International Journal of Scientific Research in Computer Science, Engineering and Information Technology
ISSN : 2456-3307 (www.ijsrcseit.com)
doi : https://doi.org/10.32628/CSEIT217690

Implementation and Optimization of Image Processing on the Map of SABRE i.MX_6

Waheed Muhammad SANYA*[1], Gaurav BAJPAI[2], Haji Ali HAJI[3]

*[1]Department of Computer Science and Information Technology, the State University of Zanzibar, Zanzibar, Tanzania.

[2]Department of Computer Engineering and Software Engineering, University of Rwanda, Kigali, Rwanda.

[3]Department of Computer Science and Information Technology, the State University of Zanzibar, Zanzibar, Tanzania

Article Info

Volume 7, Issue 6

Page Number : 402-417

Publication Issue :

November-December-2021

Article History

Accepted : 15 Dec 2021

Published : 30 Dec 2021

ABSTRACT

Vision relieves humans to understand the environmental deviations over a period. These deviations are seen by capturing the images. The digital image plays a dynamic role in everyday life. One of the processes of optimizing the details of an image whilst removing the random noise is image denoising. It is a well-explored research topic in the field of image processing. In the past, the progress made in image denoising has advanced from the improved modeling of digital images. Hence, the major challenges of the image process denoising algorithm is to advance the visual appearance whilst preserving the other details of the real image. Significant research today focuses on wavelet-based denoising methods. This research paper presents a new approach to understand the Sobel imaging process algorithm on the Linux platform and develop an effective algorithm by using different optimization techniques on SABRE i.MX_6. Our work concentrated more on the image process algorithm optimization. By using the OpenCV environment, this paper is intended to simulate a Salt and Pepper noisy phenomenon and remove the noisy pixels by using Median Filter Algorithm. The Sobel convolution method included and used in the design of a Sobel Filter and then process the image following the median filter, to achieve an effective edge detection result. Finally, this paper optimizes the algorithm on SABRE i.MX_6 Linux environment. By using algorithmic optimization (lower complexity algorithm in the mathematical sense, using appropriate data structures), optimization for RISC (loop unrolling) processors, including optimization for efficient use of hardware resources (access to data, cache management and multi-thread), this paper analyzed the different response parameters of the system with varied inputs, different compiler options (O1, O2, or O3), and different doping degrees. The proposed denoising algorithm shows the meaningful addition of the visual quality of the images and the algorithmic optimization assessment.

Keywords : Salt & Pepper Noisy, Algorithm optimization, OpenCV, Median Filter, Sobel Filter

I. INTRODUCTION

1.1 Study Background

Denoising plays an additional important role in modern image processing and analysis. Image denoising approaches are with an aim to preserve the details of an image as well as to remove the random noise to the degree that is possible. It is one of the most used concepts in most image-processing applications. A digital image is subject to a variety of noise that affects the quality of an image. This noise is salt and pepper that is generated by an image sensor defect. Salt and pepper noise is mainly caused by defective pixels in camera sensors that are frequently found in digital transmission. Once an image is corrupted by salt and pepper noise, the pixel values may have any random value inside the maximum as well as minimum values in the dynamic range [1]. In signal processing, it is often desirable to perform some notable noise reduction on an image or signal. The median filter is a nonlinear digital filtering technique that is often used to remove noise. The removal of salt and pepper noise is normally achieved by using median-type filters [2].

The Sobel operator, also sometimes referred to as the Sobel Filter, is used in image processing and computer vision, particularly in edge detection algorithms. This creates an image that emphasizes the edges and transitions. This is known as Loop unrolling or Loop unwinding. It is a Loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size (space-time trade-off). This transformation is undertaken manually by the programmer or by an optimizing compiler. On a single processor, multithreading is generally implemented by time-division multiplexing (as in multitasking). Here the processor (CPU) alternates between different software threads.

1.2 Study Objective

The objective of this research is to implement by optimizing a chain of image processing on the map SABRE i.MX_6 (ARM_Cortex-A9) in implementation.

1.2.1 Specific Objectives

To design algorithmic optimization for image processing (lower complexity algorithm in the mathematical sense, appropriate data structures).
Optimization for RISC (loop unrolling) processors.
Optimization for efficient use of hardware resources (access to data, cache management, multi-thread).

1.2.2 Expected Outputs

The main outputs of this research will be:

- Mathematic moulding and realization
- Optimization.
- Analysis for board and statistics.

1.2.3 Experimental Environment:

SABRE IMX6 (ARM_Cortex-A9)

TABLE 1: SABRE IMX6

Memory	1GB
IDE	8GB
CPU	ARM_Cortex-A9

Memory	6GB
IDE	136GB
CPU	INTEL

Table 2: Info Linux based on HP workstation

Table 1 and Table 2 detail the experimental environment of this research by using SABRE IMX6 as well as Linux based on a HP workstation.

The rest of the paper is organized as follows: The Related Works are presented in Section 2, which include the techniques for Median filter, Adaptive Median Filter and Non-Local Means Filtering. In Section 3, the System Architecture Modelling is described and Section 4 present the results and discussion of the simulation. Section 5 presents the conclusion.

