Adaptive Image Denoising IP- Core based on FPGA

Honnambika M
Dept. of E & C
SIET
Tumkur, India

Latha K
Dept. of E & C
SIET
Tumkur, India

***Abstract*—The presence of noise in images can significantly impact the performances of computer vision algorithms and digital image processing. Thus, noise should be removed to improve the robustness of the entire process. Denoising or noise reduction is one of the most essential processes for digital image processing. The main goal of denoising is how to remove the noise while keeping the important features of the image. The denoising methods should not alter the original image, most denoising methods degrade or remove the fine details. This paper presents an Adaptive Image Denoising IP-core (AIDI) for real time applications. Here core first estimates the level of noise in the input image, then applies an adaptive Gaussian smoothing filter to remove the estimated noise. The filtering parameters are computed on-the-fly, adapting them to the level of noise in the image and pixel by pixel to preserve image information (e.g., edges or corners). In this context, hardware acceleration is crucial, and Field Programmable Gate Arrays (FPGAs) best fit the growing demand of computational capabilities. The architecture uses FPGA, it shows the improvements with respect to a standard static filtering approach.**

Keywords —Gaussian noise, noise estimation, Laplacian operator, noise reduction, edge detection. Adaptive Gaussian filtering, Gaussian noise, denoising.

I. INTRODUCTION

Nowadays, computer vision is one of the most evolving areas of Information Technology (IT). Image processing is widely used in several application fields, such as aerospace, medical, or automotive. In every computer vision application, one or more images are taken from a camera, and processed, in order to extract information used for edge detection , features identification, or image registration.

Image processing is widely used in many fields, such as medical imaging, scanning techniques, printing skills, license plate recognition, face recognition, and so on. Unfortunately, the technology provided by modern Charge Coupled Device (CCD) sensors suffers from noise. In a CCD camera there are many potential sources of noise, such as Shot Noise, Dark current, Read Noise and Quantization noise are some of examples. CCD manufacturers typically Combine these on-chip noise sources, and express them in terms of a number of electrons Root Mean Square (RMS). However, in image the level of noise does not depend on the adopted sensor only but also depends on the environmental condition as well. Noise estimation and removal are thus necessary to improve the effectiveness of image processing algorithms.

To estimate how an image is affected by noise, a well characterized noise model must be defined. Since noise sources are random in nature, their values must be handled as random variables, described by probabilistic functions. In fact, Dark Current, proportional to the integration time and temperature, is modeled as a Gaussian distribution, Shot and Read Noise, caused by on-chip output amplifiers, are modeled as Poisson distributions, and, detector malfunction or hot pixels are modeled by an impulsive distribution.

In most cases, all Gaussian and Poisson distributed noises are combined, approximating the image noise with an equivalent additive zero-mean white Gaussian noise distribution, characterized by a variance σ_n^2.

While the impulsive noise can be removed in a relatively simple way, Gaussian noise removal is a non trivial task, since, to be more effective, the filter must be adapted to the actual level of noise in the image. Noise estimation is therefore a fundamental task. Nonetheless, in modern real-time systems, a software implementation of these complex algorithms cannot be used, since it does not meet real-time constraints. In this context, FPGAs are a good choice to hardware accelerate the noise estimation and removal tasks. This enables subsequent image processing algorithms to fully exploit the remaining timing budget.

This paper presents AIDI: an Adaptive Image Denoising FPGA-based IP-core for real-time applications. The core first estimates the level of noise in the input image. It then applies an adaptive Gaussian smoothing filter to remove the estimated Gaussian noise. The filtering parameters are computed on the-fly, adapting them to the level of noise of the current image. Furthermore, the filter uses local image information to discriminate whether a pixel belongs to an edge in the image or not, preserving it for subsequent edge detection or image registration algorithms. An FPGA-based implementation has been targeted, since FPGAs are increasingly used in real-time systems as hardware accelerators, even in mission-critical applications, such as aerospace field. The paper is organized as follows: Section II gives an overview on noise estimation and removal approaches, and their existing hardware implementations. Section III presents the hardware architecture of the proposed IP-core, while Section IV shows the experimental results. Finally, in Section V, some conclusions are drawn.

II. RELATED WORK

Noise estimation methods, targeting additive white Gaussian noise, can be classified in two categories: filter-based and block-based. With the former method, the noisy image is filtered by a low pass filter to suppress image structures (e.g., edges), and then the noise variance is computed based on the difference between the filtered and the noisy image (called difference image) [1]. With the latter method, the image is split into cells, and the noise variance is computed identifying the most homogeneous cells [2][3].

Proved that filter-based methods work better than block based methods at high noise levels, but they are complex and require high computational load. In addition, filter-based methods assume the difference image as the noise affecting the input image, but this assumption is not true for images with several structures or details.

To tackle this problem, [1] estimates noise by combining a simple edge detector and a low-pass filter. The proposed algorithm has good performances even with high detailed images at different level of noise, and it requires only simple mathematical operations (i.e., convolutions and averaging operations).

Denoising methods can be based on linear or on non-linear models. On the one hand, median and Gaussian filters are commonly used to remove noise, offering a good trade-off between complexity and effectiveness in smoothing out noise. These methods work well in the flat regions of images, but they do not well preserve the image edges, that appear smoothed. On the other hand, denoising methods based on non-linear models (e.g., wavelets-based methods) can handle edges in a better way, but are more complex, and often not applicable in real-time image processing for high resolution images.

