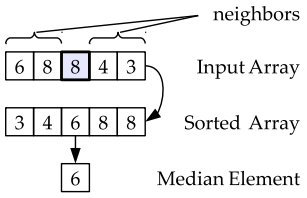


Fig. 4: Example of Median filtering operation on a window of 5 samples. After sorted, the central element is the Median, for this reason the window elements must be odd.

Two examples of median filtering are shown in figure 6 for an image (2D domain) and in figure 7 for an audio signal (1D domain). It should be recalled that if m is the *median* among an ordered set of numbers, the same amount C of numbers will precede and follow that median. If the cardinality of that set is N then it follows that

$$C = (N-1)/2 \tag{6}$$

then C represent the width of the domain boundary. The condition for the 1D signal is that the first term of the samples

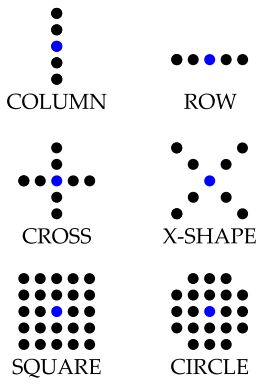


Fig. 5: Median Filter's masks: Column (5x1) and row (1x5) are a one dimensional (1D) masks, the other are 5x5 two dimensional (2D) masks; the pivots elements are highlighted in blue.

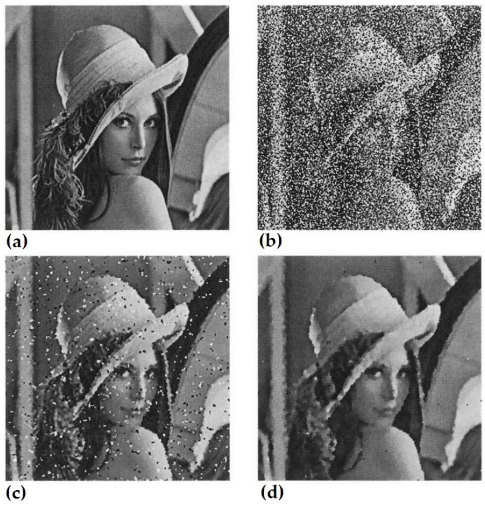


Fig. 6: Example of Median filtering on an image, A) Original image; B) Lenna with 40% additive impulse noise; C) Processed image using 3x3 median filter; D) Processed image using 5x5 median filter

C and the last term C can not be filtered because they have no neighbours. These samples will be copied without any processing. The novelty consists in determining the median element without ordering the window. For each $m_i \in w$ it is useful to count the number of elements equal to, greater and lower than each current element in the window, $m_i \in w$ *is the median if*

$$\exists \text{ exactly } C \text{ elements} > m_i \tag{7a}$$
$$\exists \text{ at least } C+1 \text{ elements} = m_i \tag{7b}$$
$$\exists \text{ exactly } C \text{ elements} < m_i \tag{7c}$$

where w denotes the window and m_i represents the i-th element of w, $i = \{1 \dots N\}$. Of course, all the said conditions must be satisfied.

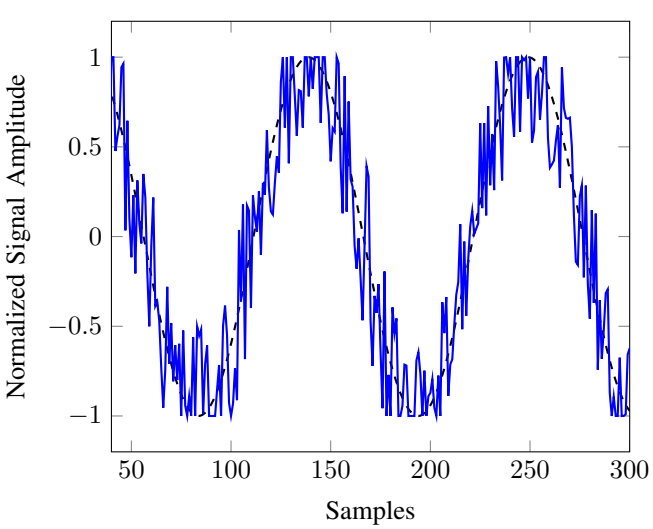
C. Computational complexity

The computational complexity (CC) defines the number of operations required for the extraction of the median from the window of N elements. The CC in an efficient sorting algorithm is estimated as $N{*}ln(N)$ [6]. Instead in the counting algorithm CC is estimated as $N*N$ because each element m_i of the window w is compared to all others and itself. Since N operations are executed in parallel in a GPU we could say that the computational complexity is reduced to N. Figure 8 shows the comparison between sorting algorithm and counting algorithm in terms of computational complexity.