II. RELATED WORKS

The most important issue in image processing is to remove noise from images. This is achieved by maintaining their details as well as features such as texture edges and colours [3–9]. Image denoising also affects the rate of segmentation, classification and similar functions. After the images are captured, some interference occurs in the pixels during the digitalization process. Moreover, vibrations ensue on the sensors during the imaging process [4–8, 11, 12]. This deterioration is categorized as salt and pepper noise (SPN) [13, 14]. SPN generally reduces the image quality [15]. Consequently, many linear/nonlinear filters have been developed to sort out this problem. SPN is easily removed with numerous filters, however, only when practically applied to some few noisy pixels [16, 17] however, others work on all noisy pixels [18]. To fix the new value of a pixel, these filters use a window that consists of the neighbouring pixels of the noisy pixel recognized as the center pixel. The most common filter is the median filter (MF) [19,20]. MF works on the whole item on all pixels. Applying the filter in this manner, nonetheless, blurs the image as well as distorts from the original pixel values. Standard Median Filtering (SMF) works well in low-intensity noise, by applying a small window size on it [21, 22]. The scheme [23] has to remove SPN by a noise level of 90% by using an adaptive median filter. This paper focuses on the quality of an image [24] for different noise densities in the range of 10%-90% with other nonlinear filters. Peak Signal to Noise Ratio (PSNR) and structural similarity (SSIM) [25], have been used for some images as the results of DAMF and other methods. An improved algorithm that used an improved Sobel operator with the integration of median filtering method [26], removes SPN more than classical edge detection. To date, many scholars have used median filter and Sobel filter, to improve the performance of their works. Most of these approaches in this benchmark have their advantages and disadvantages that distinguish them from others. Various proposed methods for decreasing noise have been in discussion. Each method has its advantages and disadvantages.

In this paper, algorithmic optimization has improvised advanced steps with the system architecture modelling. The proposed model manages to remove noise. This research will add value to other former models by introducing algorithmic optimization.

III. METHODS AND MATERIAL

3.1 The Outline of the System

The diagram below describes the flow of the system's data.

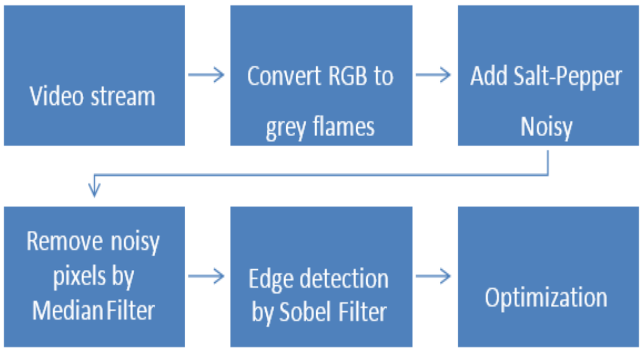

Figure 1: The process steps of the system

The Video input provided by the camera with the OpenCV environment is to convert the RGB colour image by the function (1).

$$Gris = 0.3*R + 0.59*G + 0.11*B \qquad (1)$$

After that, the flames are processed as grey images and the flames will be entered into the Doping System. In the doping system, the flames will have added some random white and black pixels these points are 'Salt-Pepper Noisy'.

srand((unsigned)time(NULL));

$$x = rand(1)\%(b - 0 + 1) + b \qquad (2)$$

The use of a median filter on these noisy points will be removed. Then the image will also be smoother in visual terms. The median filter is an important process before the Sobel edge detection process. Here, the noisy points will be enlarged by the edge detection algorithm. The median filter is non-linear. This means for two images A(x) and B(x):

$$[A(x) + B(x)] \neq [(x)] + media[B(x)] \quad (3)$$

The Sobel operator performs a 2-D spatial gradient measurement on an image. This emphasizes regions of high spatial frequency that correspond to edges. Typically, it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. In theory, at least, the operator consists of a pair of 3×3 convolution kernels as shown in Figure 2. One kernel is simply the other rotated by 90°. This is quite similar to the Roberts Cross operator.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 2: Sobel convolution kernels

These kernels are designed to respond maximally to edges running vertically and horizontally relative to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these Gx and Gy). These are combined to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The gradient magnitude is given by:

$$|G| = \sqrt{G_x^2 + G_y^2} \qquad (4)$$

Typically, an approximate magnitude is computed using:

$$|G| = |Gx| + |Gy \qquad (5)$$

This is much faster to compute. The angle of orientation of the edge (relative to the pixel grid) gives rise to the spatial gradient given by:

$$\theta = \arctan(\frac{Gx}{Gy}) \qquad (6)$$

3.2 Salt-Pepper Noise Simulation

3.2.1 Salt-Pepper Noise

SPN is a form of noise sometimes seen in images. It presents itself as sparsely occurring white and black pixels. Fat-tail distributed, or "impulsive" noise is sometimes called SPN or spike noise [27]. An image containing salt-and-pepper noise will have dark pixels in bright regions and bright pixels in dark regions [28]. This type of noise can be caused by various means including analogue-to digital converter errors and bit errors in transmission [29]. It can be mostly eliminated by using dark frame subtraction and interpolating around dark/bright pixels. Dead pixels in LCD monitors produce a similar, but non-random, display.