In [4] the authors propose an adaptive Gaussian filter which tries to limit the edge smoothing problem of standard Gaussian filtering methods. A large filter variance is effective in smoothing out noise, but, at the same time, it distorts those parts of the image where there are abrupt changes in pixel intensity. This can lead to edge position displacement, vanishing of edges, or phantom edges (i.e., artifacts in the image).

To address this problem, [4] adapts the filter variance to the local characteristics of the input image. It makes use of the local variance of the image, and the estimated Gaussian noise in the image. It has been proven that this adaptive filtering approach succeeds in preserving edges and features of an image, even in presence of noise, better than a static filtering approach.

Hardware implementations of denoising methods have been widely investigated. [5] Propose FPGA-based implementations of median filters. However, median filtering is strictly recommended for impulse noise removal (i.e., Salt and-Pepper noise), while it does not provide good results when the image is affected by Gaussian noise. An FPGA-based implementation of a Gaussian smoother has been proposed in[6], but its main drawback is the non-adaptivity of the filter, which results in edge smoothing. [7] propose implementations of wavelet-based and bilateral filter image denoisers, respectively.

However, none of these works account for a noise estimation module to be included into the hardware architecture. In Cartesian Genetic Programming (CGP) image filters have been proposed. CGP-based filters are able to reduce the noise on the image while preserving edges. Moreover, they can be efficiently implemented on FPGAs requiring few hardware resources. However, since CGP filters are based on evolutionary algorithms, they require a lot of iterations to provide the filtered image, making them inappropriate for real time applications. Hardware implementations of noise estimators have not been deeply investigated by the research community. The proposed architecture wastes a lot of hardware and memory resources to perform sorting and logarithmic operations. Moreover a noise removal module is not included in the architecture. The presented paper introduces a comprehensive FPGA-based architecture, including noise estimation and noise removal in a single IP-core. It targets the estimation and removal of additive white Gaussian noise. The chosen adaptive Gaussian filtering approach ensures edge preserving capability, while the noise estimation algorithm is able to estimate the variance of Gaussian noise with high accuracy[1][4].

The proposed adaptive FPGA-based architecture ensures real time performances, even with 1024x1024 pixels grey-scale images, with 8 bit-per-pixel resolution (bpp). Nonetheless, the proposed architecture uses few hardware resources, allowing to include, in the same device, additional image processing algorithms.

III. AIDI ARCHITECTURE

AIDI is a highly parallelized and pipelined FPGA-based IPcore that gets in input, through a 32-bit interface, a 1024x1024 grey scale image (e.g., from a CCD camera) with 8 bpp and outputs a filtered pixel each clock cycle, through a 25 bit interface. Input pixels are received as a set of 32-bit packets (i.e., 4 pixels are received in a clock cycle), without any header or padding bit. In order to self-adapt the Gaussian filter to the current input image, AIDI applies the approach presented in [4]. This approach can be mathematically formalized as follow:

$$\sigma_f^2(x,y) = \begin{cases} k \cdot \dfrac{\sigma_n^2}{\sigma_{DI}^2(x,y)} & \text{if } \sigma_n^2 << \sigma_{DI}^2(x,y) \\ k & \text{if } \sigma_n^2 >> \sigma_{DI}^2(x,y) \end{cases} \quad (1)$$

where $\sigma^2_f(x, y)$ is the variance of the Gaussian filter to be applied at the pixel of the input image in (x, y) position, σ^2_n is the estimated white Gaussian noise variance of the input

image, k is a constant equal to 1.5, and $\sigma^2_{OI}(x, y)$ is the local variance of the image without noise (i.e., noise free image) in (x, y) pixel, that can be computed as:

$$\sigma^2_{OI}(x, y) = \sigma^2_{NI}(x, y) - \sigma_n^2 \quad (2)$$

where $\sigma^2_{NI}(x, y)$ is the local variance associated with the noisy input pixel image. Basically, this algorithm adapts the variance of the Gaussian filter $\sigma^2_f(x, y)$ pixel-by-pixel, in order to strongly reduce the noise in smoothed image areas (i.e., low image local variance $\sigma^2_{OI}(x, y)$), and to reduce the distortion in areas with strong edges (i.e., high $\sigma^2_{OI}(x, y)$). In other words, $\sigma^2_f(x, y)$ is increased in the first case and decreased in the second one. $\sigma^2_f(x, y)$ can range from values near 0 to 1.5.

AIDI includes three main modules (Fig.1): the Local Variance Estimator (LVE), the Noise Variance Estimator (NVE) and the Adaptive Gaussian Filter.

First, the input pixels feed the NVE and, in parallel, they are stored into an external memory through a 32-bit interface.

The NVE, exploiting the algorithm presented in, computes the Gaussian noise variance (i.e., σ^2_n) affecting the input image. The selected algorithm involves highly parallelizable operations.