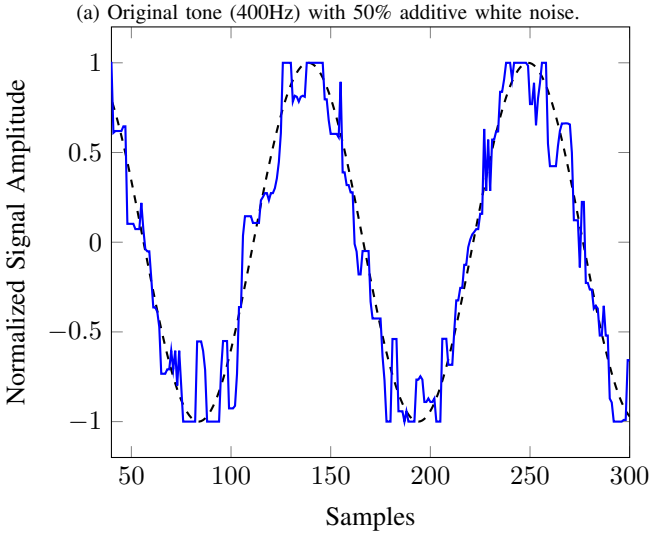
Counting algorithm equals sorting algorithm in terms of computational complexity when N=10.

IV. CUDA

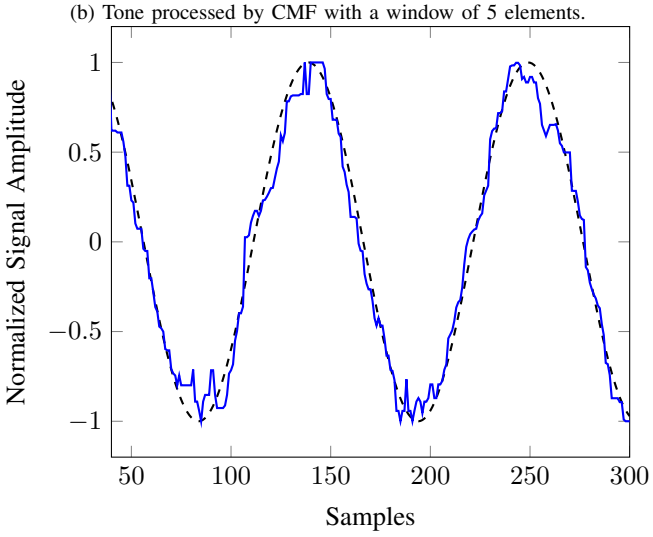
CUDA—Compute Unified Device Architecture—is a parallel computing platform and programming model used with a GPU for general purpose computing simple and elegant.



(a) Original tone (400Hz) with 50% additive white noise.



(b) Tone processed by CMF with a window of 5 elements.



(c) tone processed by CMF with a window of 15 elements

Fig. 7: Median filtering with CUDA extension applied to a fundamental audio tone@400 with additive white noise

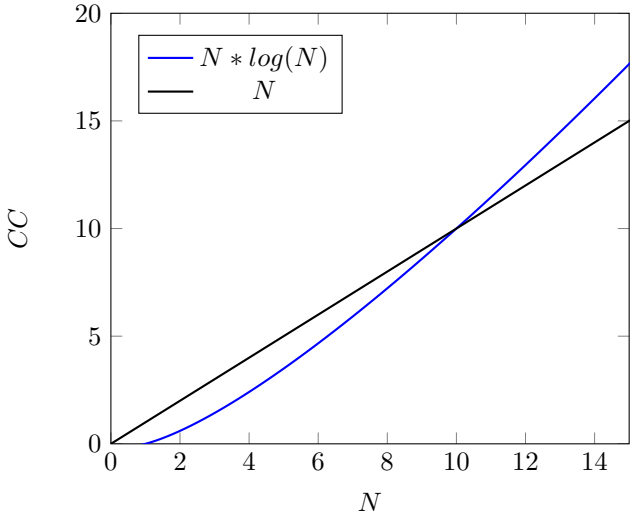


Fig. 8: Comparison between sorting algorithm and counting algorithm in terms of computational complexity

The developer still programs in the familiar C, C++ and incorporates extensions of these languages in the form of a few basic keywords [7], [8], [9]. Programs written in CUDA are compiled by NVIDIA's nvcc compiler and can be run only on NVIDIA's GPU's. The task of modifying the solution to run parallel operations, however, is given to the programmer. CUDA ensures only that after this modification parallel operations can be run on simultaneous GPU threads. A CUDA program may be run on any number of processor cores, and only the number of processors needs to be known at runtime system. A typical CUDA program consists of two parts: the main executing serially on the CPU (the host), and the kernel, called by the main, which is executed in parallel on the GPU (the device). Several smart solutions have been developed to take advantage of such a parallel paradigm [10], [11] and modularisation [12], [13], [14], [15]. In this work, the median filter has been implemented in CUDA as an extension to the C programming language.

