Efficient Implementation of Matrix-based Image Processing Algorithms for IoT Applications

Sorin Zoican * and Roxana Zoican

Posted Date: 7 April 2025

doi: 10.20944/preprints202504.0438.v1

Keywords: discrete cosine transform; random sample consensus; Visual DSP++, Blackfin processor; CNN accelerator; Internet of Things

 doi:10.20944/preprints202504.0438.v1

Article

Efficient Implementation of Matrix-Based Image Processing Algorithms for IoT Applications

Sorin Zoican * and Roxana Zoican

POLITEHNICA Bucharest National University for Science and Technology; sorin@elcom.pub.ro

Abstract: This paper analyses an implementation approaches of matrix-based image processing algorithms. As an example, an image processing algorithm that provides both image compression and image denoising using random sample consensus and discrete cosine transform is analyzed. Two implementations are illustrated: one using Blackfin processor with 32 bits fixed point representation and the second using the Convolutional Neural Network (CNN) accelerator in MAX78000 microcontroller. A comparison of these two implementations and the validation using MATLAB with 64 bits floating point representation are conducted. The obtained performance is good both in terms of quality of reconstructed image and execution time and the performance differences between the infinite precision implementation and the finite precision implementation are small. The CNN accelerator implementation, based of matrix multiplication implemented using CNN, has a better performance suitable for Internet of Things applications.

Keywords: discrete cosine transform; random sample consensus; Visual DSP++, Blackfin processor; CNN accelerator; Internet of Things

1. Introduction and Related Work

In Internet of Things (IoT) application, both reducing a transmission bandwidth and removing noise are necessary. The image compression can involve algorithm such, MPEG, but this algorithm is not able to remove the noise in the signal. Removing noise from images is quite a difficult task considering the content of the images and the possible types of noise.

An image is created by reflected light into the camera lens and captures by the sensor, that convert the variable levels of the light into digital signals for each image element (pixel). In the most common sensor, an amplifier is attached to each pixel and adjusts the output making the image darker or brighter, respectively. The output voltage is converted using an analog-to-digital converter where the variance in voltage to each pixel gets a binary value.

Noise represents unwanted content in an image caused by the condition existing when the image is captured: low light, slow shutter, sensor issues [1]. Sharp and sudden disturbances could be appeared in the image, as well as uniform or constant disturbances. There are several noise sources. Most of the noise comes from the sensor or analog-to-digital conversion. For example, gaussian noise is a type of sensor noise as an effect of sensor heat. Another example, the salt and pepper noise manifests as pixels erroneously bright values in dark parts of the image or dark values in bright parts. It is similar with dead pixels, except salt and pepper noise will produce this effect randomly. Usually, analog-to-digital conversion cause this kind of noise or in transmitting images over noisy digital links. One class of denoising methods are based on filtering. Various filters are used to remove the noise in image. However, that filters produce approximations, and the noise characteristics and the unknown positions of noised pixels may cause errors in reconstructed image. This aspect represents motivation to develop non-filtering denoising techniques, based on detection and reconstruction of affected pixels [1]. There are many methods to remove noise (median filtering, discrete cosine transform, wavelet transform, etc.) but each method can lead to blurring of the image or the removal of fine details from the image. On the other hand, the implementation of the noise elimination

 doi:10.20944/preprints202504.0438.v1

algorithm on microcontrollers with relatively limited resources and lower precision are aspects that must be studied.

This paper considers an image processing algorithm that involve compressive sensing and random sample consensus. It involves the discrete cosine transform (DCT) and can be used both to compress the image and to denoise it. This approach could remove efficiently different types of noise combined. The algorithm intensively uses the matrix multiplications.

In the literature is shown that sparse signals can be reconstructed from a reduced number of measurements. Digital images can be represented (for example in a discrete cosine transform – DCT) by a small number of coefficients with significant values and can be considered as sparse or approximately sparse in this domain. Other methods have been developed in the noise elimination from images which employ sparsity [2–6]:

- weighted encoding with sparse nonlocal regularization (WESNR) based on soft impulse pixel detection, weighted encoding, and an integration of the sparsity and non-local self-similarity
- block-matching 3D (BM3D) algorithm based on the grouping of similar blocks, a collaborative filtering (an adaptive filter and a two-stage average adaptive filter) by shrinkage in the transform domain, and then combine back the blocks into a two-dimensional signal
- total-variation (TV) L1 methodology based on solving a minimization problem, considering that the image has a high total variation
- hyperspectral denoising, using the spatio-spectral total variation
- denoising algorithms based on deep learning and convolutional neural networks
- variants of the traditional mean and median filters

The compressive sensing idea is based on the sparsity of the sampled signal. Sparsity means that a discrete-time signal depends on several degrees of freedom much smaller than its finite length [7,8]. Many natural signals are sparse or compressible in the sense that they have concise representations when expressed in the proper basis or transformation $\mathbf{\Psi}$. There is a duality between time and transformation domain (frequency) that expresses the idea that samples having a sparse representation in transformation domain (frequency) must be spread out in the domain in which they are acquired (time). We consider $\mathbf{x}=\boldsymbol{\psi}\mathbf{C}$ where $\mathbf{x}$ is an image with size (W,H) pixels represented as a vector, with $N=WH$, $\boldsymbol{\psi}$ is a matrix of size (N,N) and $\mathbf{C}$ is a vector of size $(N,1)$. The small coefficients C_i may be discard in the representation of $x(n)$ without much loss. The right choice of coefficients can lead both to the elimination of noise and to the compression of the image. Let $\mathbf{y}$ be a vector of size $(N,1)$ with S random chosen elements of $\mathbf{x}$ in the set $\mathbf{S}$ and the rest zero elements and $\mathbf{\Phi}$ a matrix of size (N,N) with rows of the inverse of matrix $\boldsymbol{\psi}$ selected by elements in set $\mathbf{S}$. The coefficients can be chosen using the compressive sensing principle. The coefficients are computing as $\mathbf{C_1}=\mathbf{\Phi y}$ and then the signal is reconstructed as $\mathbf{x}_1=\mathbf{\Psi C}_1$. The vector $\mathbf{y}$ will be selected so that $\|\mathbf{x}_1-\mathbf{x}\|$ to be minimum. The vector $\mathbf{y}$ can be chosen using random sampling consensus (CS-RANSAC) algorithm that works well in presence of relatively large number of disrupted pixels that will be considered as outliers. The other pixels, undisturbed or disturbed by weak noise will be considered as inliers and will be selected by CS-RANSAC algorithm as consensus set. The consensus set is the vector $\mathbf{y}$ and will be used in compressive sensing reconstruction as explain above.

The paper analyzes a noise elimination algorithm (CS-RANSAC) based on compressing sense and RANSAC combined with the DCT transform [8–10]. It is focused on the implementation on microcontrollers (for the IoT applications) and performance analysis (execution time and precision) to reduce the calculation time. The following sections will describe the algorithm in detail, its performance with infinite precision, the implementation of proposed algorithm using microcontrollers and the performances obtained using finite precision. To improve the execution

 doi:10.20944/preprints202504.0438.v1

time, a modified algorithm that involve DCT denoising for image's regions less affected by noise and CS-RANSAC algorithm for the image's regions more affected by noise can be used. To do this, a simplified algorithm to estimate the noise power can be used [18]. Finally, the results are shown and commented. The conclusion is that such non-filtering denoising techniques can be applied with good performance to eliminate or reduce the noise in images, and to compress the image, and suitable for IoT applications.

