METHODS FOR REAL TIME IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS

Sorin ZOICAN[1], Roxana ZOICAN[2], Dan GALAȚCHI[3]

For certain types of applications, such as videoclips processing, the image processing may not terminate within its deadline. The paper presents methods for implementation of image processing algorithms in real time. These methods involve using of graphical processing units (GPU) or digital signal processors (DSP). The illustrated methods may be applied for many image processing algorithms. In general, a denoising algorithm has two phases which are run sequentially: the first one determines the noisy pixels and the second applies a median filtering considering the only good pixels. In all such denoising algorithms, the first phase is run for multiple times depending on the noise power. The second phase also may be executed more than one time but this depends on the specific algorithm. The methods presented includes: 1) parallel processing using the GPU, 2) DSP implementation based on a Blackfin microcomputer with support of Visual DSP kernel (VDK) and 3) adjust the number of iteration to be executed in each phase of the algorithm in such way so the deadline to be respected. The paper proposes two frameworks for real time implementation of very complex image processing algorithms: a) based on GPU and b) based on DSP and integrates optimization solutions such as adjusting the algorithm and exploits the DSP parallelism. The first method is appropriate for applications in personal computers and the second may be used in mobile devices.

Keywords: impulsive noise removal; graphic processing unit; digital signal processor (DSP); real time implementation

1. Introduction

In a multimedia application image is often corrupted by impulsive noise due the errors in the transmission channel. Impulsive noise, called "salt and pepper", is caused by camera sensors, faulty hardware memory locations, or because errors occurred during communication channels transmitting images, affecting randomly a fraction of the total number of pixels, leaving other pixels

[1] Professor, Electronics, Telecommunication and Information Technology, Telecommunications Department, University POLITEHNICA of Bucharest, Romania, e-mail: sorin@elcom.pub.ro

[2] Professor, Electronics, Telecommunication and Information Technology, Telecommunications Department, University POLITEHNICA of Bucharest Bucharest, Romania, e-mail: roxana@elcom.pub.ro

[3] Associate Professor, Electronics, Telecommunication and Information Technology, Telecommunications Department, University POLITEHNICA of Bucharest, Romania, e-mail: dg@elcom.pub.ro

unchanged. It is important to eliminate this type of noise in the images before they can apply other subsequent processing methods such as contour detection, object recognition or image segmentation. Many denoising algorithms exist almost all based on median filtering [1]. However, the median filter may cause blurred in the reconstructed image. To overcome this phenomena a noised pixel detector is applied before median filtering. In such way the edges in the image will be preserved. The noised pixels detector is repeatedly applied over the image in order to achieve better results [1], [2]. Unfortunately, this additional phase added to the classical median filter increases the computation time and it is possible that the application not run in real time. Moreover, many image denoising algorithms have a second phase that computes median value adaptively using the results from the first phase. Fig. 1 illustrates an image denoising algorithm with two phases [1].

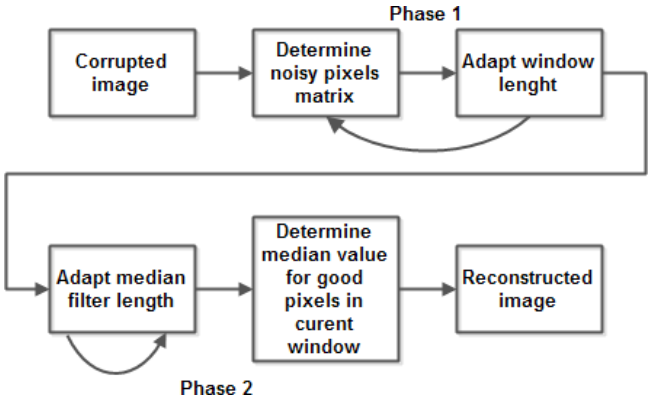

Fig. 1. The generic image denoising algorithm

For videoclips, the algorithm is applied each frame.