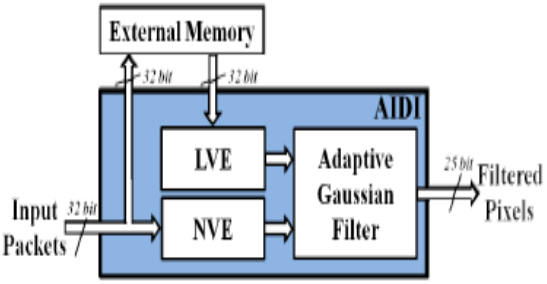

Figure 1: AIDI Internal architecture

It first requires to extract the strongest edges (or features) of the input image exploiting the Sobel features extractor. This task is performed using two 2D convolutions between the input image and the Sobel kernels (Eq. (3)).

Where I(x; y) is the pixel intensity in the (x, y) position of the input image, and G is the edge map associated with the input image. The strongest edges are then extracted by selecting the highest 10% values inside G. example (as shown in Fig.2)

$$G_x = I(x, y) * \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix},$$

$$G_y = I(x, y) * \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ 1 & 0 & 1 \end{vmatrix}$$

$$G = |G_x| + |G_y| \quad (3)$$

Figure 2: Detect edges using the Sobel method

Finally, σ^2_n can be computed as:

$$\sigma^2_n = \left(C \cdot \sum_{I(x,y)=edge} |I(x,y) * N|\right)^2 \quad (4)$$

Where N is the 3x3 Laplacian kernel and C is a constant defined as:

$$C = \sqrt{\frac{\pi}{2}} \cdot \frac{1}{6(W-2)(H-2)} \quad (5)$$

where W and H are the width and height of the input image, respectively (in our architecture W = H = 1024).

Fast Method for Image Noise Estimation Using laplacian operator

Laplacian Operator: We assume that the image is corrupted by additive, white Gaussian noise with unknown deviation σ_n, and the model is given by:

$$I_n(x, y) = I(x, y) + n(x, y) \quad (6)$$

Where x and y are the vertical and horizontal coordinates of a pixel, $I_n(x, y)$, $I(x, y)$ and $n(x, y)$ are the noisy image, the original image and the additive Gaussian noise respectively. Our goal is to estimate the standard deviation σ_n of the noise from the noisy image.

The first step of the "Fast Estimation" method is to suppress the image structures by the following Laplacian operator:

$$N = \begin{vmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{vmatrix} \quad (7)$$

Then the standard deviation of the noise can be using eq. (4)

When the computation of σ^2_n is completed, the overall image is read out from the external memory and provided in input to the LVE. The LVE computes the local variance associated with each input pixel $\sigma^2_{NI}(x, y)$. The local variance of a pixel is defined as the variance calculated on an image window (i.e. patch) centered around the considered pixel (As shown in Fig. 4).

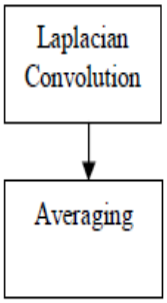

Figure 3: Block diagram of "fast estimation"

To perform this task, LVE applies the following formula:

$$\sigma^2_{NI}(x, y)=S-\left(\frac{1}{T}\sum_{(x,y)\epsilon\, patch} I\,(x,y)\right)^2 \qquad (8)$$

where T is a constant equal to the number of elements in the patch (a 11x11 pixels patch has been selected in our architecture to ensure an accurate local variance estimation), and S is equal to:

$$S=\left(\frac{1}{T}\sum_{(x,y)\epsilon\, patch} I\,(x,y)^2\right) \qquad (9)$$

Since LVE has a pipelined internal architecture, at each clock cycle it provides in output the σ^2_{NI} (x, y) and the related pixel values composing the patch.

The Adaptive Gaussian Filter receives the σ^2_n computed by NVE, and the outputs of the LVE. The filter computes equations (1) and (2), in order to find the best filter variance value (i.e., σ^2_f (x, y)). After this computation, this module applies the Gaussian smoothing on the current received pixel.

The Gaussian filtering operation is performed by means of a 2D-convolution on the input image with a 11x11 pixels Gaussian kernel. The selected filter size allows to accurately represent the Gaussian function with variance values in the selected range (i.e., (0, 1.5], as described before). The values of the Gaussian kernel are adapted pixel-by-pixel, depending on the computed σ^2_f (x, y), as described in Subsection IV -C.

In the following subsections all the hardware implementation details of the AIDI modules are deeply analyzed.

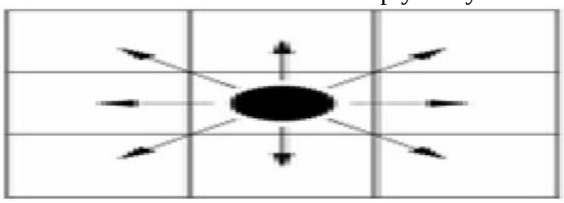

Figure 4: Pixel neighboring comparison

A. Noise Variance Estimator

The NVE module receives the input image through a 32-bit interface (4 pixels are received at each clock cycle), and it provides in output the estimated white Gaussian noise variance σ^2_n affecting the image. The internal architecture of NVE is shown in Fig. 5.