2. The Algorithm Description

The above-mentioned algorithm is based on sparsity of discrete cosine transform.

The CS-RANSAC algorithm is used to choose the non-noisy pixels. The DCT coefficients are determined using the compressive sensing method. The image to be processed is divided into blocks of smaller size (B,B) (chosen according to the size of the two-dimensional DCT transform) and each block is processed separately. The two-dimensional DCT transforms (direct and inverse) will be calculated as matrix multiplications, considering that the block to be processed is transformed into a column vector, $\mathbf{x}_b$ of dimensions $(N,1)$ with $N=B^2$ which will be multiplied by a matrix $\mathbf{W}$ of dimensions (B^2,B^2) determined by the DCT transformation matrix $\mathbf{T}$ with $\mathbf{W}=\mathbf{T}\otimes\mathbf{T}$ where $\otimes$ is the Kronecker product. The DCT transform will be $\mathbf{X}=\mathbf{W}\mathbf{x}$ and the inverse DCT transform is determined as $\mathbf{x}=\mathbf{W}^{-1}\mathbf{X}=\mathbf{W}^{T}\mathbf{X}$. For each of the blocks, a subset $\mathbf{y}_b$ of the set $\mathbf{x}_b$ with $S<B^2$ pixels is randomly chosen. A matrix $\mathbf{A}$ is then determined which contains the lines of transpose of $\mathbf{W}$ corresponding to the indices of the elements chosen from the $\mathbf{x}_b$. Using the matrix, $\mathbf{A}$ and the vector $\mathbf{y}_b$, the first k coefficients of the two-dimensional DCT transform are determined. Using these coefficients, the $\mathbf{x}_{rb}$ vector is reconstructed and the error between $\mathbf{x}_b$ and $\mathbf{x}_{rb}$ is determined. These steps are repeated until the error is small enough or the number of iterations exceeds a maximum imposed number. The last determined DCT coefficients will be the coefficients used to restore the pixels from the processed block.

The algorithm uses two functions:

- the *CSrec* function which receives as parameters the set of randomly chosen elements for determining non-noisy pixels, the modified transformation matrix and the number of DCT coefficients (sparsity factor) and returns the DCT coefficients that will be used to reconstruct the elements in the block.[9]
- the *pseudoinv* function, which calculates the inverse of a matrix using an iterative method [17].

The algorithm is designed so that it can be implemented as efficiently as possible (both on a computer with infinite precision and on a microcontroller with fewer resources and lower precision). The critical elements for a microcontroller implementation are matrix multiplication and calculation precision, especially when implementing the inverse matrix function. These aspects will be discussed in the microcontrollers' implementation section. The algorithm is illustrated in Figure 1. It was implemented in MATLAB (on a computer with an Intel I5-10210U processor at 1.60GHz, 4 cores, 6MB cache memory, 16 GB memory RAM and operating system Windows 10 Pro 64-bit) in order to validate the its functionality and to evaluate the performances obtained with infinite precision. Two implementations in C language were made using the Visual DSP++ development environment for Blackfin microcontrollers and a Maxim Eclipse SDK for MAX78000 microcontroller, including ARM , RISC V and CNN cores, with 32 bits finite precision fixed point.

 doi:10.20944/preprints202504.0438.v1

$\mathbf{D} = \Phi, D = 0, N = 0$ /* define a set $\mathbf{D}$ with D elements chosen from input vector $\mathbf{x}_b$ */
while($D < T$ and $n < N_{\max}$) /* T is the maximum number of elements in $\mathbf{D}$ */
/* n index of current iteration, $N_{\max}$ maximum number of iterations */
$n = n + 1$
randomly choose $\mathbf{S}$ with S numbers from 0 to $N-1$ /* define set $\mathbf{S}$ with S elements */
$\mathbf{A} = \mathbf{W}^{-1}$ |null elements on row $i \notin \mathbf{S}$ /* $\mathbf{A}$ is DCT matrix with rows index in set $\mathbf{S}$ */
/* all rows with index in $\mathbf{S}$ are zero */
$\mathbf{y}_b = \mathbf{x}_b$ |null elements of index $i \notin \mathbf{S}$ /* $\mathbf{y}_b$ is $\mathbf{x}_b$ vector with element index in set $\mathbf{S}$ */
/* all elements of $\mathbf{y}_b$ with index in $\mathbf{S}$ are zero */
$\mathbf{X}_b = CSrec(\mathbf{y}_b, \mathbf{A}, k)$ /* choose k DCT coefficients from observation $\mathbf{y}_b$ */
/* that reconstructs better the vector $\mathbf{x}_b$ */
$\mathbf{x}_{rb} = \mathbf{W}^{-1}\mathbf{X}_b$ /* $\mathbf{x}_{rb}$ is the reconstructed input vector */
$\mathbf{D} = \{$elements i of $\mathbf{x}_b$ with $|x_b(i) - x_{rb}(i)| < d\}$ /* update set $\mathbf{D}$ with the input vector elements */
/* that respect the distance d to reconstructed element */
cardD = number of elements $\in \mathbf{D}$ /* update number of elements in $\mathbf{D}$ */
end while
$\mathbf{A} = \mathbf{W}^{-1}$ |null elements on row $i \notin \mathbf{D}$ /* $\mathbf{A}$ is DCT matrix with rows index in set $\mathbf{S}$ */
/* all rows with index in $\mathbf{S}$ are zero */
$\mathbf{y} = \mathbf{x}_b$ |null elements of index $i \notin \mathbf{D}$ /* $\mathbf{y}$ is $\mathbf{x}_b$ vector with element index in set $\mathbf{S}$ */
/* all elements of $\mathbf{y}_b$ with index in $\mathbf{S}$ are zero */
$\mathbf{X} = CSrec(\mathbf{y}, \mathbf{A}, k)$ /* choose k DCT coefficients from $\mathbf{y}$ */
/* that reconstructs better the vector $\mathbf{x}_b$ */
$\mathbf{x} = \mathbf{W}^{-1}\mathbf{X}$ /* calculate reconstructed input /*

a) CS-RANSAC algorithm

function $\mathbf{X}_b = CSrec(\mathbf{y}, \mathbf{A}, k)$ /* determine the best k DCT coefficients /*
/* to reconstruct the input (remove noise and compress it) */
$\mathrm{K} = \Phi, \mathbf{e} = \mathbf{y}$ /* define a set K with the indexes of maximum values of $|\mathbf{A}^T\mathbf{e}|$ */
/* $\mathbf{e}$ is the error between observations $\mathbf{y}$ and reconstructed */
/* observation $\mathbf{y}_{kb}$ */
for $i = 1$ to k
$p = \arg\max(|\mathbf{A}^T\mathbf{e}|)$ /* p is the index of maximum element of $|\mathbf{A}^T\mathbf{e}|$ */
$\mathrm{K} = \mathrm{K} \cup p$ /* add p in the set K */
$\mathbf{A}_k = \mathbf{A}$ |null elements on column $j \notin \mathrm{K}$ /* $\mathbf{A}_k$ is matrix with matrix $\mathbf{A}$ columns indexes in K */
$\mathbf{X}_{kb} = pseudoinv(\mathbf{A}_k^T\mathbf{A}_k)(\mathbf{A}_k^T\mathbf{y})$ /* calculate inverse of $(\mathbf{A}_k^T\mathbf{A}_k)(\mathbf{A}_k^T\mathbf{y})$ */
$\mathbf{y}_{kb} = \mathbf{A}_k\mathbf{X}_{kb}$ /* determine reconstructed vector $\mathbf{y}_{kb}$ */
$\mathbf{e} = \mathbf{y} - \mathbf{y}_{kb}$ /* error */
end for
$\mathbf{X}_b = \mathbf{X}_{kb}$ |null elements of index $i \notin \mathrm{K}$ /* determine the best k DCT coefficients $\mathbf{X}_b$ */
/* as elements of $\mathbf{X}_{kb}$ with index in K */
end function