Figure 3 is an example of SPN, compared with the original image. The simulated SPN becomes un-clear with the white and black pixels' being.

Figure 3: Comparison Picture for Salt-Pepper Noise and original picture [30]

3.2.2 The Creation of Salt-Pepper Noise

This research mainly concerns the realization of Sobel Filter., The flame flows were transferred by the gris algorithm. Then the design of the simulation of noise will only be for white-black images. Firstly, the defining Doping Degree is required.

Definition 1: The Doping Degree is the percentage of noise pixels share of all pixels in a flame. Considering function (1), the random function of the C standard library is to add the salt points and pepper points randomly. This process can be realised by algorithm 1.

Algorithm steps:

```
Step 1: Start
Step 2: Declare variables dopingdegree= (1-doping_degree)*100), colour_tag=1,
x, y, z, height, width, i, j.
Step 3: FOR every width
        FOR every height
        z ← rand () % (dopingdegree)
Step 4: if (z==1)
        y ← rand () % (height+1);
        x ← rand () % (width+1);

Step 5: if (colour_tag==0)
        im_doping_data [addr(x, y)] =0 // PAPER noise
        colour_tag=1
   else
        im_doping_data [addr(x, y)] =255 //add salt noise randomly
        colour  tag=0; // SALT noise
        im_median_data [addr (j, i)] ←im_doping_data [addr (j, i)]
      ENDFOR
        ENDFOR
Step 6: Stop
```

Algorithm 1. The process to add noise points to an image.

The realization of the Doping Degree was also based on a random function. The system receives the input of the doping degree value A before the image process. Then the possibility of a pixel should be a noise pixel is 1/A. This process can be realized by algorithm 2.

Algorithm steps:

```
Step 1: Start
Step 2: Declare variables dopingdegree= (1-doping_degree)*100), z,y,x,width,height
Step 3: FOR every width
          FOR every height
        z ← rand () % (dopingdegree)// Doping degree control
Step 4: if (z==1)// add noise points
        y ← rand () % (height+1)
        x ← rand () % (width+1)
        ENDFOR
         ENDFOR
Step 5: Stop
```

Algorithm 2. The method to control the number of noise pixels

IV. RESULTS AND DISCUSSION

Figure 4, Figure 5 and Figure 6 indicate the test results of the Doping Module with Doping Degree 10%, 50%, and 90% respectively.

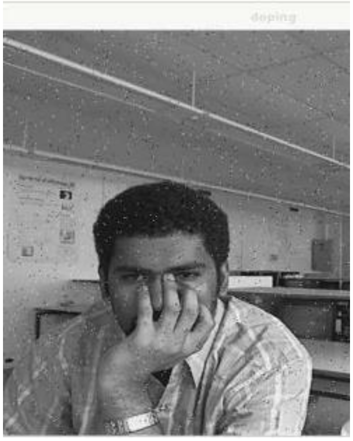

Figure 4 Doping Degree in 10%

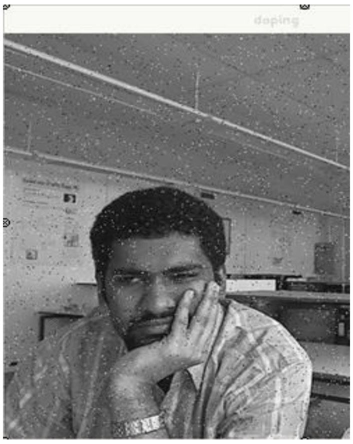

Figure 5 Doping Degree in 50%

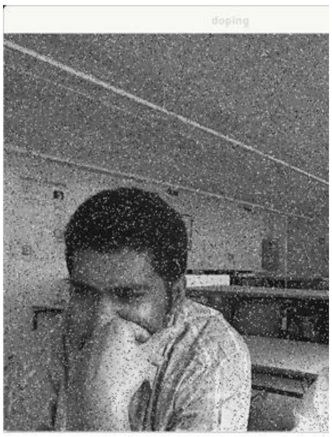

Figure 6 Doping Degree in 90%

4.1 Median Filter

Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbours to decide whether it is representative of its surroundings or not. Instead of simply replacing the pixel value with the mean of neighbouring pixel values. It replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighbourhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighbourhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.) Figure 7 illustrates an example calculation.

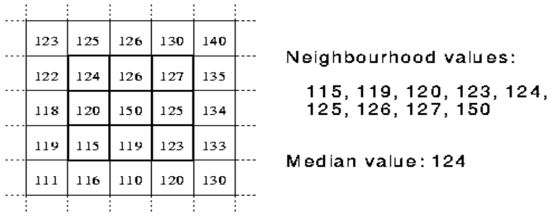

Figure 7: Calculating the median value of a pixel neighbourhood.