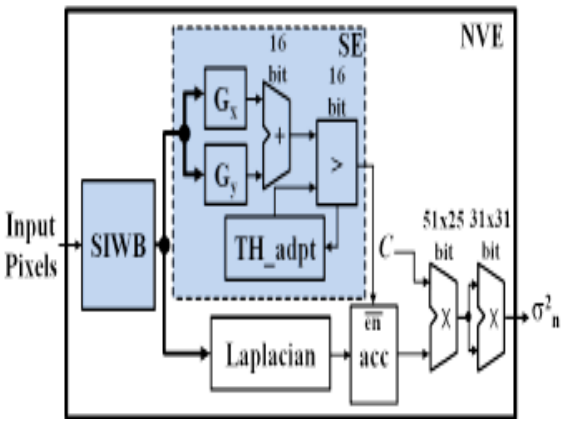

Figure 5: NVE Internal architecture

Since NVE must perform operations involving patches (see Sec. IV), in order to speed up the computation, the input pixels are stored exploiting a circular buffering approach, implemented by the Smart Image Window Buffer (SIWB) of Fig. 6.

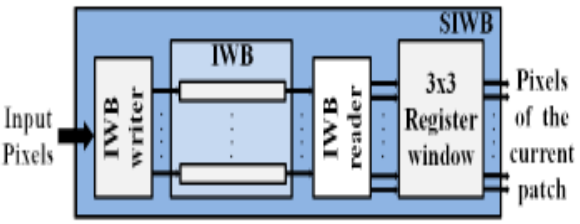

Figure 6: SIWB Internal architecture

Input pixels, grouped in 32-bit packets, are sent to the IWB writer that serializes the pixels using a FIFO, and stores them inside the Image Window Buffer (IWB in Fig. 3). IWB is composed of 3 FPGA internal Block-RAMs (BRAMs), each devoted to store an entire image row. 3 BRAMs are used since pixels from 3 different rows of the image are needed at the same time, to perform the required operations on a 3x3 pixels image patch.

Initially, the IWB writer fills each BRAM, starting from the top one to the bottom one.

During a convolution operation image borders are not processed [8], thus, when all BRAMs are filled, the pixels necessary to process the second row of the image are available to be read-out. While the second row is being processed, pixels associated with the fourth row of the image are

received. They overwrite the content of the BRAM that contains the oldest row (i.e., the first row in this case).
In general, while the i-th image row is being processed, pixels of the (i+2)-th image row are being received. The IWB writer stores received pixels in the BRAM that contains the ones associated to the (i-1)-th image row (i.e., IWB works as a circular buffer). This buffering approach leads to two advantages: (i) when the 3 BRAMs are filled, all required pixels to compute a row are available, allowing a pixel every clock cycle to be processed; (ii) it completely avoids any access to the external memory, because when an image row in the buffer is overwritten by a new one, the data of the replaced row are not needed for the following computations.
The pixels of the image, associated with the current 3x3 patch, are read-out from the IWB by the IWB reader. IWB reader is a Finite-State-Machine (FSM) charged of reading out the pixels from the IWB and providing them to the 3x3 Register window in the right order. Basically, when all pixels needed to process the i-th image row (i.e., pixels from the i-1th row to i+1th row) are stored in the IWB, the IWB reader can start to read a pixel from each BRAM of the buffer. Read pixels are loaded into the first column of the 3x3
8-bit FFs Register Window. Each row of the 3x3 Register windows is a shift register. Thus, at the next clock cycle, when another column of 3 pixels is loaded, the previous column is shifted to the next position. Whenever the 3x3 Register windows is filled with all the pixels of a patch, they are provided in output of the SIWB. It is important to highlight that the IWB writer loads the image rows in the IWB as in a circular buffer. Thus, the image rows are stored in the IWB in an out-of-order manner (w.r.t. the original image).
Consequently, IWB reader must rearrange the position of the pixels in order to store them in the 3x3 Register windows with the same order as in the original image. In this way, at each clock cycle, the pixels of the current patch are provided in output of the SIWB in the right order.
The outputs of SIWB feed the two main modules of LVE: the Sobel Extractor (SE in Fig. 5), and the Laplacian. Basically, SE extracts the features from the input image and asserts its output flag only if the currently processed pixel is one of the 10% strongest features in the image.First, SE computes the operations reported in Eq. (3). The Gx and Gy modules receive in input the pixels of the current 3x3 patch and compute the 2D convolutions between the input pixels and the Sobel kernels. These two modules are internally implemented as a MUL/ADD tree composed of 6 multipliers (only 6 values are different from zero in Sobel kernels) and 3 adder stages, for a total amount of 5 adders. Moreover, since the Sobel kernel factors can only be equal to 1, -1, 2 or -2, in order to reduce the area occupation, the multipliers are replaced by a wire, a sign inverter, a shifter, and a sign inverter & shifter, respectively.
The outputs of the Gx and Gy are then added together, through a 16 bit adder, to find the G value (see Eq. (3)). The computed G is compared with a threshold in order to set the SE output only if the current pixel is one of the 10% strongest features in the image.
The threshold value cannot be determined at design time since it strongly depends on the camera and environment conditions. Thus, the TH adpt module (see Fig. 5) is in charge of calculating the initial threshold value and adapting it frame by frame, by simply applying Algorithm1.
where N target features represents the strongest features in the input image (i.e., the 10% of the complete image).