b) Compressive sensing function

 doi:10.20944/preprints202504.0438.v1

$function\ \mathbf{B} = pseudoinv(\mathbf{A})$
$\alpha = 1/\max element\ of\ (\mathbf{A}\mathbf{A}^T)$
$\mathbf{A0} = \alpha\mathbf{A}^T$ /* initial inverse matrix */
$while(err < \varepsilon\ and\ iter < NR_{\max})$ /*update inverse matrix until */
/*the error is smaller than ε*/
/*or themaximum number of iterations
/*iter exceeds NR_{max} */
$\mathbf{A}1 = \mathbf{A0} + (\mathbf{I} - \mathbf{A0}.\mathbf{A}).\mathbf{A0}$ /*determine inverse A1*/
$err = \max element\ of\ \{(\mathbf{A}.\mathbf{A1}.\mathbf{A} - \mathbf{A}), (\mathbf{A1}.\mathbf{A}.\mathbf{A1} - \mathbf{A1})\}$ /*compute error*/
$iter = iter + 1$ /*increments the number of iterations*/
$\mathbf{A0} = \mathbf{A1}$ /*update A0 */
$end\ while$
$\mathbf{B} = \mathbf{A1}$ /*save inverse*/
$end\ function$

c) Pseudo-inverse function

Figure 1. The CS-RANSAC algorithm.

The following sections describe the performance obtained for the MATLAB implementation, the specific implementation for microcontrollers and compare the performance obtained in the two implementations.

3. The Algorithm Performance Evaluation

This section describes the performance of above described algorithm in terms of quality of the reconstructed image. Selected parameters of the algorithm are: $B = 4, S = 15, k = 3$. The image size is set to 512 pixels in height and 512 pixels in width [16] (for both type of implementations – MATLAB and microcontrollers). Gaussian noise, salt, and pepper noise (impulsive) and multiplicative noise (speckle) were successively added to the test images. Also, the image was blurred.

The performance of the algorithm was evaluated using peak signal to noise ratio $PSNR = 10\log_{10}\frac{255^2}{\frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}[I(i,j) - I_r(i,j)]^2}$ with I the noisy image, and I_r the reconstructed image, both of dimensions (N, M), and structural similarity index measure $SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$ with μ_x, μ_y, σ_x^2, σ_y^2, and σ_{xy} are the mean, the variance and the covariance of pixels in windows x and y. The constant coefficients C_1 and C_2 are used to stabilize the division with weak denominator. The $SSIM$ quantifies image quality degradation caused by data compression or by losses in data transmission. Unlike PSNR, SSIM is based on visible structures in the image and perhaps represents a more reliable indicator of image quality degradation. PSNR is an alternative measurement of quality of reconstructed image [15]. The Figures 2–4 illustrate the performance of denoising algorithm with infinite precision (MATLAB implementation – 64 bits floating point). The noise is mixed noise. The results show an image quality improvement in both PSNR (up to 6 dB) and SSIM (up to 3 times) when using the CS-RANSAC algorithm.

 doi:10.20944/preprints202504.0438.v1

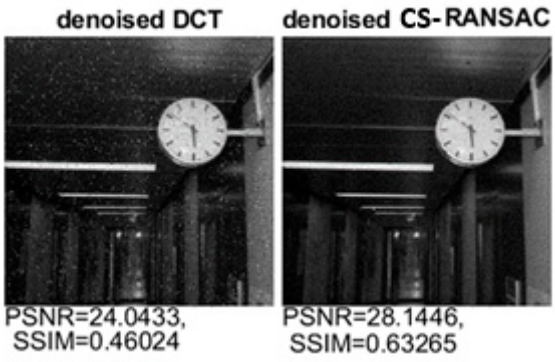


Figure 2. Performance evaluation for low noise (noised image PSNR 20 dB, SSIM 0.36). Mixed noise : gaussian variance=0.001; salt and pepper density 2%; blur kernel window length =3; speckle variance=0.001.

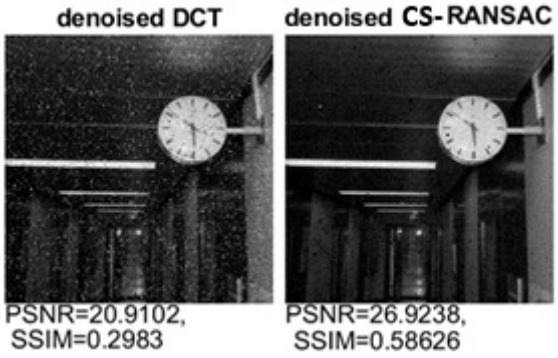


Figure 3. Performance evaluation for large noise (noised image PSNR 17 dB, SSIM 0.22). Mixed noise : gaussian variance=0.001; salt and pepper density 5%; blur kernel window length =3; speckle variance=0.001.

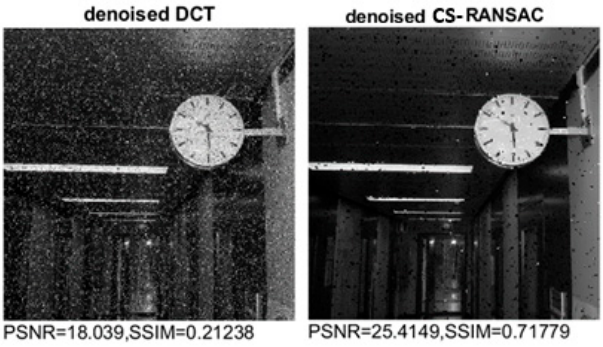


Figure 4. Performance evaluation for impulsive noise. Noise density 10% (noised image PSNR 14 dB, SSIM 0.15).

The Table 1 summarizes the performance of CS-RANSAC algorithm comparing with DCT denoising at the same level of sparsity. One can observe that the performance is netter for CS-RANSAC algorithm both for PSNR and SSIM criteria.

Table 1. DCT denoising vs CS-RANSAC denoising.

Noised Image		DCT Reconstructed Image		CS-RANSAC Reconstructed Image	
PSNR	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**
20.48	0.36	24.04	0.46	28.14	0.63
17.14	0.21	20.91	0.29	26.92	0.58
14.29	0.15	18.03	0.21	25.41	0.71

 doi:10.20944/preprints202504.0438.v1

A more detailed performance evaluation is given in [9]. In this paper, we evaluate the algorithm performance, in various implementation, to give a performance comparison between these implementations and to prove the possibility to implement, with good performance, relative complex image processing algorithms using microcontrollers in IoT applications.

4. The Microcontrollers' Implementations

Two 32 bits fixed point microcontroller (e.g. Blackfin BF5xx, and MAX7800x, Analog Device) implementations are proposed in this section.

The Blackfin architecture is designed for multimedia applications, the accessible memory is up to hundreds of Mbytes and the processor clock frequency is up to 750 MHz The instruction set is powerful (arithmetic instructions, multiplications with accumulation, dual and quad instructions, hardware loops, multifunction instructions) [13,14].

