REAL-TIME IMAGE PROCESSING WITH SOFTWARE PARALLELIZATION

Radu DOBRESCU[1], Stefan MOCANU[2], Dan POPESCU[1]

Abstract. *The aim of the paper is to present a software architecture that allows developing applications for real-time parallel image processing. The challenge was that the algorithms for processing real-time low level operations on digital images can be developed and prototyped on both a cluster of desktop PCs and on a multi-core architecture of general purpose graphic processors units (GPGPU, by using a dedicated parallel processing platform model. The validation of this model shows how to use parallelizable patterns and how to optimize the load balancing between the workstations.*

Key words: Biological system modeling, Biomedical informatics, Cancer, Fractals, Tumor growth, Simulation

1. Introduction

Real-time image and video processing systems involve processing vast amounts of image data in a timely manner for the purpose of extracting useful information, which could mean anything from obtaining an enhanced image to intelligent scene analysis. Digital images and video are essentially multidimensional signals and are thus quite data intensive, requiring a significant amount of computation and memory resources for their processing. The amount of data increases if color is also considered. Furthermore, the time dimension of digital video demands processing massive amounts of data per second. One of the keys to real-time algorithm development is the exploitation of the information available in each dimension. For digital images, only the spatial information can be exploited, but for digital videos, the temporal information between image frames in a sequence can be exploited in addition to the spatial information.

The key to cope with this issue is the concept of parallel processing which deals with computations on large data sets. In fact, much of what goes into implementing an efficient image/video processing system centers on how well the implementation, both hardware and software, exploits different forms of parallelism in an algorithm, which can be data level parallelism - DLP or/and instruction level parallelism – ILP [1]. DLP manifests itself in the application of the same operation on different sets of data, while ILP manifests itself in scheduling the simultaneous execution of multiple independent operations in a pipeline fashion.

---

[1]Prof., Control Systems and Industrial Informatics Dept., Univ. “Politehnica” Bucharest, Romania.
[2]Conf., Control Systems and Industrial Informatics Dept., Univ.“Politehnica” Bucharest, Romania.

Usually, in a decision theoretic based pattern recognition system for industrial applications, the classification is performed in the feature space by a distance function criterion. In applications like visual servoing, vehicle navigation, industrial inspection, multimedia, medical engineering, etc., the main requirement for the video system is the real time execution of the algorithms. In order to obtain a very high image processing speed, the primary operators (pre-processing operators) are transferred from the central computer to the sensory level.

There are two classes of digital primary image processing operators: local operators and global operators. The global operators require information from the complete image frame. They are not suitable for industrial video applications because they have two main disadvantages: long time execution and edge alteration. On the other hand, many functions like noise rejection, binary segmentation, edge extraction, erosion, dilation, area evaluation, and perimeter evaluation can be calculated with the aid of local bi-dimensional filters.

Generally, software implementation of many image processing procedures is not compatible with on-line, real time operation requirements and with hard industrial environment conditions. Moreover, most of the required primary image processing procedures can be hardware implemented, using programmable devices. Thus, for an efficient industrial image processing system, the hardware/software co-design approach is highly recommended.

Operations like noise rejection, edge detection, binary segmentation of image, are frequently encountered. Due to the development of the integrated circuits like FPGA and DSP, these primary image processing algorithms can be implemented together with the video camera like embedded system.

Based on a multi-core architecture and high memory bandwidths, today’s graphic processors (GPU) offer a great support and speedup not only for dedicated multimedia and graphic applications but for a wide variety of general purpose software. Originally designed as accelerators for 2D and 3D graphic operations, GPUs offer extensive resources for massive parallelism and, even more, they show superior performance to the CPUs for certain classes of applications.

Driven by the insatiable market demand for real time, high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multithreaded, multi-core processor with enormous computational power and very high memory bandwidth. The first performing professional application was introduced in 2006 by NVIDIA under the name *Compute Unified Device Architecture* (CUDA). This concept involves a general purpose parallel computing architecture (hardware and software) which allows efficient solving of a wide range of complex computational problems. The evolution in the GPU’s field was impressive.

Today, parallel GPUs have begun making computational inroads against the CPU, and a subfield of research, dubbed GPU Computing or GPGPU for General Purpose Computing on GPU, has found its way into fields as diverse as machine learning, oil exploration, scientific image processing, linear algebra, statistics, 3D reconstruction and many other. In the same time the complexity of a GPU has increased. A GPU has several streaming multiprocessors, each of which has multiple cores. For example, NVIDIA GeForce GTX 590 has dual GPUs, where each GPU has 16 streaming multiprocessors (SMs); each of these SMs has 32 cores, which gives a total of 1024 cores in the overall GTX 590 graphics card [2].