In real time systems, there are specific deadlines to be met [3]. In particular, for a video processing application, deadlines are determined by the number of frames per second. If processing time exceeds the deadline then the following methods may be used to maintain system functionality: reserving of additional resources, tasks skip, adaptation of the task activation period and an adaptation of task execution time. In embedded systems, with limited resources, additional resources reserving can not be a viable option. More, the activation period is fixed and it not be modified without a severe degradation in performance. For example, a videoclip cannot play at a different (larger) frame per second rate, different from the original rate. In such systems only the task skipping and execution time adaptation may lead to deadline meeting while preserving the functionality. The videoclip processing should be divided into two task category: mandatory tasks and additional tasks. [3], [4]. The mandatory task ensures a basic quality of the videoclip and the additional tasks improves the quality. In case that the deadline may be exceeded some of additional tasks (or all of them) will be skipped. The execution time adaptation is as similar methods. In this situation, the videoclip processing may be divided into several tasks consists of iterative

sequences. The image processing performance increases with the number of iterations of each task.

2. The framework for real time implementation using GPU

This section proposes a framework for real time implementation of image processing algorithm using Computer Unified Device Architecture (CUDA) technology - a software programming model for programmers to write scalable parallel programs using C. In the CUDA programming model, the Graphical Processing Unit (GPU), or device, is viewed as a computing element that works in cooperation with the main Central Processing Unit (CPU), or host. The issues regarding with mapping the noise removal algorithm on CUDA are the following: the parallelization of the algorithm, the efficient configuration of the kernel and the efficient memory accesses in device and between device and host. The CUDA device architecture is illustrated in Fig. 2 [5].

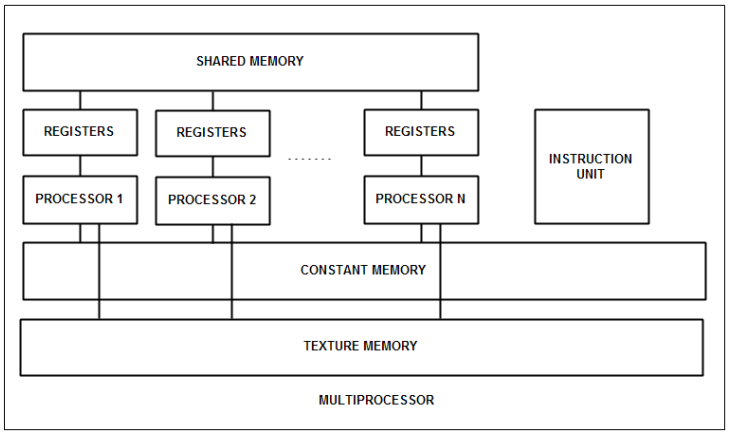

Fig. 2. CUDA Hardware architecture

A CUDA program consists of one or more phases that are executed on either the host or a device. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. The device code consists of data-parallel functions, called kernels, and their associated data structures [6]. The kernel function is executed, in parallel, for a large number of threads. The execution of a typical CUDA program is illustrated in Fig. 3. Threads are organized into blocks and blocks are organized into a grid. A multiprocessor executes one block at a time. A set of threads executed in parallel represents a warp. The execution starts with host execution. When a kernel function is invoked, the execution is moved to a device, where a large number of threads are generated to take advantage of data

parallelism. In a processing image application for each pixel the same computations are performed, so the number of threads is the number of total pixels in the image.

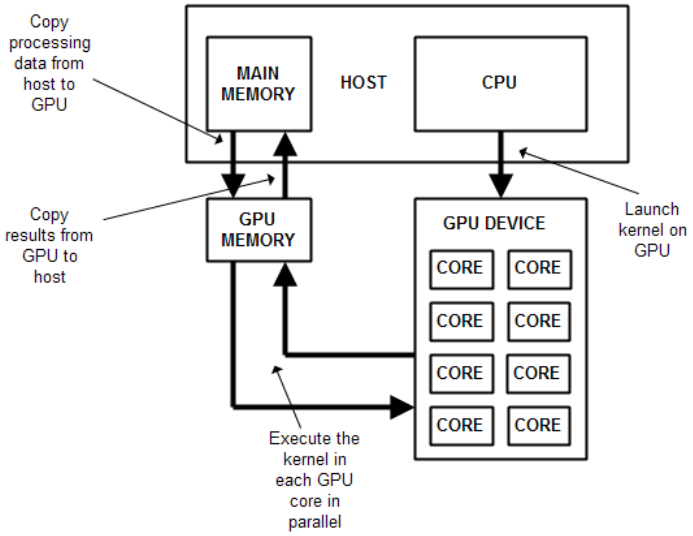

Fig. 3. CUDA program flow

Once a block is assigned to a multiprocessor; it is further divided into 32-thread units called warps that contain threads with consecutive, increasing thread identifier. The multiprocessor scheduler manages warps and will execute a SIMT (Single Instruction Multiple Threads) instruction by all the threads in a warp as soon as the warp is ready and resources are available. The branching instruction should be avoided in order to ensure that all the executing threads of a warp be executing the same instruction.