A. CPU versus GPU Architecture

In this section the main differences between the CPU and the GPU are analysed. Originally GPUs—Graphic Processing Units—were used to accelerate the drawing of lines, circles and arcs, rectangles, and character bitmaps, then were also used to perform computations. Instead the CPU architecture allows performing computations directly and it is more evolved and complex with respect to the GPU. GPUs have far more processor cores than CPUs, but because each GPU core runs significantly slower than a CPU core and does not have the features needed for modern operating systems, it is not appropriate for performing most of the processing in every-day computing. GPUs are more suited to compute-intensive operations such as video processing and physics simulations. A GPU is simpler than a CPU constituted by full adders, multipliers and comparators. A program is executed in an

element called *thread*. The GPU has multiple hardware threads per core comparing to CPU that it is constituted by multi-core (2,4,8 or 16) and single thread architecture. The CPU architecture has multi-core and multi-thread at the same time.

Computing threads on a GPU are organized into thread blocks and grid. Each thread belonging to the same block communicates with others via a shared memory and can be synchronized manually by the user. CUDA contains special C function called kernels, which are simply C code that is executed on the GPU on fixed number of threads. Prior to launching a kernel, all the data required for the computation, must be transferred from the host (CPU) memory to the device (GPU) memory across the PCI Express bus. This operation takes a considerable time, because the PCI Express bus is managed by the operation system according to the system requirements and not the single job. If the amount of data is not great, the total time (the computation time plus the data transfer time) may be larger than the execution time of the serial program which does not require data transfer because it operate on the RAM locations directly.

Threads organization in grids and blocks is a logical structure that the user defines at the time of the call to the global function. In the next subsection IV-C we will see how to call a global function and how define the logical structure during call.

B. CUDA paradigm

To write a program in a parallel paradigm we need to follow some rules reported in listing 1 [9], [7]. This procedure is necessary to ensure the proper functioning and data transfers.

```
[...]

__global__ void MyKernel(...){...};

[...]

//allocates variables in DEVICE
cudaMalloc(void** devPtr, size);

//Copy variables from HOST to DEVICE
cudaMemcpy(dst, src, size, kind);

[...]

//call cuda MyKernel
MyKernel<<<bkdims,thdims>>>(...);

[...]

//Copy variables from DEVICE to HOST
cudaMemcpy(dst, src, size, kind);

[...]

//free DEVICE memory
cudaFree(void** devPtr);
```

Listing 1: CUDA paradigm: sequence of steps for executing a function on a GPU

The keyword **__global__** indicates a function that runs on the device and is called from host code. Such a function is called kernel and runs in multiple instances on several blocks and threads. The number of blocks and threads is determined by using the $<<<, >>>$ symbols, with $bkdims$ and $thdims$ indicate the number of blocks and threads to execute.

cudaMalloc(...) is used to allocate variables in the device memory, it takes the parameter *(void**) devPtr*, a variable address to allocate; *size* is an integer requested to allocate size in bytes, generally it is used *N*sizeof(...)* to calculate it.

cudaMemcp(...) is used to copy data from host to device end vice versa; the needed parameters are: *dst* destination memory address; *src* source memory address; *size* the requested allocation size in byte and *kind* the type of transfer and is a specific token

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost

the first copies the data through the PCI Express bus from HOST to DEVICE, instead the second from DEVICE to HOST.

cudaFree(...) is used to free device memory, being limited, when the computation is ended. In this manner, we indicate the device which are the useful variables to keep in memory for other calculation and which are not.

cudaError: each cuda function returns an enumerator type, named *cudaError* or simply "cuda status". It indicates if the operation in the GPU was successful or not. If all is well, it returns the cuda status: *cudaSuccess*, in case of error the cuda status will be a keyword describing the type of error. A complete list of cuda errors can be found on NVIDIA site in "related_page/cudaError" [16].

C. Indexes linearization

Indexes linearization is an important issue in cuda programming, so great attention is required. Indexes linearization technique allows us to express a multidimensional index as one dimensional. Let us express the concept with some example.

1) Example of indexes linearization, from 2 indexes to 1: Assume that we have a squared sheet and y is the row index, row is the row size, x is the column index and col is the column size; the element $(y, x) = (2, 5)$ in a linear index is determined by equation (8)

$$Id = y * row + x \tag{8}$$

as shown in figure 9; note that the index starts from 0.

2) Example of indexes linearization, from n indexes to 1: This example extends the previous, i.e. assume to have a new dimension where k is the sheet index and $sheet$ is the block size; now an element in the position $(k, y, x) = (2, 2, 5)$ has the linear index determined by equation (9)

$$Id = k * row * col + y * row + x \tag{9}$$

If we have one more dimension, for example the ream index r, the linear index Id is determined by equation (10)

$$Id = r * sheet * row * col + k * row * col + y * row + x \tag{10}$$

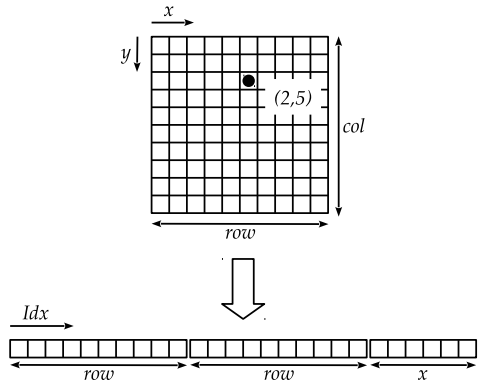


Fig. 9: example of indexes linearization from two to one index.