The central pixel value of 150 is rather unrepresentative of the surrounding pixels and is replaced with the median value: 124. A 3×3 square neighbourhood is used here. Larger neighbourhoods will produce more severe smoothing [31]. Figure 8 indicates the effect of the median filter for clearing noisy points.

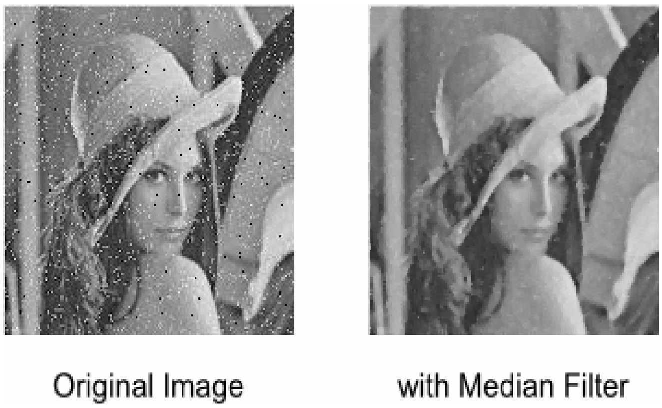

Figure 8: Comparison Picture for Salt-Pepper Noise and original picture [32].

Algorithm Realization of Median Filter

Algorithm 3 indicates the process of the window's creation with the movement of the window on flames. The median filter received the input image by "im_doping_data [addr (jj, ii)]" that was offered by the Doping Module

```
Algorithm steps:
Step 1: Start
Step 2: Declare variables, height, width, i, j, ii, jj,k=0,min,n,window[n],temp
Step 3: FOR every width-1
          FOR every height-1
        unsigned char window [9]
        FOR every j+2
         FOR every i+2
        window[k++] ←im_doping_data [addr (jj, ii)]
Step 4:  FOR every 5// Order elements (only half of them)
        min←m
Step 5:  FOR every 9 //Find the median value among all the 9 points
Step 6:        if (window[n] < window [min]) min← n; //Put found minimum element in its place
            unsigned char temp ← window[m]
            window[m] ←window [min]; window [min] ←temp
            im_median_data [addr (j, i)] ←window [4]
            im_sobel_data [addr (j, i)] ←im_median_data [addr (j, i)]
        ENDFOR
          ENDFOR
        ENDFOR
         ENDFOR
        ENDFOR
        ENDFOR
Step 7: Stop
```

Algorithm 3 realizations of the median filter

A. Module Test Result

Based on different doping degrees different filter effects are given in the graph below.

Figure 9: Filter effects

4.2 DESİGN OF SOBEL FİLTER

4.2.1 Sobel Filter

The Sobel operator is slower to compute than the Roberts Cross operator. It is largely convolution kernel smoothed the input image to a greater extent. It also makes the operator less sensitive to noise. The operator also generally produces considerably higher output values for similar edges, compared with the Roberts Cross.

As with the Roberts Cross operator, output values from the operator can easily overflow the maximum allowed pixel value for image types that only support smallish integer pixel values (e.g. 8- bit integer images). When this happens, the standard practice is to simply set overflowing output pixels to the maximum allowed value. The problem can be avoided by using an image type that supports pixel values with a larger range.

Natural edges in images often lead to lines in the output image that are several pixels wider due to the smoothing effect of the Sobel operator. Some thinning may be desirable to counter this. Failing that, some sort of hysteresis ridge tracking could be used as in the canny operator.

(a)Median filter Result

(b) Sobel filter Result

Figure 10 : Comparison Picture for Median filter and Sobel filter [31]

4.2.2. Realization of Sobel Filter

Based on the mathematic analysis conducted in section 3, and due to its definition, the Sobel operator can be implemented by simple means in both hardware and software. Only eight image points around a point are needed to compute the corresponding result. Furthermore, only integer arithmetic is needed to compute the gradient vector approximation. Also, these two discrete filters

P1	P2	P3
P4	P5	P6
P7	P8	P9

described above are both separable:

|G|=|(P1+2x P2+ P3)+(P7+2xP8+ P9)|+|(P3+2xP6+ P9)-(P1+2x P4+ P7)| (7)

```
Step 1: Start
Step 2: Declare variables, height=118, width, i, j,H,V, and GN=0, i00, i01, i02
Step 3: FOR every width-2
          FOR every height
           i00 ←im_median_data [addr (i-1, j-1)]
           i01 ←im_median_data [addr (i-1, j)]
           i02 ←im_median_data [addr (i-1, j+1)]
           H ← -i00 + i02 - 2*i10 + 2*i12 - i20 + i22;
           V ← i00 + 2*i01 + i02 - i20 - 2*i21 - i22;
Step 4: GN ← abs (H) + abs (V)
Step 5: if (GN > 255) GN ← 255
         im_sobel_data [addr (i, j)] ← GN
      ENDFOR
         ENDFOR
Step 6: Stop
```

Algorithm 4: Calculation of the Sobel convolution

```
#define N_frames 150
cvWaitKey(1);
Doping_Degree = 10%
```

4.2.3 Module Test of Sobel Filter

Based on different doping degrees we achieve different filter effects of the graph below.