Algorithm 1 Adaptive Thresholding algorithm

$N\_target\_features \leftarrow 0.1 * size(G)$
$Gap \leftarrow N\_Sobel\_features - (N\_target\_features)$
$Offset \leftarrow Gap * (0.5/3000) * Current\_TH$
if $Gap < -3000 \;||\; Gap > 3000$ **then**
  $New\_TH \leftarrow Current\_TH + Offset$
else
  $New\_TH \leftarrow Current\_TH$
end if

Gap is the difference between the current number of extracted Sobel features (N Sobel features) and N target features. If the value of Gap is less than -3000 or more than 3000, the current value of the threshold (i.e., Current_TH) is incremented or decremented (depending on its value) by one Offset. The new calculated value for the threshold (i.e., New_TH) represents the threshold to be provided in input to the comparator for the next input image. Since at high frame rates the image conditions between two consecutive frames are approximately the same, the threshold value calculated from the previous frame can be applied to the current processed frame. This task is performed for every input frame, in order to maintain the number of extracted features around N target features. Obviously, at startup the Current TH is initialized to a low value, and experiments using a MATLAB implementation of the NVE, applied on the Affine Covariant regions Datasets [9], have shown that TH adpt need a maximum of 8 frames to reach a stable threshold value.
In parallel to the SE operations, the Laplacian module computes the convolution between the input image and the 3x3 Laplacian Kernel (see Sec. III) This operation is performed adopting the same approach used in the Gx and Gy modules.
Although, in this case the MUL/ADD tree is composed of 9 multipliers (all Laplacian Kernel factors are different from zero) and 4 adder stages, for a total amount of 8 adders.
The Laplacian output is provided in input to an accumulator (acc in Fig. 5). This accumulator is enabled only when SE provides in output a zero, in other words only when the current processed pixel is not one of the 10% strongest features. In this way, when the complete image has been received acc contains the value of the sum in Eq. (4).
The following two multipliers conclude the computation of Eq. (4). To ensure a minimal error, the C constant needs to be represented in the 0.25 fixed-point formats and, for the same reason, the following multipliers maintain the same number of bits for the fractional part. The estimated noise variance in output is then truncated to 12.25 fixed-point formats. Thus, the

NVE is able to estimate Gaussian noise variance values up to 4000.
Finally, to improve the timing performances of the NVE module, pipeline stages have been inserted in the MUL/ADD trees and between the two output multipliers.

B. Local Variance Estimator

The LVE module receives in input the pixels read from the external memory, and it provides in output σ^2_{NI} (x, y), computed exploiting Eq. (8). The internal parallel architecture of LVE is shown in Fig. 7.
It is composed of three main blocks: the SIWB, the Mean2 and the S-comp. Since both Mean^2 and S-comp perform operations involving patches, the input pixels are stored exploiting the same buffering approach adopted in the NVE module (i.e. SIWB explained in Sec. IV-A). The only difference concerns the IWB, which is composed of 11 BRAMs, because the LVE operations involve 11x11 pixels patches, as discussed in Sec. III.

The SIWB output pixels are provided in input to the Mean^2 and the S-comp modules. Moreover, the SIWB output pixels are also provided in output of LVE.
Mean^2 computes the second term of Eq. (9). The received pixels are sent to the ADD tree that computes the sum by means of a balanced tree composed of 7 adder stages, for a total amount of 120 adders. Finally, the output of the tree is sent to the two following multipliers to complete the computation of the second term of Eq. (9). To ensure a high precision, the value of the 1/T constant and of the two multiplier outputs are represented in fixed-point format, with 15 bit for the fractional part.In parallel to the operations performed by Mean^2, S-comp computes the S variable (see Eq. 9)).

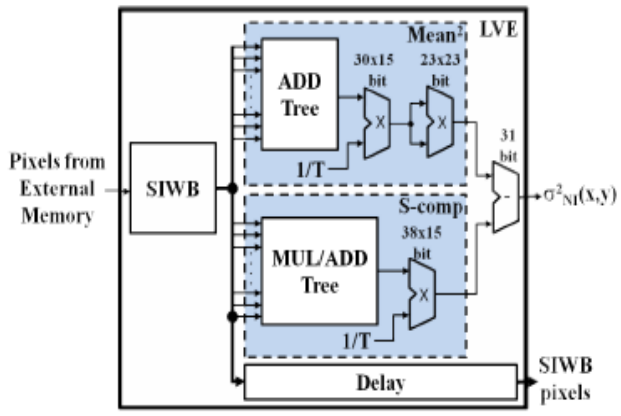

Figure 7: LVE Internal architecture

The outputs of SIWB are provided in input to the MUL/ADD Tree. This tree is composed of a multiplier stage (i.e., 121 8x8-bit multipliers), that computes the square of the pixels in the current patch, and 7 adder stages (i.e., 120 adders), that compute the sum in Eq. (8). In order to obtain the S value, the output of the tree is multiplied by the 1/T constant.
Finally, the local variance σ^2_{NI} (x, y) is computed as the difference between the output of the S-comp module and the one of the Mean^2 module, resorting to a 31-bit subtractor.
As shown in Fig. 7, in order to reduce the area occupation, the data parallelism of each arithmetic component (i.e., multiplier or subtractor) has been truncated to a fixed format able to represent the maximum achievable value. The maximum values obtainable during the computation have been defined exploiting an exhaustive validation campaign using a MATLAB LVE implementation, applied on the Affine Covariant Regions Datasets.
Moreover, several pipeline stages have been inserted to improve the timing performances of the LVE module. For this reason, since σ^2_{NI} (x, y) must be provided in output with the associated patch, the SIWB pixels are delayed in order to synchronize the LVE outputs.