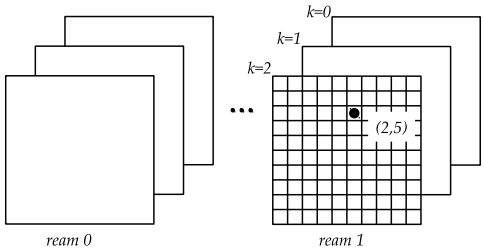


Fig. 10: Example of indexes linearization from n to one index

In general, assume that $I \subseteq \mathbb{Z}^n$ with $n \in \mathbb{N}$ is a discrete and limited set, so that for each dimension it exists a maximum M_i and a minimum m_i:

$$\begin{aligned} I &= < m_1, M_1 >< m_2, M_2 > \cdots < m_n, M_n > \\ &= \prod_{i=1}^{n} < m_i, M_i > \end{aligned} \tag{11}$$

the **i-th volume** is defined by

$$V_i = (M_i - m_i) \tag{12}$$

so index Id is determined by equation (13)

$$Id = x_1 + \sum_{j=2}^{n} \Big(x_j * \prod_{i=1}^{j-1} V_i \Big) \tag{13}$$

V. Algorithm implementation

This section explains the implementation of an efficient sequential and parallel algorithm for a median filtering. The 1D audio signal is called track_in. In a sequential approach, to determinate the median from a window of N elements, first of all, the window must be sorted in ascendant or descendant order, then the median element can be taken. This element is situated in the middle of the sorted window, for this reason N must be an odd number, as shown in figure 4 with $N = 5$. The Serial Median Filter (SMF) is shown in Listing 2, relating to input audio track_in. Then, C is calculated using equation (6) and the window of N elements is created. Moreover, the first C samples of the input track (track_in) are copied from it to the output track (track_out) because they are in the boundary of the track and also the last C samples. Finally, the real Median filter is applied, and in order to filter all samples of track_in the window must select from the beginning to the end of it, so that a scrollable index k is implemented for the loop (k starts from C because the first track_in element starts from 0 and it ends with the $samples - C$ that represent the total number of samples in filter); therefore the window is filled with the samples as shown on the top of figure 4 and the median function is on the window. Median function sorts the window and returns the C-th element that is the median in the window. These procedures are repeated for each sample of interval $[C, sample - C]$

```
// serial median filter SMF
void SMF(int* track_in, int* track_out) {
 int c = (N-1)/2;
 int window[N] = {0};
 //copy the first c boundary elements from TRACK_IN to
      TRACK_OUT
 for (int i = 0; i < c; i++)
  track_out[i] = track_in[i];

 //copy the last c elements from TRACK_IN to TRACK_OUT
 for (int i=samples-c; i<samples; i++)
  track_out[i] = track_in[i];

 //scroll the input track
 for (int k=c; k<samples-c; k++) {
  //fill the window
  for (int i=0; i<N; i++)
   window[i] = track_in[k-c+i];
  track_out[k] = Median(window, N, c);
 }
}

// extract the median element from input array w
int Median(int *w, int N, int c) {
 bool exchange = true;
 int last = N-1, i=0;
 while (exchange) {
  exchange = false;
  for (i = 0; i<last; i++) {
   if (w[i]>w[i+1]) {
    int tmp = w[i];
    w[i] = w[i+1];
    w[i+1] = tmp;
    exchange = true;
   }
  }
  last--;
 }
 return w[c];
}
```

Listing 2: Serial algorithm used to implement the Median filter.

We have used the property described in equation (7) to determine the median. An example of application is shown in figure 11 and we can find on the top an input array, on the bottom the output array, instead, in the center three array: major (**M**), equal (**e**) and minor (**m**). These arrays have the same dimensions of the window and take into account the number of elements greater, equals or minor than the current element. In the example of figure 11, the number 6 has two elements much greater in the window, one element equals to it and two minor elements. In this case the number 6 has all requirements to be the median element as specified in equation (7).

The parallel algorithm (CMF—Median Filter with CUDA extensions) is shown in listing 4, that is the global function, furthermore the cuda model is shown in the listing 1

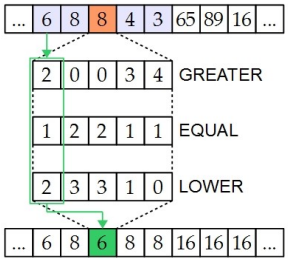


Fig. 11: Example of median filtering operation with a window of 5 elements executed in the GPU with the CMF method

with memory allocations and data transfer. It is important to determine the number of threads for the simulation of the filter. Each sample must be compared to all samples of the N elements of the window to obtain the total number of threads:

$$T_{tot} = samples * N \tag{14}$$

the maximum number of threads per core in the GPU (1024) will be indicated as $nthread$. So, as shown in figure 12, each block will represent the dimension $bk_{area} = bkd * N$ where bkd is gived by

$$bkd = \frac{nthread}{N} \tag{15}$$

finally,we have assumed a mono grid of dimension $grd*1*1$, the total block with dimension $bkd * N$ will be:

$$gnd = \frac{T_{tot}}{bk_{area}} + 1 = \frac{samples}{bkd} + 1 \tag{16}$$

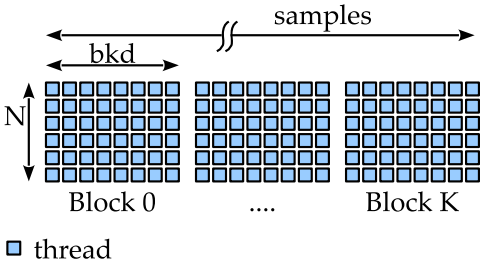


Fig. 12: Threads organization on the blocks

The main program for CMF is shown in listing 3

```
[...]
int bkd=(nthread-(nthread % N)) / N;
int grd=((samples-(samples % bkd))/bkd)+1;

dim3 thdims(bkd, N, 1);
dim3 bkdims(grd, 1, 1);

// call median filter with cuda extension
CMF<<<bkdims,thdims>>>(IN, OUT, c);
[...]
```

Listing 3: This listing shows how to call a global function.

The global CMF function is in listing 4. While in the sequential algorithm *k* was a scrollable index, now in the parallel version it is a global index as in the indexes linearization, example IV-C1, therefore the k-th sample will be assigned to the k-th thread with a correspondence $1:1$. The term C is calculated using equation (6) and takes into account the boundary condition, defining another index *i*. The column index is used to create the window. For each sample, arrays with major, equal and minor elements are created in order to determine the median m_i in the window as shown in figure 11. Each thread executes two comparison to determine the major, minor or equal sample. The median can be found by observing the three arrays vertically. The median is present if a column complies with the specifications (7). Note that the sum of number in this column is always the window width of N.

```
//CMF = Median Filter with CUDA extension
__global__ void CMF(int* in, int* out, int c) {
 // k = input array index
 int k = c+blockDim.x*blockIdx.x+threadIdx.x;
 int M[N] = {0};  //major array
 int e[N] = {0};  //equal array
 int m[N] = {0};  //minor array
 //window index
 int i = k-c+threadIdx.y;
 for (int j=k-c; j<k-c+N; j++) {
  if (in[j]>in[i]) M[threadIdx.y]+=1;
  else if (in[j]<in[i]) m[threadIdx.y]+=1;
   else e[threadIdx.y]+=1;
 }
 for (int j = 0; j<N; j++) {
  if (M[j]==c || m[j]==c || e[j]>=c) {
   out[k] = in[k-c+j];
   return;
  }
 }
}
```

Listing 4: Parallel algorithm used to implement the Median filter in CUDA.

VI. Experiment

For computing results, two different GPUs and an Intel CPU, in two different OSs, have been used: the *NVIDIA's GeForce 820M* in a node with Windows 10, the *NVIDIA's GeForce GTX 480* in a node with Linux MIND 14.4 and the *Intel Core i7-4510U* in a node with Windows 10. The 820M is considered an entry level GPU, instead the GTX 480 is a midrange GPU. The main features of the three devices are given in Table I.

TABLE I: Specifications of the three used device

Spec.	i7-4510U	820M	GTX 480
Total amount of core:	2	96	480
Total amount of Thread:	4		
Total amount of SM:		15	
Core per SM:			32
Frequency:	2 Ghz	775 Mhz	700 Mhz
Total Global Memory:	8 GB (RAM)	2 GB	1.5 GB
Cache:	4 MB	131 KB	786 KB
Shared Memory per block:		49 KB	49 KB
Maximum number of threads per multiprocessor:		1536	1536
Maximum number of threads per block:		1024	1024

Different results are obtained for different OSs and GPUs. The GPU is a convenient solution for big data processing for

the realization of real-time systems in image processing. These results have been obtained by filtering the audio signals as shown in figure 7 with different duration and several sample numbers. During the experiments different window sizes have been used. The work on 1D data can be easily extended to 2D data.

The characteristic of the different audio tracks are reported in Table II and Table III

TABLE II: Audio tracks used for this experiment

Track	Duration	Unit	num of Samples
Track 1	22.7	msec	$1 \cdot 10^3$
Track 2	226.8	msec	$10 \cdot 10^3$
Track 3	2.3	sec	$100 \cdot 10^3$
Track 4	22.7	sec	$1 \cdot 10^6$
Track 5	3.8	min	$10 \cdot 10^6$
Track 6	37.8	min	$100 \cdot 10^6$

TABLE III: Common Audio tracks property.