Figure 11: Soble Filter

5 OPTİMİZATİON OF THE SYSTEM

5.1 Statistic Data of Original Code

Based on the 'gprof' instruction of the Linux System we analysed our code and acquired the original information.

The program parameters were set as below. The test result was shown in Figure 12.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.13     13.24    13.24                             main
  0.00     13.24     0.00        4     0.00     0.00  cvSize(int, int)
  0.00     13.24     0.00        1     0.00     0.00  _GLOBAL__sub_I_size
  0.00     13.24     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
Call graph
granularity: each sample hit covers 2 byte(s) for 0.08% of 13.24 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0   13.24    0.00                 main [1]
                0.00    0.00       4/4           cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[8]      0.0    0.00    0.00       4         cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [16]
[9]      0.0    0.00    0.00       1         _GLOBAL__sub_I_size [9]
                0.00    0.00       1/1
__static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_size [9]
[10]     0.0    0.00    0.00       1
__static_initialization_and_destruction_0(int, int) [10]
Index by function name

   [9] _GLOBAL__sub_I_size     [8] cvSize(int, int)
  [10]___static_initialization_and_destruction_0(int, int) [1] main
```

Figure 12: statistic result of the original code without optimization

5.2 Optimization for RISC

5.2.1 Loop Unrolling

Most of today's image processing applications rely on the computing power delivered by RISC processors. RISC processors are load/store architectures in the sense that their instructions can process only operands present in CPU registers. To find a register allocation that reduces or possibly minimizes the number of load/store instructions. This is one of the main concerns in the efficient implementation of Image processing programs on load/store architectures. The execution speedups are delivered by source program transformations. In particular by External Loop Unrolling transformation applied to Image Processing programs as largely experimented in previous works. This ultimately led to undertake an analytical investigation on the register allocation delivered by such source program transformations.

After analysing the program with the study of the ARM processor architecture this paper proposes the Loop Unrolling method to optimize the code.

```
for(j=1;j<width-1;j++)
        for(i=1;i<height-1;i++){
    int k = 0;
    unsigned char window[9];
        for (int jj = j - 1; jj < j + 2; ++jj)
for (int ii = i - 1; ii < i + 2; ++ii)

    window[k++] = im_doping_data[addr(jj,ii)];
            //
        for (int m = 0; m < 5; ++m)
                {
                    int min = m;
                    for (int n = m + 1; n < 9; ++n)
if (window[n] < window[min])

    min = n;
        //
        unsigned char temp = window[m];
        window[m] = window[min];
        window[min] = temp;
}

im_median_data[addr(j,i)]=window[4];

im_sobel_data[addr(j,i)]=im_median_data[addr(j,i)
];

}
```

```
for(j=1;j<width-1;j++){

        for(i=1;i<height-1;i++){

  int k= 0;
            unsigned char window[9];


window[k] = im_doping_data[addr(j-1,i-1)];

 window[k+1] = im_doping_data[addr(j,i-1)];
 window[k+2] = im_doping_data[addr(j+1,i-1)];
 window[k+3] = im_doping_data[addr(j-1,i)];
 window[k+4] = im_doping_data[addr(j,i)];
 window[k+5] = im_doping_data[addr(j+1,i)];
 window[k+6] = im_doping_data[addr(j-1,i+1)];
 window[k+7] = im_doping_data[addr(j,i+1)];
 window[k+8] = im_doping_data[addr(j+1,i+1)];
 //
 for (int m = 0; m < 5; ++m)
       {
          int min = m;
          for (int n = m + 1; n < 9; ++n)
          if (window[n] < window[min])
                      min = n;
 //
 unsigned char temp = window[m];
 window[m] = window[min];
 window[min] = temp;
       }
 im_median_data[addr(j,i)]=window[4];
 im_sobel_data[addr(j,i)]=im_median_data[addr(j,i)];
                }
 }
```

Form 1: The original code and the code after loop unrolling

As Form 1 shows, this paper replaced the third 'for' loop as assignment operations. In theory after this process, there are "height * width * 9" loops that will be reduced as "height * width" loops mean "height * width * 8" loops will be removed in Median Filter Module every flame.

5.2.2 Statistic of Loop Unrolling Optimization

This paper tested the time cost of the program that processed by loop unrolling and achieved the statistic result of Figure 13.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.10     12.30    12.30                             main
  0.00     12.30     0.00        4     0.00     0.00  cvSize(int, int)
  0.00     12.30     0.00        1     0.00     0.00  _GLOBAL__sub_I_size
  0.00     12.30     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
      Call graph


granularity: each sample hit covers 2 byte(s) for 0.08% of 12.30 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0   12.30    0.00                 main [1]
                0.00    0.00       4/4           cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[8]      0.0    0.00    0.00       4         cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [16]
[9]      0.0    0.00    0.00       1         _GLOBAL__sub_I_size [9]
                0.00    0.00       1/1
__static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_size [9]
[10]     0.0    0.00    0.00       1
__static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------


Index by function name

 [9] _GLOBAL__sub_I_size     [8] cvSize(int, int)
 [10]___static_initialization_and_destruction_0(int, int) [1] main
```