Following the algorithm presented above, we split it into two kernels that will be launched by the host but will be executed by the GPU. The two kernels implement the two phases of the noise removal algorithm and they are illustrated in Fig. 4. One of the keys to good performance is to keep the multiprocessors on the device as busy as possible. A device in which work is poorly balanced across the multiprocessors will deliver suboptimal performance. Hence, it's important to design the application to use threads and blocks in a way that maximizes hardware utilization and to limit practices that impede the free distribution of work. The ratio of the number of active warps per multiprocessor to the maximum number of possible active warps is called occupancy. One of several factors that determine occupancy is register availability. If each thread block uses many registers, the

number of thread blocks that can be resident on a multiprocessor is reduced, thereby lowering the occupancy of the multiprocessor.

Optimization could be made in order to increase the performance of the implementation: the image will be stored in texture memory of CUDA device, avoid the unnecessary memory transfers between host and device and carefully configure the kernels [7]. The texture memory space is cached and optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve the best performance. Reading device memory through texture fetching is an advantageous alternative to reading device memory from global or constant memory. Another way to optimize is using of coalesced memory accesses. Coalescing means that a memory read by consecutive threads in a warp is combined by the hardware into several, wide memory reads. The requirement is that the threads in the warp must be reading memory in order.

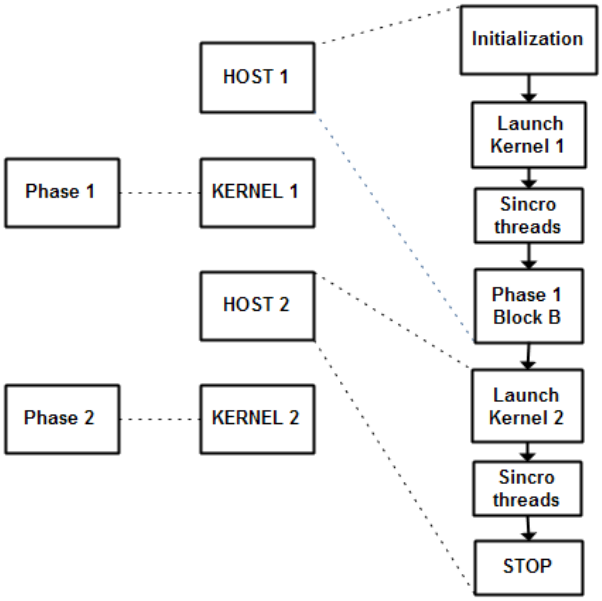

Fig. 4. CUDA mapping of noise removal algorithm

The dimension of the grid must be carefully chosen. If the grid dimension is multiple of the number of multiprocessors then all the multiprocessors will be equally loaded and the device is kept busy as much as possible. On the other hand, the number of coalesced memory accesses will be increased if such condition is met. The grid dimension is calculated accordingly with the widths and height of the image to be processed. If the image width is W and the image height is H, then the grid dimension is $(W/T_x, H/T_y)$, where T_x and T_y represent the number of threads, in horizontal and vertical direction, in a block. The product

$W/T_x * H/T_y$ should be multiple of the number of multiprocessors. It may be necessary to resize the image to meet such condition. The mentioned algorithm was run on two devices: GeForce 8400 GS (with one multiprocessor and eight cores) and GeForce 8800 GTS (with twelve multiprocessors and ninety six cores). The program was also run on a PC with Windows 7, Intel dual core processor at 2.4 GHz and 3 Gbytes of RAM. All the figures in Table 1 are for the second case, considering the first case as 100%. The two kernels have need of 14 or 15 registers and 29 bytes of shared memory each. The occupancy is 66% and it is as large as possible, so the kernels are optimally implemented. As it is shown in Table 1, the global memory efficiency and the coalesced memory accesses are improved and the miss texture cache rate is reduced when the grid dimension is multiple of the number of multiprocessors.