Parameter	Value	Unit
f_s	44100	hz
codification	32	bit

Figure 13 shows the simulation results: the numbers of samples to process are represented in the x axis; the y axis represents the response time of the SMF and CMF expressed in milliseconds. Figure 13a, 13b and 13c show the obtained results varying the window width from 5 to 9 and then to 15, respectively. Data have been expressed in logarithmic scales and the results highlight the behavior of few samples during filtering. The CPU graphic has a computational complexity, when the filtering of the samples increases, the time increase exponentially. For a few samples the GPU uses more time than the CPU. The reason is the data transfer from the host to device and vice versa through the PCI Express bus, this time isn't negligible. The PCI Express bus is managed by the OS, the data flow through the bus when it isn't busy. A dedicated bus will make the computation more easy. In figure 7 we can see the median effect of the filter in noise reduction. The noise, clearly, can't be totally removed but is reduced. In fact, the spectral analysis still reveals the presence of noise. The same data of figure 13 are reported in the table IV with a more clear result.

To highlight the behavior of a lot of samples during filtering, figure 13c reports a linear scale in figure 13d. The speed gain is about 2 for the 820M and 4 for the GTX 480. In figure 13e and 13f are reported the ratio between the CPU performance and GPU. The trend of the three graphs is ascendant, this demonstrates that GPU computing is more powerful than CPU computing.

To measure the execution time was used the following code.

```
#include <time.h>

[...]

clock_t start, end;
float duration = 0;

start = clock();
/*your codes here*/
end = clock();

//time in milliseconds
duration = (float)(end-start)*1000...
 .../CLOCKS_PER_SEC;
```

Listing 5: Execution time measure method

TABLE IV: The simulation results shown in figure 13 the response time expressed in milliseconds
(SMF = Serial Median Filter)
(CMF = MEdian Filter with CUDA extension)

(a) Case N = 5

num of samples	i7-4510U	820M	GTX 480
1k	2	371	58
10k	14	380	68
100k	132	505	80
1M	1335	1695	570
10M	13063	13853	4756
100M	130404	111905	48810

(b) Case N = 9

num of samples	i7-4510U	820M	GTX 480
1k	2	384	66
10k	19	377	80
100k	155	524	158
1M	1530	1894	881
10M	15006	11542	4648
100M	153771	111498	49190

(c) Case N = 15

num of samples	i7-4510U	820M	GTX 480
1k	2	384	70
10k	21	400	86
100k	204	585	196
1M	1977	2464	766
10M	19663	11441	4540
100M	196776	110202	48820

VII. Considerations

In this section we evaluate the extension of this work to a 2D domain in the real-time scenario. The total number of samples is $100 \cdot 10^6$

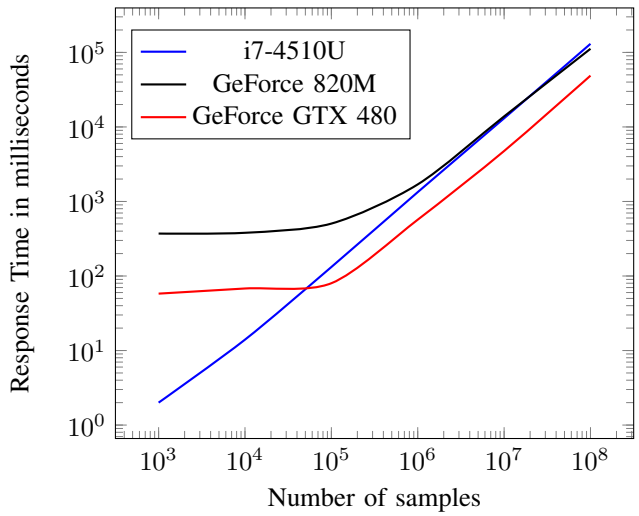
$$s = d \cdot f_s = \frac{d}{T_s} \tag{17}$$

where s represents samples and d represents the duration. So that the samples per sec (sample rate = sr) are given by

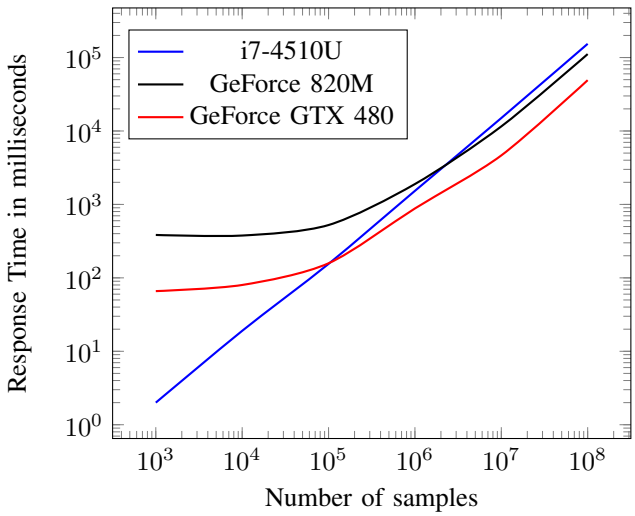
$$sr = \frac{s}{d} = f_s \tag{18}$$

Supposing negligible the data transfer time for many samples, the elaboration capability of the GeForce GTX 480 is evaluated by:

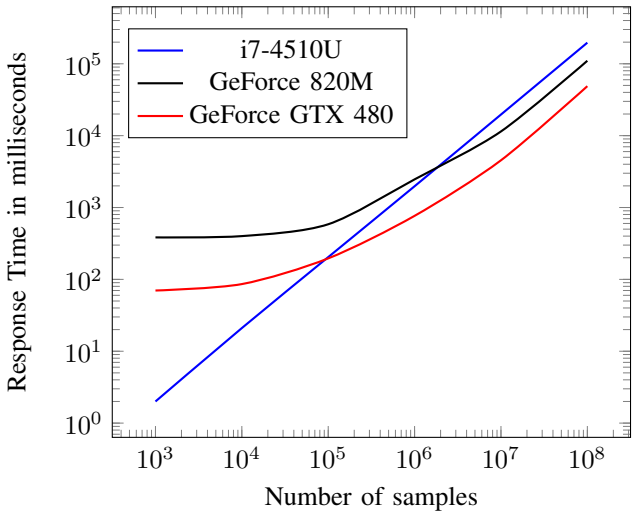
$$sr_{GTX480} = \frac{s}{t_r} = \frac{100 \cdot 10^6}{48820 \cdot 10^{-3}} \sim 2 \cdot 10^6 \left[\frac{samples}{sec}\right] \tag{19}$$



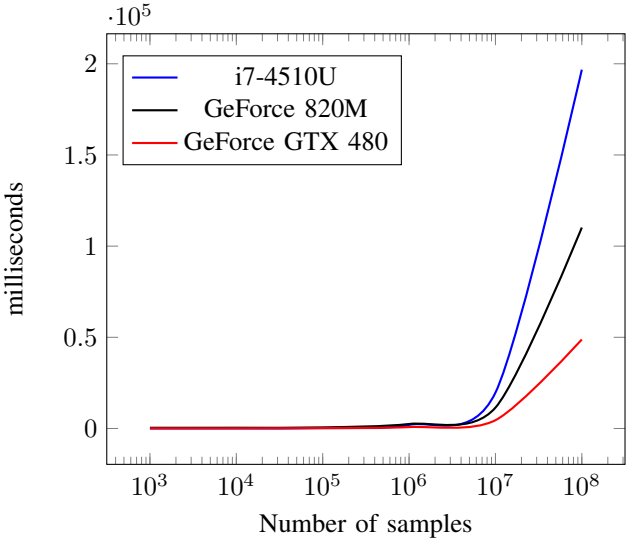
(a) Case N=5 in logarithmic scale



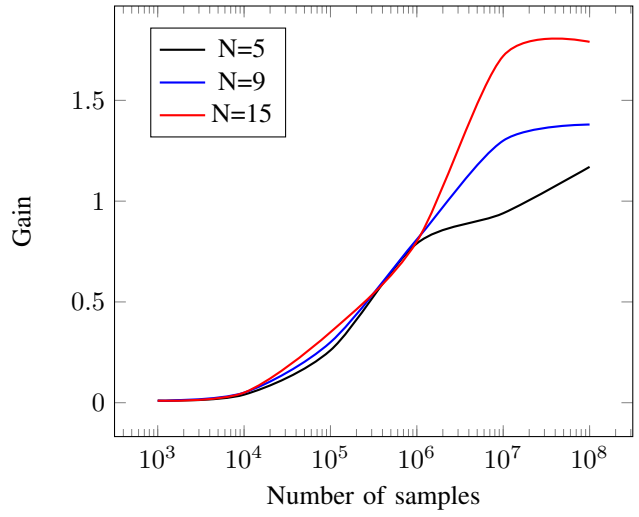
(b) Case N=9 in logarithmic scale



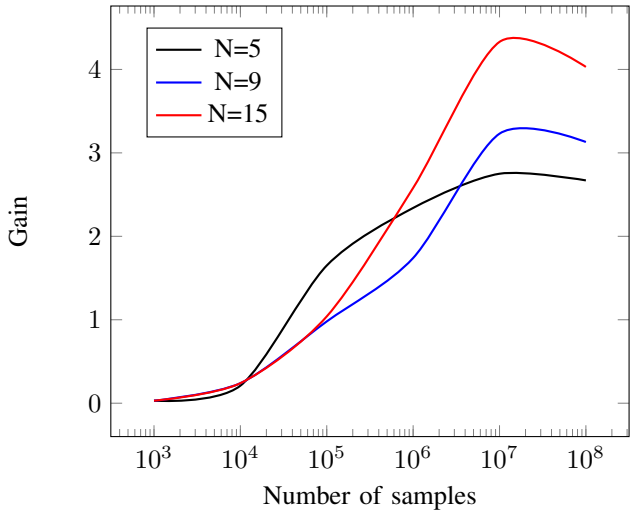
(c) Case N=15 in logarithmic scale



(d) Case N=15 in linear scale



(e) Speed Gain of GeForce 820M



(f) Speed Gain of GeForce GTX 480.

Fig. 13: Simulation results: Comparison between sequential algorithm and parallel algorithm in terms of response time. a) case N=5; b) case N=9; c) case N=15; e and f) Representation of Speed Gain: $CPU_{ResponseTime}/GPU_{ResponseTime}$.

where t_r is the response time. Now we consider three different modern resolutions for image and video: 480p, 720p, 1080p.