Figure 13: Statistic result of the code after loop unrolling

After the loop unrolling of the median filter module, based on Figure 12 and Figure 13. This paper concludes this optimization process is effective. Before optimization, the cumulative was 13.24 seconds. Nonetheless, after optimization, this value changed to 12.30 seconds, which means that this method helps to improve the speed of response by:

gcc -D -REENTRANT -lpthread xxx. cpp

This paper concludes that optimization on this part

pthread_create(pthread_t *thread,
const pthread_attr_t *attr, void *
(start_routine)(void*), void *arg);

5.3 Optimization based on Threads Management

5.3.1 Thread Management of Linux

LinuxThreads is an implementation of the Posix 1003.1c thread package for Linux. Unlike other implementations of Posix threads for Linux, LinuxThreads provides kernel-level threads; threads are created with the new clone() system call, and all scheduling is completed in the kernel.

The main strength of this approach is that it can take full advantage of multiprocessors. It also results in a simpler, more robust thread library, especially w.r.t. blocking system calls. LinuxThreads is now obsolete and is being replaced by NPTL. The initial author of LinuxThreads (Xavier Leroy) stopped working on LinuxThreads a long time ago. The glibc development team, Ulrich Drepper, continued working on LinuxThreads for a while, but are now developing NPLT instead.

Using the instruction of this paper, created a new Thread under process.
The information about the Thread management of Linux that be found in form 6-3

Items	Linux
Created a thread	pthread_create
Stop a thread	After running thread stopped automatically；

	pthread_exit ; pthread_cance
Get Current Thread Id	pthread_self
Create Mutex	pthread_mutex_init
Wait for SingleObject	pthread_mutex_lock
Release Mutex	phtread_mutex_unlock
Create Semaphore	sem_init
Wait for Single Object	sem_wait
Release Semaphore	sem_post

Form 2: Thread management of Linux

5.3.2 Statistic of Threads Management Optimization

This paper includes the different modules in the main function, otherwise, the program is a linear program where all the data flow is in sequence. Hence, the multi-thread will have limited used in this program. To test the thread optimization method this research used another simple function and called it on the main. Then demonstrated how to use 'pthread' in linux.

In Linux, different options for compilation exist.

These options control various sorts of optimizations.

Without an optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if stopped, the program with a breakpoint between statements, can then assign a new value to any variable ,or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

```
#include <stdio.h> #include <pthread.h> void thread(void)
{
int i; for(i=0;i<3;i++)
printf("This is a pthread.\n");
}
Int main(void)
{
pthread_t id; int i,ret;
ret=pthread_create(&id,NULL,(void *) thread,NULL);
if(ret!=0){
printf ("Create pthread error!\n"); exit (1);
}
for(i=0;i<3;i++)
printf("This is the main process.\n");
pthread_join(id,NULL);
return (0);
}
```

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all the files when compiling each of them. Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed in this section. Set the compiler's optimization level based on form 3.

gcc -O option flag

Option	Optimization level	Execution time	Code size	Memory usage	Compile-time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

Form 3: Comparison for different Compiler Optimization level

5.4.1 Statistic of Compilation Optimization

This paper tested the time cost of the program that was processed by Compilation Optimization. The statistical result is presented in Figures 14, 15, and 16.

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self

                                                             total
```

```
 time   seconds
100.19     15.37
  0.00     15.37
  0.00     15.37
  0.00     15.37
__static_initialization_and_destruction_0(int, int)

                    Call graph


granularity: each sample hit covers 2 byte(s) for 0.07% of 15.37 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0   15.37    0.00                 main [1]
                0.00    0.00       4/4           cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[8]      0.0    0.00    0.00       4         cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [16]
[9]      0.0    0.00    0.00       1         _GLOBAL__sub_I_size [9]
                0.00    0.00       1/1
__static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_size [9]
[10]     0.0    0.00    0.00       1
__static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
Index by function name

[9] _GLOBAL__sub_I_size     [8] cvSize(int, int)
[10]___static_initialization_and_destruction_0(int, int) [1] main
```

Figure 14: Statistic result of the code with O1

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.16     15.39    15.39                             main
  0.00     15.39     0.00        4     0.00     0.00  cvSize(int, int)
  0.00     15.39     0.00        1     0.00     0.00  _GLOBAL__sub_I_size
  0.00     15.39     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)


                    Call graph


granularity: each sample hit covers 2 byte(s) for 0.06% of 15.39 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0   15.39    0.00                 main [1]
                0.00    0.00       4/4           cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[8]      0.0    0.00    0.00       4         cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       1/1           __libc_csu_init [16]
[9]      0.0    0.00    0.00       1         _GLOBAL__sub_I_size [9]
                0.00    0.00       1/1
  static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_size [9]
[10]     0.0    0.00    0.00       1
  static initialization and destruction 0(int, int) [10]
-----------------------------------------------


Index by function name

   [9] GLOBAL  sub I size     [8] cvSize(int, int)
  [10]   static initialization and destruction 0(int, int) [1] main
```