Table 1

GPU efficiency

Global load efficiency	-10%
Global store efficiency	-15%
Miss texture cache	500%
Global load un-coalesced	17%
Global store un-coalesced	150%

Table 2 indicates the performance (execution time in frames per seconds - FPS) for various image sizes. The difference between cases GeForce 8800 (1) and GeForce 8800 (2) is that in the second case the grid dimension is multiple of the number of multiprocessors. One can observe that, in this case, the performance is improved.

Table 2

Execution time for GPU implementation

Image size	**FPS**		
	(480,480)	(320,240)	(640,480)
GeForce 8400	7	22	5
GeForce 8800 (1)	38	115	29
GeForce 8800 (2)	184	553	138

If the image size is not properly chosen, so the dimension of the grid to be multiple of the number of processors, the algorithm was not implemented so FPS is not applicable. The FPS in table 2 is calculated for a grayscale image therefore for a color image (say in RGB format) the figures should divide by 3. In table 2, one may observe that the noise removal algorithm may be implemented in real time (about 15-25 FPS) even for color videos.

3. The framework for real time implementation using DSP

This section illustrates a framework for real time implementation of image processing algorithms suitable for mobile devices using DSP microcomputers, such Blackfin family. Below we will be explain in more details the fundamentals of the modification in the image processing algorithm and how the video pixel instructions works and reduce the processing time.

The Blackfin microcomputer is a 16-bit fixed-point processor that is based on the MSA core, developed in cooperation by Analog Devices and Intel. Because of its low cost and high performance, this processor is suitable in power-sensitive applications (such as cellular phones) and computationally intensive applications (video equipment, third-generation cellular phones). The Blackfin core combines dual multiply-accumulate (MAC) units, an orthogonal reduced instruction-set computer (RISC) instruction set, single instruction, multiple data (SIMD) programming capabilities and multimedia processing features, into a unified architecture. The Blackfin BF5xx processor includes system peripherals such as a parallel peripheral interface (PPI), serial peripheral interface (SPI), serial ports (SPORTs), general-purpose timers, universal asynchronous receiver transmitter (UART), real-time clock (RTC), watchdog timer, and general-purpose input/output (I/O) ports. Blackfin processors have high peripheral supports, a memory management unit (MMU) and RISC-like instructions, which are usually found in many high-end microcontrollers. These processors have high-speed buses and highly developed computational units that support variable-length arithmetic hardware operations. The Blackfin processor uses a modified Harvard architecture which allows multiple memory accesses per clock cycle. The Blackfin processor instruction set is optimized so that 16-bit operation codes represent the frequently used instructions. Complex DSP instructions are encoded into 32-bit operation codes like multifunction instructions. Blackfin microcomputers bear a limited multi-issue facility, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use several of the core resources in a single instruction cycle. The Blackfin architecture supports instructions that control vector operations. We take advantage of these instructions to carry out concurrent operations on multiple 16-bit values, as well as add, subtract, multiply, shift, negate, pack and search [9]. Fig. 5 illustrates the Blackfin core architecture [8].

The image processing algorithm may be implemented on a Blackfin microcomputer evaluation board EZ-KIT-Lite BF533 in C language using the Visual DSP++ integrated development environment with compiler optimization for speed [9], [10].