The MAX7800x chip is a dual core ultra-low power microcontroller with an ARM Cortex M4 processor with FPU up to 100 MHz with 16KB instruction cache, 512KB flash memory and 128KB SRAM, and a RISC-V Coprocessor up to 60MHz for digital signal processing instructions. There are many interfaces (general purpose IO pins – GPIO, serial ports , analog to digital convertor (10 bit, 8 channels), neural network accelerator optimized for deep convolutional neural networks (442k 8-bit weight capacity, network depth up to 64 layers with up to 1024 channels per layer), power management for battery operations, real time clock, timers, AES 128/192/256 and CRC hardware acceleration engine. The ARM Cortex-M4 with FPU processor CM4 is well suited for the neural networks system control and combines high-efficiency signal processing functionality with low energy consumption. The 32-bit RISC-V coprocessor is dedicated for ultra-low power consumption signal processing. The instructions set include: four parallel 8-bit additions/subtractions, floating point single precision operations, two parallel 16-bit additions/subtractions, two parallel MACs, 32- or 64-bit accumulate, signed, unsigned, data with or without saturation. A Convolutional Neural Network (CNN) unit is included in MAX7800x chip.

A more detailed architecture description of MAX 7800x and how the proposed implementation uses the CNN accelerator, and ARM and RISC V cores is shown in a next section.

The above presented algorithm was written in C programming language, using as integrated development environment Visual DSP ++ 5.1 and Maxim Eclipse SDK. The code was automatically optimized for speed (hardware loops, interprocedural analysis) [12]. Some adaptations of the algorithm were made to reduce the execution time: for S = 15, the method of determining the set S has been changed (considering that the number of possible combinations is 16, a combination will be chosen randomly and the number of iterations in the CS-RANSAC algorithm is limited to a maximum of 16 iterations) and VSDP++ library functions were used for all matrix and vector operations [11,12]: matrix multiplication - *matmmltf,* matrix addition and subtraction - *matsadd, matssub,* matrix transpose- *transpm,* maximum and minimum element in a vector- *vecmax, vecmin,* location of maximum and minimum element in a vector, *vecmaxloc, vecminloc*. The code uses a 32 bits representation for floating point algorithm variables and computations (multiplications and additions) [11]. This approach will cause a slight decrease in precision and therefore the quality of the reconstructed image, but the use of a 32-bits fixed-point representation would excessively increase the execution time. The use of fixed-point representation keeps the processing time at reasonable values with an acceptable decrease in performance. The execution time was measured, in processor cycles, using the IDEs' code profiler.

5. The Performance Using 32-Bits Fixed Point Microcontrollers

This section describes the results obtained using the 32 bits processor. The execution time and the effect of finite precision is shown in the Figures 5–7. One can observe, in these figures, that the performance is good. For low and medium levels of mixed noise, the CS-RANSAC algorithm has a PSNR greater with up to 4 dB and a SSIM greater up to 80%. The SSIM obtained with CS-RANSAC

 doi:10.20944/preprints202504.0438.v1

is better than that of DCT even the differences in PSNR are not so high for high level of mixed noise. The CS-RANSAC algorithm responds better for impulsive noise, as is shown in Figure 7. Table 2 summarizes the performances and Table 3 compares the implementation in MATLAB with Blackfin implementation.

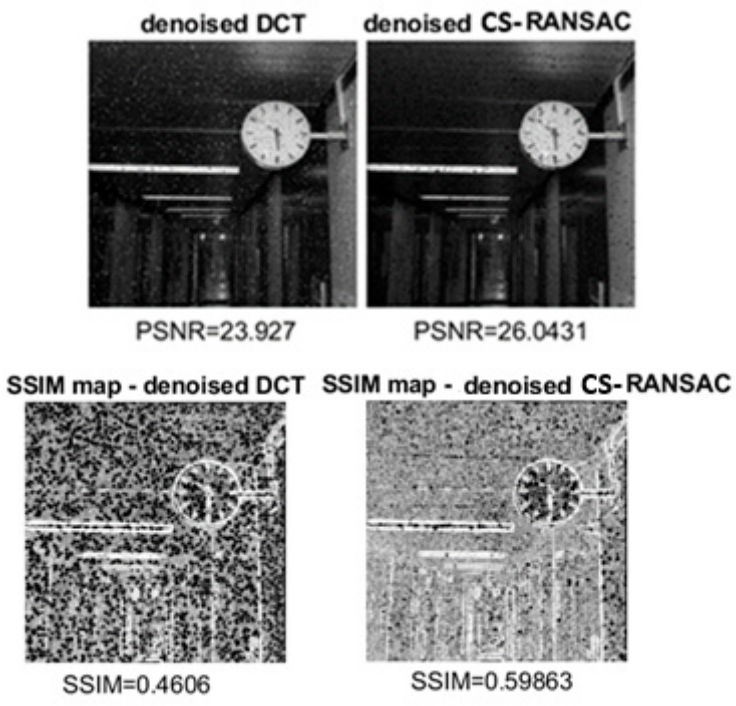


Figure 5. Performance evaluation for low noise (noised image PSNR 20 dB, SSIM 0.36). Mixed noise: gaussian variance=0.001; salt and pepper density 2%; blur kernel window length =3; speckle variance=0.001.

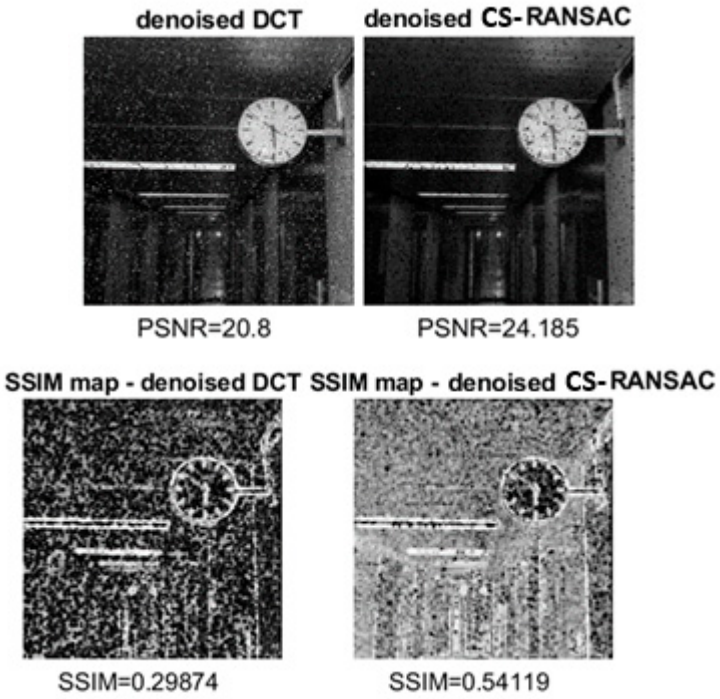


Figure 6. Performance evaluation for large noise (noised image PSNR 17 dB, SSIM 0.22). Mixed noise: gaussian variance=0.001; salt and pepper density 5%; blur kernel window length =3; speckle variance=0.001.

 doi:10.20944/preprints202504.0438.v1

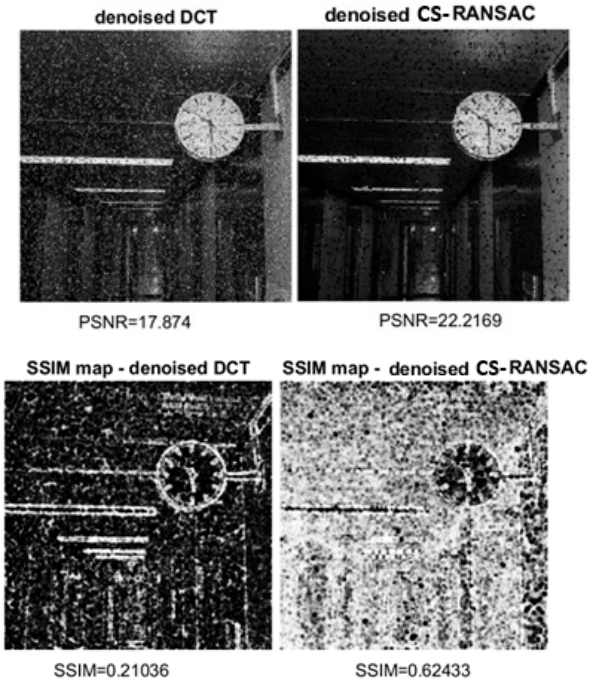


Figure 7. Performance evaluation for impulsive noise (noised image PSNR 14 dB, SSIM 0.15). Noise density 10%.