C. Adaptive Gaussian Filter

Gaussian Filter receives the σ^2_n, the σ^2_{NI} (x, y), and the pixels in output from the SIWB of the LVE (see Sec. III-B), and it outputs a filtered pixel each clock cycle. The internal architecture of this module is summarized with Fig. 8.
The Adaptive Gaussian Filter is composed of three main modules: the Filter Variance Estimator (FVE), the Kernel Factors Selector (KFS), and the Gaussian Filter. FVE computes σ^2_f by applying Eq. (1). Thanks to a test campaign using a MATLAB implementation of the Adaptive Gaussian Filter, applied on the Affine Covariant Regions Datasets, it is possible to understand that Eq. (1) can be modelled exploiting Algorithm 2.
The selected model allows a very efficient hardware implementation of the selection condition, by simply adopting a shifter and a comparator (see Fig. 8). Then, σ^2_f (x, y) is computed using a pipelined divider and a multiplier, and it is provided in input to KFS.
This module aims at defining the Gaussian kernel factors associated with the current σ^2_f (x, y). These values cannot be computed in real-time, because the associated formula [8] is very complex and time consuming, so they are precomputed and stored inside the hardware.

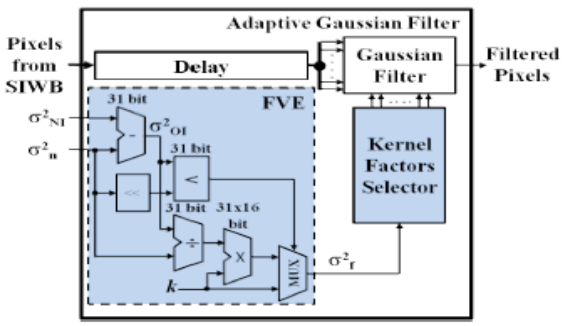

Figure 8: Adaptive Gaussian Internal architecture

Since each value of σ^2_f (x, y) (represented using 31 bit) has a different associated kernel of 121 factors (i.e., the size of the kernel used to perform the filtering task is 11x11 pixels), a huge amount of data should be stored (2^{31}. 121 kernel factors). In order to reduce the required memory resources, in the proposed hardware implementation, the range of σ^2_f (x, y) (i.e. (0, 1.5], see Sec. III) has been discretized adopting a resolution of 0.1.

Algorithm 2 Modelled selection condition

if $\sigma^2_{OI}(x,y) < 2\sigma^2_n$ then
  $\sigma^2_f(x,y) \leftarrow k \cdot \frac{\sigma^2_n}{\sigma^2_{OI}(x,y)}$
else
  $\sigma^2_f(x,y) \leftarrow k$
end if

In this way, the number of sets of 121 Gaussian kernel factors has been limited to 14. Moreover, the required storage capability has been limited exploiting the symmetry of Gaussian kernel, also. Since Gaussian kernels are circularly symmetric matrices, many factors inside them are equal to each others. Fig. 9 shows an example of a 5x5 Gaussian kernel structure, in which the kernel factors to be stored have been highlighted.

a_{00}	a_{10}	a_{20}	a_{10}	a_{00}
a_{10}	a_{11}	a_{21}	a_{11}	a_{10}
a_{20}	a_{21}	a_{22}	a_{21}	a_{20}
a_{10}	a_{11}	a_{21}	a_{11}	a_{10}
a_{00}	a_{10}	a_{20}	a_{10}	a_{00}

Figure 9: Example of a 5 x 5 Gaussian Kernel Structure

Since in a 11x11 Gaussian kernel the number of distinct kernel factors is equal to 21, in the proposed hardware architecture the internally stored data for each σ^2_f (x, y) has been limited to this value.
For these reasons, KFS has been implemented has a cluster of 14 21-input multiplexers, in which each multiplexer is driven by the same selection signal, whose value is defined depending on the current σ^2_f (x, y). In this way, the cluster of multiplexers is able to provide in output the 21 factors useful to represent the Gaussian kernel associated with the current σ^2_f (x, y). Finally, the multiplexer outputs are duplicated in order to reconstruct the complete set of 121 kernel factors for a given σ^2_f (x, y).
The reassembled set of kernel factors are then provided in input to the the Gaussian Filter together with the input pixels from the SIWB, that are delayed to be synchronized with the kernel factors. Then, Gaussian Filter computes the 2D convolution between the input pixel patch (i.e., Pixels from SIWB in Fig. 6) by means of a MUL/ADD tree composed of a multiplier stage (i.e., 121 multipliers) and 7 adder stages (i.e., 120 adders).