So GPUs now offer a compelling alternative to computer clusters for running large, distributed applications. However, the progress of GPU performance has slowed due to excessive power dissipation at GHz clock rates and diminishing returns in instruction-level parallelism [3]. Hence, application developers are increasingly shifting their algorithms to parallel computing architectures for practical processing times. In this aim, this paper proposes a different approach for creating a software architecture containing a set of abstract data types and associated pixel level operations executed in data parallel fashion.

2. Performing real-time image processing on parallel platforms

2.1. Definition of “real-time image processing” concept

Considering the need for real-time image/video processing and how this need can be met by exploiting the inherent parallelism in an algorithm, it becomes important to discuss what exactly is meant by the term “real-time”. From the literature, it can be derived that there are three main interpretations of the concept of “real-time” when describing image processing systems and algorithms, namely real-time in the perceptual sense, real-time in the software engineering sense, and real-time in the signal processing sense [4].

Real-time in the perceptual sense is used mainly to describe the interaction between a human and a computer device for a near instantaneous response of the device to an input by a human user. Let also note that “real-time” implies the idea of a maximum tolerable delay based on human perception of delay, which is essentially some sort of application-dependent bounded response time.

Real-time in software engineering sense refers to the case where missed real-time deadlines result in performance degradation rather than failure. Real-time in signal processing sense is based on the idea of completing processing in the time available between successive input samples. In the following the discussion is focused on the possibility to perform software implementation on a parallel processing platform of some primary image processing algorithms, corresponding to real-time in the software engineering sense.

2.2. Software operations involved in real time image processing

Traditionally, image/video processing operations have been classified into three main levels, namely low, intermediate, and high, where each successive level differs in its input/output data relationship [5].

Low-level operators take an image as their input and produce an image as their output, while intermediate-level operators take an image as their input and generate image attributes as their output, and finally high-level operators take image attributes as their inputs and interpret the attributes, usually producing some kind of knowledge-based control at their output.

One can hope that with an adequate task scheduling and a well-designed cluster of processors one can perform in real time low-level operations.

Low-level operations transform image data to image data. This means that such operators deal directly with image matrix data at the pixel level. Examples of such operations include color transformations, gamma correction, linear or nonlinear filtering, noise reduction, sharpness enhancement, frequency domain transformations, etc.

The ultimate goal of such operations is to either enhance image data, possibly to emphasize certain key features, preparing them for viewing by humans, or extract features for processing at the intermediate-level. These operations can be further classified into point, neighborhood (local), and global operations [6]. Point operations are the simplest of the low-level operations since a given input pixel is transformed into an output pixel, where the transformation does not depend on any of the pixels surrounding the input pixel. Such operations include arithmetic operations, logical operations, table lookups, threshold operations, etc. The inherent DLP in such operations is obvious, as depicted in Fig. 1 (a), where the point operation on the pixel shown in black needs to be performed across all the pixels in the input image. Local neighborhood operations are more complex than point operations in that the transformation from an input pixel to an output pixel depends on a neighborhood of the input pixel. Such operations include two-dimensional spatial convolution and filtering, smoothing, sharpening, image enhancement, etc.

Since each output pixel is some function of the input pixel and its neighbors, these operations require a large amount of computations. The inherent parallelism in such operations is illustrated in Fig. 1 (b), where the local neighborhood operation on the pixel shown in black needs to be performed across all the pixels in the input image.

Finally, global operations build upon neighborhood operations in which a single output pixel depends on every pixel in the input image [see Fig. 1 (c)].

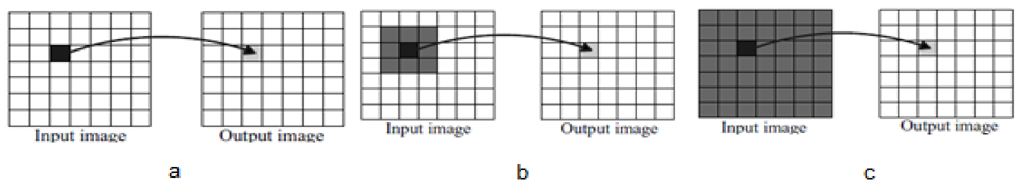

Fig.1. Parallelism in low-level image/video processing: a) point b) neighborhood c) global.

All low-level operations involve nested looping through all the pixels in an input image with the innermost loop applying a point, neighborhood, or global operator to obtain the pixels forming an output image. For this reason low-level operations are excellent candidates for exploiting DLP.