Table 4 illustrated the execution time considering a Blackfin processor at 750MHz. An improved implementation using MAX7800x (with ARM core at 100MHz, RISC V core at 60 MHz and CNN accelerator with 64 cores at 50 MHz) will be described in next sections.

Table 2. Performance comparison (32-bits fixed point implementation).

Noised Image		DCT Reconstructed Image		CS-RANSAC Reconstructed Image		Execution Time CS-RANSAC Blackfin	
PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	Cycles	Time
20.48	0.36	23.92	0.46	26.04	0.59	4,675,862,063	6.23
17.14	0.21	20.8	0.29	24.18	0.54	5,662,802,735	7.55
14.29	0.15	17.87	0.21	22.21	0.62	8,728,624,005	11.64
15.04	0.18	18.04	0.25	19.47	0.42	7,584,207,993	10.11
19.6	0.38	22.45	0.46	23.45	0.57	4,520,337,537	6.03

Table 3. Performance comparison (32-bits fixed point vs MATLAB implementation).

Noised Image		DCT Reconstructed Image (Fixed Point 32 Bits)		CS-RANSAC Reconstructed Image (Fixed Point 32 Bits)		DCT Reconstructed Image (MATLAB)		CS-RANSAC Reconstructed Image (MATLAB)	
PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
20.48	0.36	23.92	0.46	26.04	0.59	24.04	0.46	28.14	0.63
17.14	0.21	20.8	0.29	24.18	0.54	20.91	0.29	26.92	0.58
14.29	0.15	17.87	0.21	22.21	0.62	18.03	0.21	25.41	0.71

Table 4. Execution time – seconds (32 bits fixed point vs MATLAB implementation).

Noised Image		Execution Time CS-RANSAC 32 Bits Fixed Point		Execution Time CS-RANSAC MATLAB
PSNR	SSIM	Cycles	Time	Time
20.48	0.36	4,675,862,063	6.23	2.49
17.14	0.21	5,662,802,735	7.55	4.79

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 April 2025 doi:10.20944/preprints202504.0438.v1

One can observe, for medium to high noise, that the execution time is reasonable (about seconds). The execution time can be decrease by using a dual-core Blackfin processor. An implementation based on an accelerator for convolutional neural networks (CNN) is possible by implementing matrix multiplication using a 1x1 convolution performed with CNN.

6. The Improving of Processing Time Using CNN Accelerator

The multiplication of two matrices $\mathbf{A}=\left[a_{ij}\right]$ and $\mathbf{B}=\left[b_{ij}\right]$ with the result $\mathbf{C}=\mathbf{AB}=\left[c_{ij}\right]$ and $i,j=1..N$ can be performed using a fully interconnected layer as in Figure 8:

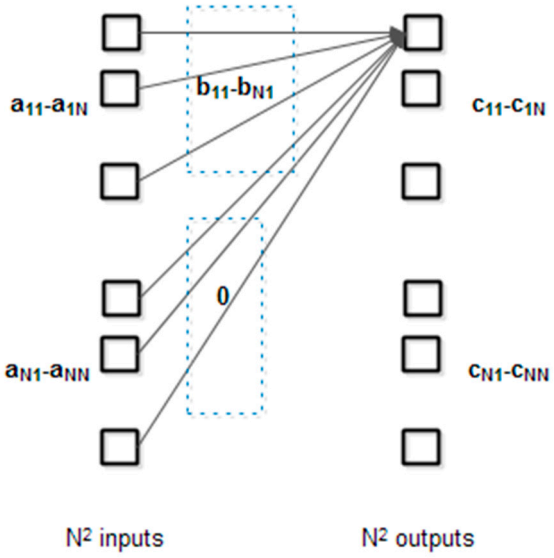


Figure 8. Neural network fully interconnected layer used for matrix multiplication.

The input layer consists of each row in matrix **A** and the output layer contains the elements of product matrix. For each input elements, the weights are the corresponding elements of columns in matrix **B** or zero elements. For clarity, only the weights for one output elements are shown.

Figure 9 details the weights for a simple example ($N=2$):

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \text{ and } \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \text{ then}$$

$$\mathbf{C}=\mathbf{AB}=\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}=\begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix}$$

For more clarity, the weights have shown individually for each output element.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 April 2025 doi:10.20944/preprints202504.0438.v1

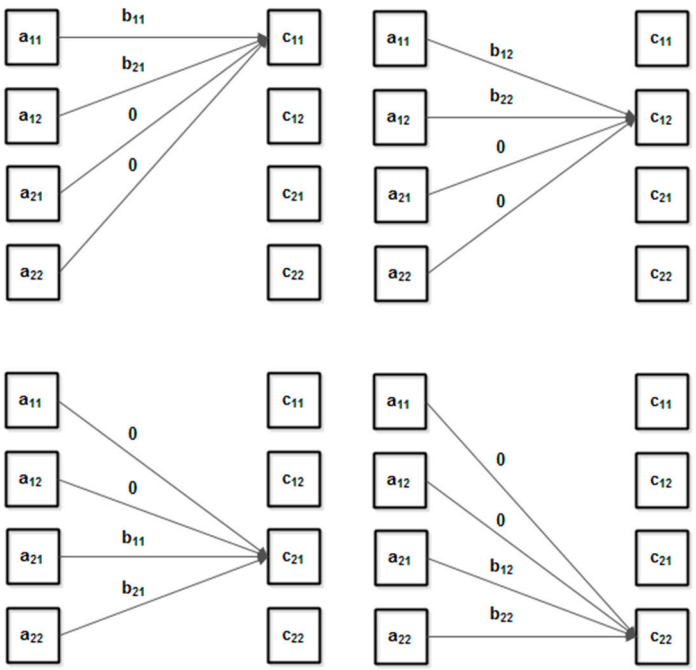


Figure 9. Neural network fully interconnected layer used for matrix multiplication (detailed example for $N = 2$).

The implementation of fully interconnected layer can be done in CNN by enabling the flatten mode (this mode supports a series of 1×1 convolution emulating a fully interconnected network with up to 1024 inputs).

The matrix multiplication (fixed point) is shown in Figure 10.