IV. EXPERIMENTAL RESULTS

To evaluate the hardware resources usage and the timing performances, the proposed architecture has been synthesized, resorting to Xilinx ISE Design Suite 14.4, on a Xilinx Virtex 6 VLX240 FPGA device. Post-place and route simulations have been done with Modelsim SE 10.0c. Table I shows the resources utilization and the maximum operating frequency of each module composing AIDI.

To compare our architecture with the FPGA-based architectures for noise estimation and static Gaussian filtering presented, AIDI has been also synthesized on a Virtex II FPGA. Concerning the NVE module, it uses 3,202 LUTs and 3 BRAMs, while the real-time noise estimator presented uses 4,608 LUTs, 72 BRAMs and 24 DSP elements.
The performances achieved by AIDI have been also compared with the architecture presented in [6]. Regarding the area occupation on a Virtex II FPGA device, the proposed architecture uses 37,695 LUTs and 24 BRAMs, whereas the FPGA-based static Gaussian filter presented in [6] uses 22,464 LUTs, 39 BRAMs and 32 DSP elements. The higher logic resource occupation (i.e., LUTs) of the proposed architecture is due to two main aspects. The former concerns the kernel used to perform the filtering task, that in AIDI is 11x11 while in [6] is 7x7 (i.e., the 7x7 kernel size does not provide high filtering performance for high level of noise). The latter regards the adaptivity provided by AIDI that is not supported by [6]. Moreover, AIDI provides better timing performance than [6].
In order to evaluate the improvements provided by AIDI w.r.t. a static Gaussian filtering approach, an evaluation campaign has been performed on the image dataset reported in Fig.7.
On these images, different levels of white Gaussian noise have been injected, spanning from a noise variance of 100 to 4,000, exploiting the imnoise function provided by the MATLAB Image Processing Toolbox. Fig. 8 shows some examples of the injected noise on an image.
The benefits provided by the adaptivity have been quantified computing the Mean Square Error (MSE):

$$MSE = \frac{1}{HSE} \sum (I(x, y) - I_F(x, y))^2 \qquad (11)$$

where H and W are the height and the width of the input image, and I(x; y) and I_F (x; y) are the pixel intensities in the (x, y) position of the noise free and the filtered images, respectively.

Each noisy image has been filtered using:

(i) A static 11x11 Gaussian filter (with a σ^2_f equal to k (see Sec. IV).
(ii) A MATLAB model of AIDI (Adaptive (SW)), involving the double precision.
(iii) The AIDI hardware implementation (Adaptive (HW)), which involves fixed-point representation. The graphs in Fig. 12 plot the trends of the MSEs, computed for each image composing the adopted image dataset (see Fig. 11), versus the variance of the injected noise. Fig. 12 highlights two main aspects:

1) The error introduced by the fixed-point representation w.r.t. the double precision implementation can be neglected (Adaptive (SW) vs. Adaptive (HW) in Fig. 12)
2) The MSE associated with the output of AIDI is always lower than the one affecting the output of a static Gaussian filter (Adaptive (HW) vs. Static in Fig. 12). Moreover, the benefits increase for noise levels with $\sigma^2_f \leq 1;000$, while for higher noise levels, the improvement decreases because the local variance of the image is greatly influenced by the noise, and so it cannot be accurately computed.

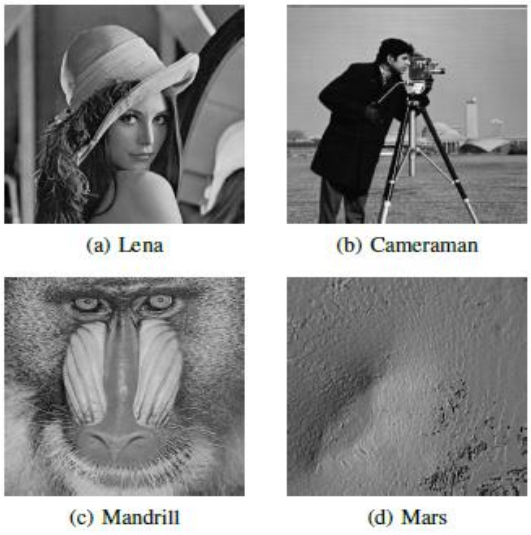

Figure 10: Image dataset exploited for the evaluation campaign

Finally, to prove the effectiveness of the proposed FPGA based adaptive filter in preserving edges w. r. t. a standard static Gaussian filtering approach, the images filtered with both methods have been provided in input to a Laplacian edge detector. Fig. 10a shows an example of image affected by white Gaussian noise with σ^2_n = 1,500, while Fig. 12b, Fig. 12c, and Fig. 12d show the edges extracted from the non-filtered image, the filtered image with a static Gaussian filter, and the image filtered with AIDI, respectively. Despite the high injected noise, AIDI is able to filter the image without smoothing edges, improving the performance of the edge detector. Instead, the static Gaussian filter outputs a smoothed image, in which edges are weakened and difficult to be detected.