Figure 15: Statistic result of the code with O2

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.19     15.60    15.60                             main
  0.00     15.60     0.00        4     0.00     0.00  cvSize(int, int)
  0.00     15.60     0.00        1     0.00     0.00  _GLOBAL__sub_I_size
  0.00     15.60     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)

                   Call graph


granularity: each sample hit covers 2 byte(s) for 0.06% of 15.60 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0   15.60    0.00                 main [1]
                0.00    0.00       4/4           cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[8]      0.0    0.00    0.00       4         cvSize(int, int) [8]
-----------------------------------------------
                0.00    0.00       1/1           _libc_csu_init [16]
[9]      0.0    0.00    0.00       1         _GLOBAL__sub_I_size [9]
                0.00    0.00       1/1
_static_initialization_and_destruction_0(int, int) [10]
-----------------------------------------------
                0.00    0.00       1/1           _GLOBAL__sub_I_size [9]
[10]     0.0    0.00    0.00       1
-----------------------------------------------
_static_initialization_and_destruction_0(int, int) [10]
Index by function name

   [9] GLOBAL  sub I size     [8] cvSize(int, int)
  [10]___static_initialization_and_destruction_0(int, int) [1] main
```

Figure 16: Statistic result of the code with O3

V. CONCLUSION

As the complexity as well as requirements of image denoising have enlarged, research in this domain is still in the high mandate. This paper presents a new approach to realise the Sobre imagine process algorithm optimization as well as develop an effective algorithm by using different optimization methods. This will be leading to essential advances in image denoising methods. The analysis is conducted with different compiler options with a different degree. Figures 1 to 3 conclude that the system developed has a better compiler optimization level at O3. Future work will explore other types of noise especially those that are existing in real life.

VI. REFERENCES

[1]. Raymond H. Chan, Chung-Wa Ho, and Mila Nikolova, “Salt-and-Pepper Noise Removal by MedianType Noise Detectors and Detail-Preserving Regularization”- IEEE Transactions on Image Processing, Vol. 14, No. 10, October 2005. DOI: 10.1109/TIP.2005.852196.

[2]. J. Astola and P. Kuosmanen, “Fundamentals of Nonlinear Digital Filtering”, Boca Raton, CRC, 2020. https://doi.org/10.1201/9781003067832.

[3]. Erkan, Uğur, and Levent Gökrem. "A new method based on pixel density in salt and pepper noise removal." Turkish Journal of Electrical Engineering & Computer Sciences 26.1 (2018): 162-171. doi:10.3906/elk-1705-256.

[4]. González-Hidalgo, M., Massanet, S., Mir, A., & Ruiz-Aguilera, D. (2013, September). “A fuzzy

filter for high-density salt and pepper noise removal". Spanish Association for Artificial Intelligence (pp. 70-79). Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-642-40643-0_8.

[5]. Xiao L, Li C, Wu Z, Wang T. "An enhancement method for X-ray image via fuzzy noise removal and homomorphic filtering". Neurocomputing 2016; 64: 195:56-64. https://doi.org/10.1016/j.neucom.2015.08.113.

[6]. Coupe P, Manjon JV, Robles M, Collins DL, "Adaptive multiresolution non-local means filter for three-dimensional magnetic resonance image denoising". IET Image Process 2012; 6: 558-568. DOI: 10.1049/iet-ipr.2011.0161.

[7]. Baljozovic D, Kovacevic B, Baljozovic A. "Mixed noise removal filter for multi-channel images based on halfspace deepest location". IET Imag Proc 2013; 7: 310-323. DOI: 10.1049/iet-ipr.2012.0105.

[8]. Sreenivasulu, P., and N. Krishna Chaitanya. "Removal of Salt and Pepper Noise for Various Images Using Median Filters: A Comparative Study." IUP Journal of Telecommunications 6.2 (2014).

[9]. Sakthidasan K, Sankaran A, Nagappan VN, "Noise-free image restoration using hybrid filter with adaptive genetic algorithm". Computers & Electrical Engineering Volume 54, August 2016, Pages 382-392. https://doi.org/10.1016/j.compeleceng.2015.12.011.

[10]. Thanha DNH, Dvoenkoa SD, "A method of total variation to remove the mixed Poisson–Gaussian noise". Pattern Recogn 2016; 26: 285-293. https://doi.org/10.1134/S1054661816020231.

[11]. Zhang C, Wang K. "A switching median–mean filter for removal of high-density impulse noise from digital images". Optik 2015; 126: 956-961. https://doi.org/10.1016/j.ijleo.2015.02.085.