Inputs:

- matrices order N

- matrices values a_{ij}, b_{ij} *with* $i, j = 1..N$

Outputs:

- product of given matrices: c_{ij} *with* $i, j = 1..N$

1. Define a neural network layer – full interconnected with N^2 *inputs* a_{ij} *(read on rows),* N^2

outputs and weights initialized with b_{ji} *for input* a_{ij} *and output* c_{ij} *and*

with 0 otherwise

2. Enable mode flattened for CNN

3. Load CNN memory with the defined inputs and weights

4. Start CNN

5. Wait for CNN to complete the computation

6. Retrieve the results c_{ij} *with* $i, j = 1..N$

Figure 10. Matrix multiplication CNN-based algorithm (fixed point).

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 April 2025 doi:10.20944/preprints202504.0438.v1

Using the above algorithm, a speedup about 30 times can be achieved for integer matrix multiplication, comparing with the implementation on a 32 bits fixed-point microcontroller. This can be useful for algorithms based on matrix computation that does not require large dynamic range.

In common CNNs the values of neural network layers and weights are represented in fixed point with 8 bits. There are certain applications that require more precision. For example, CS-RANSAC algorithm, presented in previous sections, requires higher precision due the DCT coefficients of lower order.

We proposed an approach that make possible the matrix multiplication with increased precision, considering a floating-point representation of matrix elements.

We assume that the values of matrix elements are $a_{ij} = A_{ij} 2^{ea_{ij}}$, $b_{ij} = B_{ij} 2^{eb_{ij}}$ and $c_{ij} = C_{ij} 2^{ec_{ij}}$ with A_{ij}, B_{ij}, C_{ij} - mantises and $ea_{ij}, eb_{ij}, ec_{ij}$ - exponents represented as fixed point integers with 8 bits. The output elements are $c_{ij} = \sum_{k=1}^{N} A_{ik} 2^{ea_{ik}} B_{kj} 2^{eb_{kj}} = \sum_{k=1}^{N} A_{ik} B_{kj} 2^{ea_{ik}+eb_{kj}}$. The term $O_{ikj} = A_{ik} B_{kj}$ will be computed using a full interconnected layer as it has shown previously (with a slight modification of weights – see Figure 11.) and the term $ea_{ik} + eb_{kj}$ will be computed using the element-wise function (the element-wise function must be enabled in CNN and the addition function must be selected).

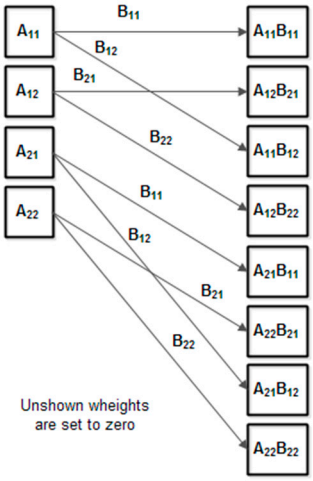


Figure 11. The full interconnected layer for floating point implementation ($N = 2$).

Then the maximum exponent is calculated as $E_{ij,\max} = \max_{k=1..N}(ea_{ik} + eb_{kj})$ and all the terms O_{ikj} will multiplied with $2^{ea_{ik}+eb_{kj}-E_{ij,\max}}$ in a second full interconnected layer. The results $X_{ikj} = O_{ikj} 2^{ea_{ik}+eb_{kj}-E_{ij,\max}}$ are summed using the element wise CNN features. The sum is calculated iteratively using the element wise addition in $\log_2 N^2$ steps as in Figure 12.

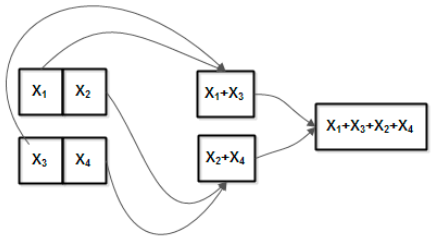


Figure 12. Element wise addition (example for $N = 4$).

Finally, in a third full interconnected layer the elements of matrix products $c_{ij} = 2^{E_{ij,\max}} X_{ikj}$ are calculated. We assume that in an image processing one matrix (image to process) has sub-unitary

 doi:10.20944/preprints202504.0438.v1

mantises and $ea_{ij} = 0, i, j = 1..N$. The other matrix's exponents and mantises are constant, therefore $E_{ij,\max}$ and $2^{ea_{ik}+eb_{kj}-E_{ij,\max}}$ can be passed as parameters in the matrix multiplication function. The complete algorithm is illustrated in Figure 13:

Inputs:

- matrices mantises and exponents A_{ij}, (B_{ij}, eb_{ij}), $i, j = 1..N$

- the terms $2^{eb_{kj}-E_{ij,\max}}$ and $E_{ij,\max} = \max_{k=1..N}(eb_{kj})$, $i, j = 1..N$

Outputs:

- matrix product elements c_{ij}, $i, j = 1..N$

1. Define a neural network layer – full interconnected with N^2 *inputs* A_{ij} *(read on rows),* N^3 *outputs and weights initialized with* B_{kj} *for input* A_{ik} *and output* (i,k,j),

$$0 \text{ otherwise}$$

2. Enable mode flattened for CNN

3. Load CNN memory with the defined inputs and weights

4. Start CNN - compute $O_{ikj} = A_{ik} B_{kj}$, $i, k, j = 1..N$

5.. Load CNN memory with O_{ikj} *and* $2^{eb_{kj}-E_{ij,\max}}$

6.. Start CNN - compute $X_{ikj} = O_{ikj} 2^{eb_{kj}-E_{ij,\max}}$, $i, k, j = 1..N$

7. Partitioning the elements of X_{ikj} in N^2 partitions $P_{ij} = \{X_{ikj}\} | k = 1..N$

8. $M = N^2$

9. Load CNN memory with $P_{ij}^{(1)} = P_{ij} | k = 1..(M/2)$, $P_{ij}^{(2)} = P_{ij} | k = (M/2+1)..M$,

$$i, j = 1..N$$

10. Enable mode element wise with addition function

11. For $r = 1..\log_2 N^2$

12. $M = M/2$

13. Start CNN - compute sum $P_{ij} = P_{ij}^{(1)} + P_{ij}^{(2)}$

14. End For

15. Enable mode flattened for CNN

16. Load CNN memory with $2^{E_{ij,\max}}$ *and* P_{ij}

17. Start CNN - compute $c_{ij} = P_{ij} 2^{E_{ij,\max}}$

Figure 13. Matrix multiplication CNN-based algorithm (floating point).

 doi:10.20944/preprints202504.0438.v1

The algorithm illustrated in Figure 13 can be efficiently implemented using the MAX78000 chip [19,20]. The block diagram of MAX7800 is illustrated in Figure 14. The CNN accelerator consists of 64 parallel processors with 512KB of SRAM-based storage. Each processor includes a pooling unit and a convolutional engine with dedicated weight memory. Four processors share one data memory. These are further organized into groups of 16 processors that share common controls. A group of 16 processors operates as a slave to another group or independently. Data is read from SRAM associated with each processor and written to any data memory located within the accelerator. Any given processor has visibility of its dedicated weight memory and to the data memory instance it shares with the three others.

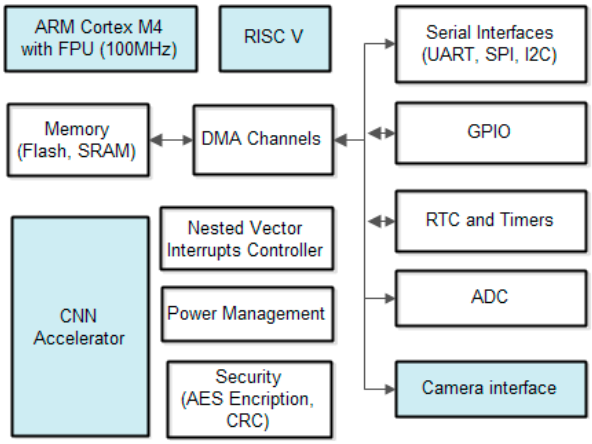


Figure 14. The MAX7800x general architecture.