TABLE V: Common Image property.

Image	Resolution	number of pixels
480p	640 x 480	307200
720p	1280 x 720	921600
1080p	1440 x 1080	1555200

By considering a 480p image, if it is a RGB (color) image it has three samples per pixel, so that the total amount of samples is

$$s_{RGB@1480p} = 307200*3 = 921600 \sim 1{\cdot}10^6[samples] \quad (20)$$

Applying the CMF takes about 766 milliseconds. If we have a video at 480p@24fps, it means we have the following sample rate

$$sr_{480p@24fps} \sim 22.1 \cdot 10^6 \left[\frac{samples}{sec}\right] \quad (21)$$

In this case the GeForce GTX 480 can not be used for a real-time filtering process on a video at 480p@24fps so that a more powerful GPU is required.

VIII. Conclusions

Other similar works are "High Performance Median Filter" [17] and "Parallel biomedical image processing with GPUs in cancer research" [18] that concern the image processing in medical application with median filtering technique using the branchless vectorized median (BVM) filter [19].

In conclusion, the method reported by equation (7) is effective for the CUDA implementation. The simulation results demonstrate that is possible to obtain gain in response time with an entry level GPU, allowing real-time image and audio filtering. However, the bottleneck of these systems is the PCI Express bus, for dedicated and direct bus through GPU/RAM and CPU/ GPU the response time is reduced.

References

[1] M. Woźniak, D. Połap, M. Gabryel, R. K. Nowicki, C. Napoli, and E. Tramontana, "Can we preprocess 2d images using artificial bee colony?" in *Proceedings of International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, ser. Lecture Notes in Artificial Intelligence, vol. 9119. Springer, 2015, pp. 660–671, DOI: 10.1007/978-3-319-19324-3_59.

[2] D. Polap, M. Wozniak, C. Napoli, E. Tramontana, and R. Damasevicius, "Is the colony of ants able to recognize graphic objects?" in *Information and Software Technologies*, ser. Communications in Computer and Information Science, G. Dregvaite and R. Damasevicius, Eds. Springer International Publishing, 2015, vol. 538, pp. 376–387.

[3] M. Wozniak, C. Napoli, E. Tramontana, and G. Capizzi, "A multiscale image compressor with rbfnn and discrete wavelet decomposition," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2015, pp. 1219–1225, doi: 10.1109/IJCNN.2015.7280461.

[4] P. Diniz, E. da Silva, and S. Netto, *Digital Signal Processing: System Analysis and Design*. Cambridge University Press, 2010.

[5] S. Salivahanan and A. Vallavaraj, *Digital Signal Processing*. McGraw-Hill Education (India) Pvt Limited, 2001.

[6] M. Juhola, J. Katajainen, and T. Raita, "Comparison of algorithms for standard median filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 204–208, Jan 1991.

[7] C. Nvidia, "Nvidia cuda compute unified device architecture programming guide," *NVIDIA Corporation*, 2007.

[8] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010.

[9] "https://developer.nvidia.com/cuda-zone."

[10] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A novel cloud-distributed toolbox for optimal energy dispatch management from renewables in igss by using wrnn predictors and gpu parallel solutions," in *Proceedings of IEEE International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*, Ischia, Italy, June 2014, pp. 1077–1084.

[11] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappala', "A Cloud-Distributed GPU Architecture for Pattern Identification in Segmented Detectors Big-Data Surveys," *Computer Journal*, 2014.

[12] C. Napoli, G. Pappalardo, and E. Tramontana, "A mathematical model for file fragment diffusion and a neural predictor to manage priority queues over bittorrent," *International Journal of Applied Mathematics and Computer Science*, vol. 26, no. 1, 2016.

[13] F. Bonanno, G. Capizzi, G. Lo Sciuto, C. Napoli, G. Pappalardo, and E. Tramontana, "A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in openmp," in *Artificial Intelligence and Soft Computing*. Springer International Publishing, 2014, pp. 22–33.

[14] C. Napoli, G. Pappalardo, and E. Tramontana, "An agent-driven semantical identifier using radial basis neural networks and reinforcement learning," in *XV Workshop "From Objects to Agents" (WOA)*, vol. 1260. Catania, Italy: CEUR-WS, September 2014.

[15] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*. Bangkok, Thailand: IEEE, December 2013, pp. 105–110.

[16] "http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit-/docs/online/index.html."

[17] W. Chen, M. Beister, Y. Kyriakou, and M. Kachelries, "High performance median filtering using commodity graphics hardware," in *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, Oct 2009, pp. 4142–4147.

[18] A. Remenyi, S. Szenasi, I. Bandi, Z. Vamossy, G. Valcz, P. Bogdanov, S. Sergyan, and M. Kozlovszky, "Parallel biomedical image processing with gpgpus in cancer research," in *Logistics and Industrial Informatics (LINDI), 2011 3rd IEEE International Symposium on*, Aug 2011, pp. 245–248.

[19] M. Kachelriess, "Branchless vectorized median filtering," in *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, Oct 2009, pp. 4099–4105.