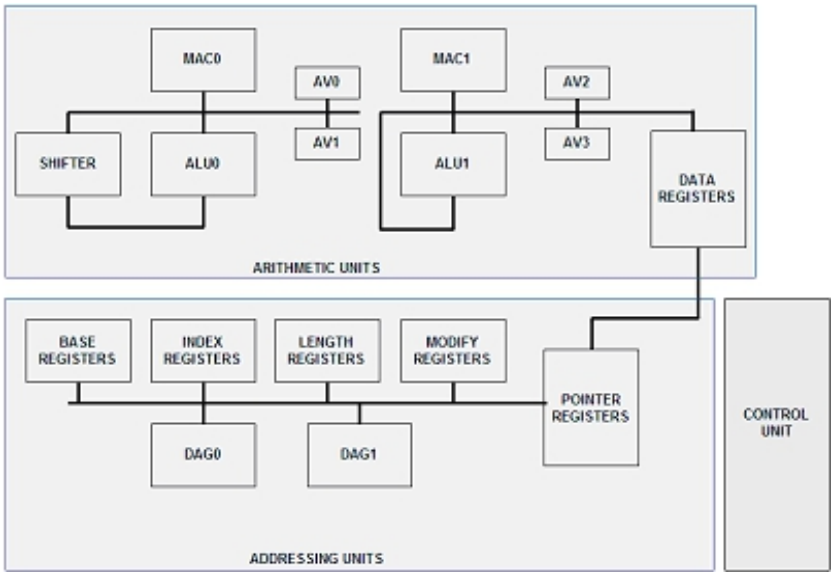

Fig. 5. The Blackfin microcomputers core architecture

Due the algorithm complexity it is possible that it does not run in real time in this implementation if the scanning window length is large. Therefore, optimization methods are necessary in order to achieve a real time implementation (that is the frame processing to be performed in less than 40 milliseconds). Execution time increases when impulsive noise is high because the number of iterations in *Phase 1* and *Phase 2* will increase. It was shown that the improvement is small if the number of iterations is increased over a specific limit [11], so the algorithm can have less iteration in each phase, decreasing the execution time, image quality still being good. A modified algorithm where the execution time adaptation is used was illustrated in [12]. The modified algorithm is aware about the remaining execution time of each task before it reaches its deadline. If the remaining time is less than a specific threshold the scheduling algorithm notify the task to modify the running parameters in *Phase 1* and in *Phase 2* so that the execution time to be reduced and the deadline to be not exceeded. The implementation of the modified algorithm integrates the basic image processing algorithm presented above with the phase running control in order to achieve the real time constraints [12]. The operating system kernel (Visual DSP Kernel- VDK) [10] was involved to support the implementation. The image processing algorithm may be implemented easily using VDK functionality: each phase in algorithm will be separately coded in a dedicated task and a time measurement mechanism will be defined using a periodic semaphore [12]. Specific methods to optimize the tasks execution were involved. These methods will be discussed below for achieving a real time implementation using the digital

signal processing Blackfin microcomputer family [9]. The Blackfin processor has a dual multiply and accumulate (MAC) signal processing engine, an orthogonal instruction set and single instruction multiply data (SIMD) instructions. Issuing parallel instructions and using vector operations may be used to obtain a real time functioning. The Blackfin processor does permit up to three instructions to be issued in parallel: one 32-bit DSP instruction and two 16-bit instructions (load/store, DSP load). A powerful feature of Blackfin processors is the existence of instructions that manipulate video pixels. Such instructions perform 8-bit pack and unpack, quad 8-bit subtract operation that can be used to compute the minimum and maximum values in four windows simultaneously. Also, four values are updated in parallel. Using assembly language implement the iterative instruction as hardware loops that save processor cycles. A more detailed description of implementing quad operation, specifically for such image processing algorithms, using Blackfin microcontroller was illustrated in [11],[12].

Additionally, dual-core Blackfin processors, (BF561 and BF60x) may be involved. Each dual-core Blackfin processor has two cores, core A and core B, each with its own internal memory. There is a common memory shared between the two cores, and both cores share access to external memory. Each core functions independently. Common routines and data will be placed in shared memory without the need for explicit positioning.

The application program and data blocks are transferred from an external memory device to specified internal memory locations. Once all blocks are loaded, core A program execution is started. Core B remains in a held-off state until a certain register bit is cleared, by execution of an instruction sequence in the core A. After that, core B will start execution of its own application program. The frame processing task is designed as a dual - core application that allows for splitting the main code on the two cores and for all of the shared memory areas to be used efficiently by both cores. Images can be seamlessly captured or displayed using the parallel peripheral interface (PPI) using the appropriate direct memory access (DMA) mode such that images can be processed in real time without loosing a frame [13]. The size of data buffers involved in multimedia applications exceeds the processor's internal memory space. To take advantage of the low latency access of the processor's on-chip memory the data can be transferred to the internal memory before it is requested by the programs runs by the core. In this manner, cycles consumed due to the core being held off for a memory request will be avoided. The DMA unit may be used to hide the latency of the memory transfer [14]. In Fig. 6 is illustrated how the DMA unit is involved in order to acquire the input frames or to store the output frames. Four input DMA buffers, *DMA Buffer In 0*, *DMA Buffer In 1, DMA Buffer In 2* , *DMA Buffer In 3* and similar four output buffers *DMA Buffer Out 0*, *DMA Buffer Out 1, DMA Buffer Out 2* , *DMA Buffer Out 3* are defined. Fig. 6 illustrates both the input and output