In general, an algorithm (or working task) with M instructions with t_1 average execution time per instruction can be divided in two parts: fM - running using one processor with t_2 average execution time per instruction and $(1-f)M$ - running using N processors with t_3 average execution time per instruction, with $f<1$. The speedup is calculated as $s=\frac{Mt_1}{fMt_2+\frac{(1-f)Mt_3}{N}}=\frac{t_1}{ft_2+\frac{(1-f)t_3}{N}}$. Considering $t_2=rt_1$ and $t_3=qt_1$ with $r,q<1$ the speedup becomes $s=\frac{t_1}{frt_1+\frac{(1-f)qt_1}{N}}=\frac{1}{fr+\frac{(1-f)q}{N}}=\frac{N}{Nfr+(1-f)q}=\frac{N}{f(Nr-q)+q}$

All the computations involved in matrix processing (multiplication, addition) can be implemented using the CNN block in flattened or element wise modes. The above presented CS-RANSAC algorithm illustrated above was implemented using MAX78000 and its CNN accelerator. The numerical precision is similar with numerical precision obtained with previous implementation on Blackfin (the ARM and RISC cores in MAX78000 also use 32 bits fixed point representation).

In the speedup relation we set $N=64$ (the number of cores in CNN), $r=0.13$ (the ratio between ARM microcontroller speed and Blackfin microcontroller), $q=0.06$ (the ratio between CNN cores speed and Blackfin microcontroller), and $f=0.78$ (the algorithm code that not contain matrix operations that can be performed in CNN). With this value the theoretical speedup (between Blackfin implementation and MAX7800 implementation) is $s=9.84$. The effective speedup (obtained by counting processor cycles by the IDE code profiler) is lower due the data transfers performed using RISC V.

Figure 15 illustrated the execution time obtained with CNN implementation. In this case (software floating point implementation) the speedup obtained is about 7 times for the CS-RANSAC algorithm.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 7 April 2025 doi:10.20944/preprints202504.0438.v1

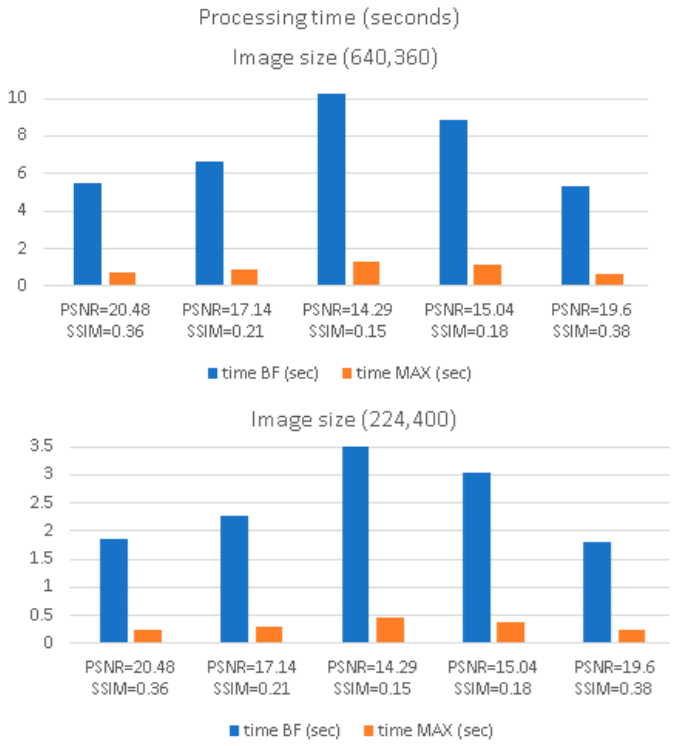
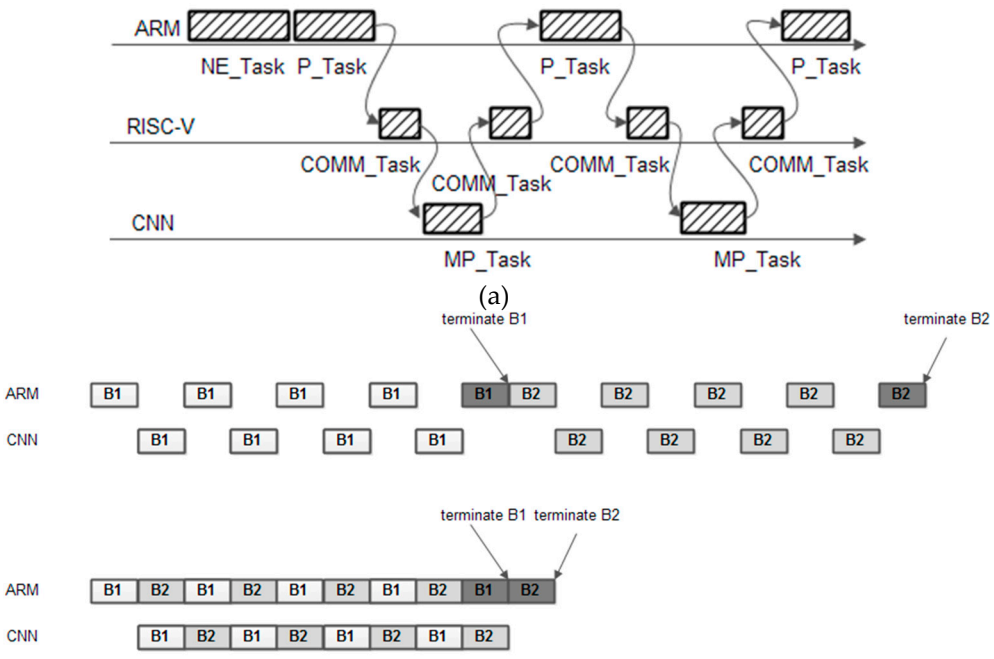


Figure 15. Processing time (various noised image sizes) for two 32-bits fixed point implementations (Blackfin and Max78000 CNN).

The computing time of the algorithm can be improved modifying the way to calculate DCT coefficients using original DCT or CS-RANSAC depends if the block is noisy or not, and using in parallel the ARM and CNN

For the first improving method, the original CS-RANSAC algorithm was combined with a noise estimator [21]. Each block is marked as light noised or heavy noised and is processed using simple DCT or CS-RANSAC, respectively. The noise estimator can be implemented in fixed point using in parallel the ARM and RISC-V microcontrollers in MC78000 chip. Figure 16(a) shows how the tasks for such implementation can be scheduled.



 doi:10.20944/preprints202504.0438.v1

(b)

Figure 16. (a) Tasks scheduling for the algorithm (b) algorithm improving using parallel processing (RISC V is not shown, for clarity).

The following tasks is defined: *NE_Task* - for noise estimate, *P_Task* – the processing task that implement all the processing for DCT and CS-RANSAC noise removal and compression algorithms excepts the matrix multiplications and matrix additions which are computed by the a matrix processing task - *MP_Task* and *COMM_task* – a communication task that transfer using DMA channels the information (matrix values) between CNN and ARM. The tasks : *NE_Task* and *P_Task* are running in ARM core, the task *COMM_task* is running in ARM coprocessor (that acts as a direct memory access - DMA controller). All the matrix manipulations are passed to CNN accelerator (programmed in flatten mode for matrix multiplications or element wise mode for matrix additions or subtractions) and are computed by *MP_Task*. All tasks are synchronized using global semaphores.