[12]. Gellert, Arpad, and Remus Brad. "Context-based prediction filtering of impulse noise images." IET Image Processing 10.6 (2016): 429-437. DOI: 10.1049/iet-ipr.2015.0702.

[13]. Vasanth, K., and V. Jawahar Senthil Kumar. "Decision-based neighborhood-referred unsymmetrical trimmed variants filter for the removal of high-density salt-and-pepper noise in images and videos." Signal, Image and Video Processing 9.8 (2015): 1833-1841. https://doi.org/10.1007/s11760-014-0665-0.

[14]. Chan RH, Ho CW, Nikolova M. "Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization". IEEE T Image Process 2005; 14: 1479-1485. DOI: 10.1109/TIP.2005.852196.

[15]. Zhao, Feng, Rui Chuan Ma, and Jia Qing Ma. "An Algorithm for Salt and Pepper Noise Removal Based on Information Entropy." Applied Mechanics and Materials. Vol. 220. Trans Tech Publications Ltd, 2012. DOI:https://doi.org/10.4028/www.scientific.net/AMM.220-223.2273.

[16]. Erkan, Uğur, and Adem Kilicman. "Two new methods for removing salt-and-pepper noise from digital images." scienceasia 42.1 (2016): 28. doi:10.2306/scienceasia1513-1874.2016.42.028.

[17]. Roig, Bernardino, and Vicente D. Estruch. "Localised rank-ordered differences vector filter for suppression of high-density impulse noise in colour images." IET Image Processing 10.1 (2015): 24-33. doi: 10.1049/iet-ipr.2014.0838.

[18]. Hwang, Humor, and Richard A. Haddad. "Adaptive median filters: new algorithms and results." IEEE Transactions on image processing 4.4 (1995): 499-502. DOI: 10.1109/83.370679.

[19]. Jin, Lianghai, Caiquan Xiong, and Hong Liu. "Improved bilateral filter for suppressing mixed noise in color images." Digital Signal Processing 22.6 (2012): 903-912. https://doi.org/10.1016/j.dsp.2012.06.012.

[20]. Sreenivasulu, P., and N. Krishna Chaitanya. "Removal of Salt and Pepper Noise for Various Images Using Median Filters: A Comparative Study." IUP Journal of Telecommunications 6.2 (2014).

[21]. Sun, Chen, et al. "An efficient method for salt-and-pepper noise removal based on shearlet transform and noise detection." AEU-International Journal of Electronics and Communications 69.12 (2015): 1823-1832. https://doi.org/10.1016/j.aeue.2015.09.007.

[22]. Xiao, Yu, et al. "Restoration of images corrupted by mixed Gaussian-impulse noise via l1–l0 minimization." Pattern Recognition 44.8 (2011): 1708-1720. https://doi.org/10.1016/j.patcog.2011.02.002.

[23]. Chan, Raymond H., Chung-Wa Ho, and Mila Nikolova. "Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization." IEEE Transactions on image processing 14.10 (2005): 1479-1485. DOI: 10.1109/TIP.2005.852196.

[24]. Sohi, Prateek Jeet Singh, et al. "Noise density range sensitive mean-median filter for impulse noise removal." Innovations in computational intelligence and computer vision. Springer, Singapore, 2021. 150-162. https://doi.org/10.1007/978-981-15-6067-5_18.

[25]. Uğur Erkan, Levent Gökrem, Serdar Enginoğlu, "Different applied median filter in salt and pepper noise.", Computers & Electrical Engineering, Volume 70, 2018, Pages 789-798, ISSN 0045-7906,https://doi.org/10.1016/j.compeleceng.2018.01.019.

[26]. Patil, Swati A., and Atul D. Patil. "An Effective Multi-Frame Super Resolution of Image from Blurry and Noisy Images Using PCA." International Journal of Electronics Communication and Computer Engineering 5.1 (2014): 12.

[27]. Bovik, Alan C. Handbook of image and video processing. Academic Press, 2010.

[28]. Linda G. Shapiro and George C. Stockman (2001, Computer Vision, pp 279-325, New Jersey, Prentice-Hall, ISBN 0-13-030796-3.

[29]. Sampat, Mehul P., Mia K. Markey, and Alan C. Bovik. "Computer-aided detection and diagnosis in mammography." Handbook of image and video processing 2.1 (2005): 1195-1217.

[30]. Tien, Kun-yu & Samani, Hooman & Lui, Jui. (2017). A survey on image processing in noisy environment by fuzzy logic, image fusion, neural network, and non-local means. 1-6. 10.1109/CACS.2017.8284240.

[31]. Yao, Yuqin. "Image Segmentation Based on Sobel Edge Detection." (2016).

[32]. https://www.graphicsmill.com/docs/gm/minimum-maximum-median-filters.htm.

Cite this article as :

Waheed Muhammad SANYA, Gaurav BAJPAI, Haji Ali HAJI, "Implementation and Optimization of Image Processing on the Map of SABRE i.MX_6", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 7 Issue 6, pp. 402-417, November-December 2021. Available at

doi : https://doi.org/10.32628/CSEIT217690

Journal URL : https://ijsrcseit.com/CSEIT217690