transfers and for simplicity the buffers are denoted as *DMA Buffer 0*, *DMA Buffer 1*, *DMA Buffer 2* and *DMA Buffer 3*. The buffers are manipulated using pointers which are stored in two tables *TABPin* = { *DMA Buffer In 0*, *DMA Buffer In 1*, *DMA Buffer In 2* , *DMA Buffer In 3*} and *TABPout* = { *DMA Buffer Out 0*, *DMA Buffer Out 1*, *DMA Buffer Out 2* , *DMA Buffer Out 3* }. The DMA transfers use alternatively as start address the four pointers defined in these two tables (for input or output transfers).

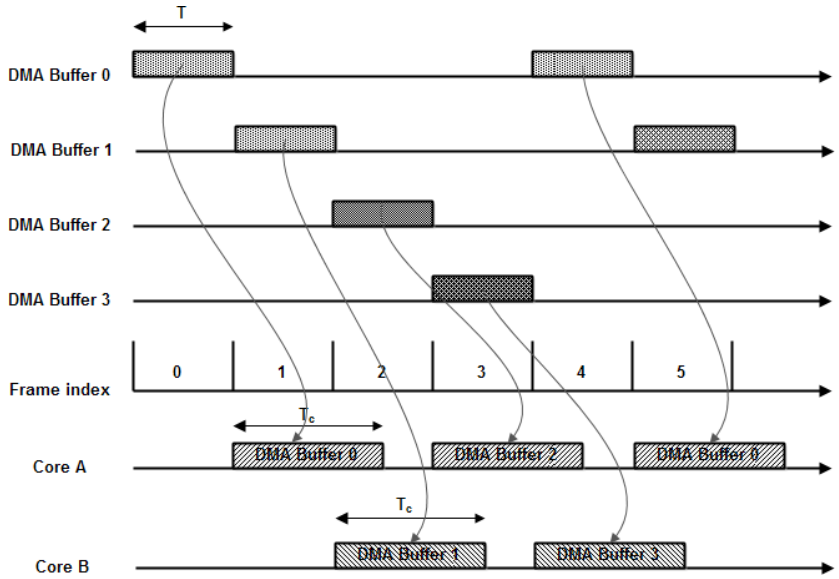

Fig. 6. The DMA buffers for input frame processing

Two successive input frames may be processed in the two cores of the processor. When a DMA transfer is ready a specific interrupt is generated. A variable, *frame_index* is defined in order to establish which input or output frame buffers will be processed. The interrupt service routine will be setup the input and the output frame pointers, *BufferIn* and *BufferOut* as follows:

$$BufferIn = TABPin[frame\_index]; BufferOut = TABPout[frame\_index];$$
$$frame\_index = (frame\_index + 1) \bmod 4;$$

The image denoising algorithm, with number of iterations control, was implemented using the Blackfin BF561 microcomputer and run with different values of impulsive noise and maximum size of the scanning window. The above presented techniques for code optimizing were involved.

Table 4 illustrates the average execution time, in frames per seconds (FPS), for various image sizes. In this table the maximum number of iterations in each phase, L_{max} is considered as a parameter.

Table 3

The execution time in FPS for various grayscale image sizes

Image size	FPS		
	$L_{max} = 1$	$L_{max} = 2$	$L_{max} = 3$
(176,144)	266	95	48
(320,240)	87	31	16
(480,320)	43	15	8
(640,480)	21	7	4
(960,540)	13	4	2

The average time represents the arithmetic mean of the execution time for noise levels ranging from 10% to 90%. One can observe that the new median filtering can be used for color image size about 320×240, if the admitted frame per second is minimum 25. For this limit there is a possibility to obtain a real time functioning for the minimum scanning window.