Depend on noise level, the execution time can be reduced as is illustrated in Figure 17 [22].

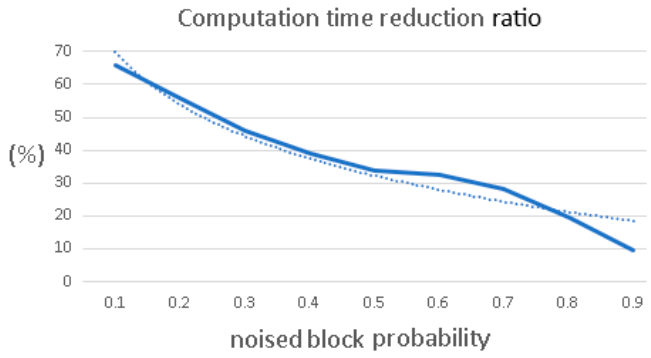


Figure 17. Computing time reduction (in dot line – trendline of ratio).

For an average noise probability of 50% one can observe that the computation time reduction ratio is about 35%. If the noise estimator is not used, the task *NE_Task* in the scheduling tasks from Figure 16 is removed.

The second method ensures halving of computation time. This goal is achieved by partitioning the computation in matrix operations (multiplications, addition)- performed in CNN and non-matrix operations (all the remaining computations) – performed in ARM. Two blocks are processing in parallel in ARM and CNN, alternatively, as it is shown in Figure 16 (b).

7. Conclusion

This paper focuses on the analysis of the possibility of accelerating the necessary processing in algorithms based on matrix operations. Accelerating these operations can be achieved using neural network processing units (NPUs) integrated into the architecture of today's high-performance microcontrollers. As an example, the paper presents implementations and performance analysis of an image compression and noise removal algorithm based on compressive sensing and CS-RANSAC.

This algorithm was validated as good algorithm in terms of noise removal and image compression using infinite precision implementation (e.g. MATLAB simulations). The main goal of this work is to evaluate if a microcontroller implementation is feasible in terms of processing accuracy and computation time to be used in IoT applications that involve hardware nodes with resources constrains.

The obtained results show that a good quality of the reconstructed image can be obtained for medium to high noise levels in a calculation time of the order of seconds or tenth of seconds.

Also, the paper proposes methods to improve the algorithm: (1) by selectively applying DCT or the CS-RANSAC to each block in the image (without degrading the quality of the image), and (2) be

 doi:10.20944/preprints202504.0438.v1

using in parallel the ARM microcontroller and CNN cores or using a dual core Blackfin microcontroller.

Author Contributions: Conceptualization, S.Z..; methodology, R.Z.; software, S.Z.; validation. R.Z.; formal analysis, R.Z.; investigation, S.Z.; resources, S.Z..; data curation, R.Z.; writing—original draft preparation, S.Z.; writing—review and editing, R.Z., S.Z..; visualization, R.Z.; funding acquisition, S.Z. All authors have read and agreed to the published version of the manuscript.

Funding: The research leading to these results received funding from the Research Centre on Systems Software and Networks in Telecommunication (CCSRST).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bartyzel K., "Adaptive Kuwahara filter", Signal Imag Video Process 10:663–670, 2016
2. Jiang J, Zhang L, Yang J, "Mixed Noise removal by weighted encoding with sparse nonlocal regularization", IEEE Trans Image Process 23(6):2651–2662, 2014
3. Strong, D.; Chan, T. (2003). "Edge-preserving and scale-dependent properties of total variation regularization"
4. Dabov, Kostadin; Foi, Alessandro; Katkovnik, Vladimir; Egiazarian, Karen (16 July 2007). "Image denoising by sparse 3D transform-domain collaborative filtering". IEEE Transactions on Image Processing. 16 (8): 2080–2095
5. Yang Y, Sun J, Li H, Xu Z, "ADMM-CSNEt: a deep learning approach for image compressive sensing", IEEE Trans Pattern Anal Mach Intell 42(3):521–538, 2020
6. Yu, Guoshen & Sapiro, Guillermo, "DCT Image Denoising: A Simple and Effective Image Denoising Algorithm", Image Processing Online. 1. 10.5201/ipol.2011.ys-dct.
7. Strutz, T., "Data Fitting and Uncertainty (A practical introduction to weighted least squares and beyond)", 2nd edition, Springer Vieweg. ISBN 978-3-658-11455-8, 2016
8. Raguram, Rahul & Frahm, Jan-Michael & Pollefeys, Marc, "A Comparative Analysis of RANSAC Techniques Leading to Adaptive Real-Time Random Sample Consensus", Proceedings of the 10th European Conference on Computer Vision Part II. 500-513, 2008
9. Stanković, I., Brajović, M., Lerga, J. et al. "Image denoising using RANSAC and compressive sensing", Multimedia Tools Appl 81, 44311–44333 (2022).
10. E. J. Candes and M. B. Wakin, "An Introduction to Compressive Sampling," IEEE Signal Processing Magazine, vol. 25, no. 2, pp. 21-30, March 2008,
11. Fast Floating-Point Arithmetic Emulation on Blackfin Processors. https://www.analog.com/media/en/technical-documentation/application-notes/ee.185.rev.4.08.07.pdf, Retrieved May 2024
12. VisualDSP++ 5.0 C/C++ Compiler and Library Manual for Blackfin Processors, https://www.analog.com/media/en/dsp-documentation/software-manuals/50_bf_cc_rtl_mn_rev_5.4.pdf, Retrieved May 2024
13. ADSP-BF533 Blackfin Processor Hardware Reference, https://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP- BF533_hwr_rev3.6.pdf, Retrived May 2024
14. ADSP-BF533 EZ-KIT Lite® Evaluation System Manual, https://www.analog.com/media/en/technical-documentation/user-guides/ADSP- BF533_ezkit_man_rev.3.2.pdf
15. A. Horé and D. Ziou, "Image Quality Metrics: PSNR vs. SSIM," 2010 20th International Conference on Pattern Recognition, Istanbul, Turkey, 2010, pp. 2366-2369,
16. https://webpages.tuni.fi/imaging/tampere17/tampere17_grayscale.zip, Retrieved May 2024
17. https://aalexan3.math.ncsu.edu/articles/mat-inv-rep.pdf, Retrieved May 2024
18. M.Ponomarenko, N.Gapon, V.Voronin, K. Egiazarian, "Blind estimation of white Gaussian noise variance in highly textured images", Image Processing: Algorithms and Systems XVI, 2018
19. https://www.analog.com/media/en/technical-documentation/data-sheets/MAX78000.pdf

 doi:10.20944/preprints202504.0438.v1

20. https://www.analog.com/media/en/technical-documentation/user-guides/max78000-user-guide.pdf
21. M.Ponomarenko, N.Gapon, V.Voronin, K. Egiazarian, "Blind estimation of white Gaussian noise variance in highly textured images", Image Processing: Algorithms and Systems XVI, 2018
22. S. Zoican, R. Zoican and D. Galatchi, "Image denoising algorithm for IoT based on compressive sensing principle and Blackfin microcontrollers," *2024 15th International Conference on Communications (COMM)*, Bucharest, Romania, 2024, pp. 1-4. https://doi.org/10.1109/COMM62355.2024.10741495