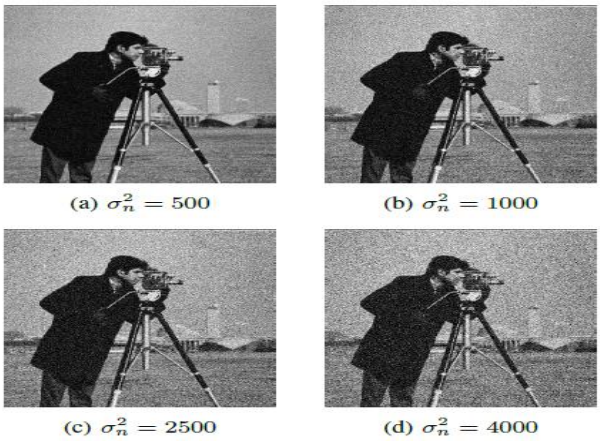

Figure 11: Example of injected level of noise

V. CONCLUSION

This paper presented AIDI a high performance FPGA based image denoiser for real-time applications. This IP core enables to self adapt the filtering parameters to the level of noise in the input image pixel by pixel, resulting in a more accurate filtered image.
The experimental results show a strong improvement of the quality of the filtered image w.r.t. the one obtained from a static Gaussian filter, especially for noise level with $\sigma^2_n \leq 1;000$. These enhancements allow to increase the precision of all the modules, composing an image processing chain, that receive in input the filtered image (e.g., edge detector).

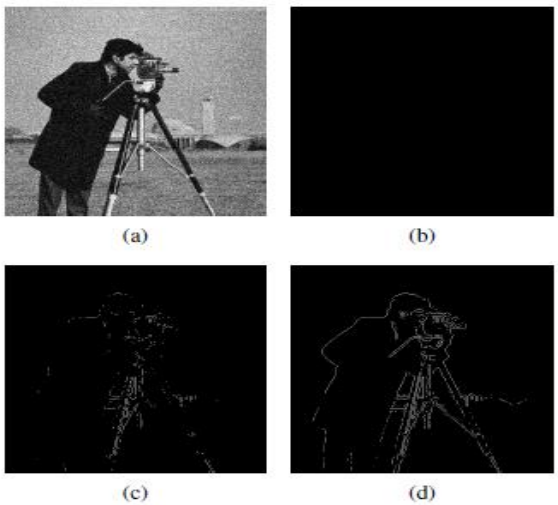

Figure 12: Laplacian edge extraction – (a) Noisy image in input (σ^2_n = 1500) (b) Edge extracted from noisy image (c) Edge extracted From the image filtered by a static 11 x 11 filter (d) Edge extracted from image filtered by AIDI

ACKNOWLEDGEMENT

I would like to convey my hearty thanks to all those who have helped me in the successful completion of my project.

I am deeply indebted to my Principal **Dr. M.N. S.M.Shashidhara.** His constant support and valuable guidance was a source of inspiration for me.

I express my deep sense of gratitude to **Dr. Eshwarappa,** Professor and Head, department of Electronics and Communication Engineering, for providing me the motivation, confidence and support required for completing this project.

I feel grateful to Mrs. Latha K Assistant Professor, department of Electronics and Communication Engineering, for accepting to be my internal guide and for his constant and valuable guidance in completing this project.

Finally, I take this opportunity to extend my whole hearted thanks, gratitude and respect to my parents, my brother, all my friends and well wishers, for giving me their valuable advices and support at all time and in all possible ways, and without whom it would not have been possible to successfully complete my project.

REFERENCES

[1] S.-C. Tai and S.-M. Yang, "A fast method for image noise estimation using laplacian operator and adaptive edge detection," in Proc. Of 3rd international Symposium on Communications, control and Signal Processing (ISCCSP), pp. 1077 – 1081, 2008.

[2] F. Russo, "A method for estimation and filtering of gaussian noise in images," IEEE Transactions on Instrumentation and Measurement, vol. 52, no. 4, pp. 1148 – 1154, 2003.

[3] J. Tian and L. Chen, "Image noise estimation using a variation-adaptive evolutionary approach," IEEE Signal Processing Letters, vol. 19, no. 7, pp. 395 – 398, 2012.

[4] G. Deng and L. Cahill, "An adaptive gaussian filter for noise reduction and edge detection," in Proc. of Nuclear Science Symposium and Medical Imaging Conference, pp. 1615 – vol.3, 1993.

[5] Z. Vasicek and L. Sekanina, "An area-efficient alternative to adaptive median filtering in FPGAs," in Proc. of International Conference on Field Programmable Logic and Applications (FPL), pp. 216 – 221, 2007.

[6] Joginipelly, A. Varela, D. Charalampidis, R. Schott, and Z. Fitzsimmons, "Efficient FPGA implementation of steerable Gaussian smoothers," in Proc. of 44th Southeastern Symposium on System Theory (SSST), pp. 78 – 82, 2012.

[7] T. Q. Vinh, J. hyun Park, Y.-C. Kim, and S. H. Hong, "FPGA implementation of real-time edge-preserving filter for video noise reduction," in Proc. of International Conference on Computer and Electrical Engineering (ICCEE), pp. 611 – 614, 2008.

[8] R. Gonz´alez and R. Woods, Digital image processing 3rd edition. Prentice Hall, 2007.

[9] "University of Oxford - Affine Covariant Regions Dataset." www.robots. ox.ac.uk/_vgg/data/data-aff.html.

[10] F.-X. Lapalme, A. Amer, and C. Wang, "FPGA architecture for realtime video noise estimation," in Proc. of International Conference on Image Processing, pp. 3257 – 3260, 2006.