4. Conclusion

This work proposed two methods (frameworks) for real time implementation of very complex image processing algorithms such as image noise reduction algorithm. A generic image noise reduction algorithm, with two phases, is implemented in order to validate the proposed methods. Each phase is running for multiple times in order to efficiently eliminate the noise.

The first method involves the CUDA enabled graphics cards (CUDA-GPU), already available in many computers. The kernel configuration, minimizing the number of the registers per thread and the number of global memory accesses are key factors that lead to a very efficient implementation of image processing algorithms. If the program meets certain performance rules, the obtained speedup is more than hundreds times compared to a PC computer.

The second method may be used in mobile devices and it involved a DSP microcomputer and slightly modifies the image processing algorithm in order to reduce the number of iterations of the denoising algorithm. A modified algorithm, that controls the number of iterations in each phase, which trades off between the quality of the restored image and the constraint to meet the deadlines, should be implemented for Blackfin DSP in order to achieve the real time functionality. Some optimization solutions, such as using of parallel instructions, vector instructions and dual core microcomputers are involved in order to achieve best performance.

The modified algorithm may be implemented in the CUDA framework in order to achieve more performance in terms of speed of computations.

The presented frameworks are indented to be used in different applications: the first one in personal computer applications and the second one in mobile devices. The GPU approach has the advantage of greater FPS comparing with the DSP method, but it is dependent of resources in graphic unit and of image size (that is, the image size determines the optimality of the parallelism by kernel configuration). The DSP approach provides a smaller FPS, but greater enough for a real time implementation if the image size is reasonable. Both methods may be used for processing medium color image sizes.

R E F E R E N C E S

[1] *Manohar Annappa Koli* , "Robust Algorithm for Impulse Noise Reduction ", International Journal on Computer Science and Engineering (IJCSE), Vol. 02, No. 07, 2010, 2375-2377

[2] *Zhou Wang and David Zhang*, "Progressive Switching Median Filter for the Removal of Impulse Noise from Highly Corrupted Images", IEEE Transactions On Circuits And Systems—II: Analog And Digital Signal Processing, vol. 46, no. 1, Jan. 1999, pp. 78-80

[3] Real Time Systems, Architecture, Scheduling and Application, Seyed Morteza Babamir, editor, Ed. Intech 2012, ISBN 978-953-51-0510-7

[4] Embedded Systems and Wireless Technology, Raul Aquino Santos and Arthur Edwards Block, editors, CRC Press, 2012, ISBN 978-1-57808-803-4

[5] *Jason Sanders, Edward Kandrot*, CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, ISBN-978-0131387683, 2011

[6] GPU Programming Guide, 2011, http://developer.download.nvidia.com/ GPU_Programming_Guide/

[7] CUDA C Best Practices Guide, 2011, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/

[8] Analog Devices, Blackfin Processors Manual, 2007

[9] Analog Devices, Inc., Blackfin Processor Programming Reference, 2012

[10] Analog Devices, Inc., VisualDSP 5.0 Kernel (VDK) Users Guide , 2011

[11] *Sorin Zoican*, "QoS Scheduling Algorithm for Videoclips Denoising", Recent Advances in Electronics, Signal Processing and Communication Systems, Proceedings of the 2013 International Conference on Electronics, Signal Processing and Communication Systems (ESPCO 2013), Venice, Italy, September 28-30, 2013, pp. 40-44

[12] *Sorin Zoican*, "Videoclip Denoising Algorithm Implementation Using the Blackfin Microcomputer Family", IERI Procedia, vol. 4 (2013), ELSEVIER Journal, ISSN: 2212-6678, International Conference on Electronic Engineering and Computer Science (EECS 2013), Beijing, China, May 22-23 2013, pp. 139-147.

[13] *Kunal Singh and Ramesh Babu*, Video Framework Considerations for Image Processing on Blackfin Processors, Analog Devices EE276 , 2005

[14] *Kaushal Sanghai*, Video Templates for Developing Multimedia Applications on Blackfin Processors, Analog Devices EE301 , 2006