The higher degree operations are difficult to implement for real time execution. Intermediate-level operations transform image data to a slightly more abstract form of information by extracting certain attributes or features of interest from an image. This means that such operations also deal with the image at the pixel level, but a key difference is that the transformations involved cause a reduction in the amount of data from input to output. The goal by carrying out these operations (which include segmenting an image into regions/objects of interest, extracting edges, lines, contours, or other image attributes of interest such as statistical features) is to reduce the amount of data to form a set of features suitable for further high-level processing. Some intermediate-level operations are also data intensive with a regular processing structure, thus making them suitable candidates for exploiting DLP.

High-level operations interpret the abstract data from the intermediate-level, performing high level knowledge-based scene analysis on a reduced amount of data. These types of operations (for example recognition of objects) are usually characterized by control or branch-intensive operations. Thus, they are less data intensive and more inherently sequential rather than parallel. Due to their irregular structure and low-bandwidth requirements, such operations are suitable candidates for exploiting ILP, although their data-intensive portions usually include some form of matrix–vector operations that are suitable for exploiting DLP.

3. Principles of parallel platforms architecture design

3.1. Hardware Architecture Design

As discussed in the previous section, practical image/video processing systems include a diverse set of operations from structured, high-bandwidth, data-intensive, low-level and intermediate-level operations such as filtering and feature extraction, to irregular, low-bandwidth, control-intensive, high-level operations such as classification. Since the most resource demanding operations in terms of required computations and memory bandwidth involve low-level and intermediate level operations, considerable research has been devoted to developing hardware architectural features for eliminating bottlenecks within the image/video processing chain, freeing up more time for performing high-level interpretation operations. While the major focus has been on speeding up low-level and intermediate level operations, there have also been architectural developments to speed up high-level operations.

From the literature, one can see there are three major architectural features that are essential to any image/video processing system, namely single instruction multiple data (SIMD), very long instruction word (VLIW), and an efficient memory subsystem. The concept of SIMD processing is a key architectural feature found in one way or another in most modern real-time image/video processing systems. It embodies broadcasting a single instruction to multiple processors, which simultaneously execute the instruction on different portions of data in parallel, thus allowing more computations to be performed in a shorter time.

While SIMD can be used for exploiting DLP, VLIW can be used for exploiting instruction level parallelism (ILP) and thus for speeding up high-level operations. VLIW furnishes the ability to execute multiple instructions within one processor clock cycle, all running in parallel, hence allowing software-oriented pipelining of instructions by the programmer. Besides the fact that for VLIW to work properly there must be no dependencies among the data being operated on, the ability to execute more than one instruction per clock cycle is essential for image/video processing applications that require operations in the order of Giga operations per second.

An efficient memory subsystem is considered a crucial component of a real-time image/video processing system, especially for low-level and intermediate-level operations that require massive amounts of data transfer bandwidth as well as high-performance computation power. Concepts such as direct memory access (DMA) and internal versus external memory are important. DMA allows transferring of data within a system without burdening the processing unit with data transfers, so it is a well-known tool for hiding memory access latencies, especially for image data.

According to the algorithmic process complexity, there are different possible hardware implementation platforms that one can consider for the real-time implementation. For the selection of an appropriate hardware platform one must precise what are the important features of an image/video processing hardware platform and its advantages and disadvantages in order to be best suited for the real-time application under consideration.

There are two types of General Purpose Processors (GPP) on the market today, one geared toward non embedded applications such as desktop PCs and the other geared toward embedded applications. Desktop GPPs are extremely high-performance processors with highly parallel architectures, containing features that help to exploit ILP in control-intensive, high-level image/video operations. GPPs have been outfitted with the multilevel cache feature. This feature provides the potential of having low latency memory accesses for frequently used data. However, desktop GPPs are characterized by their large size, requiring a separate chip set for proper operation and communication with external memory and peripherals.

On the embedded front, there are also several GPPs available on the market today with high-performance general-purpose processing capability suitable for exploiting ILP coupled with low power consumption and SIMD-type extensions for moderately accelerating multimedia operations, enabling the exploitation of DLP for low-level and intermediate-level image/video processing operations.

Both embedded and desktop GPPs are supported by mature development tools and efficient compilers, allowing quick development cycles. While GPPs are quite powerful, they are neither created nor specialized to accelerate massively data parallel computations.

3.2. Software Architecture Design

While translating source code from a research development environment to a real-time environment is an involved task, it would be beneficial if the entire software system is well thought out ahead of time. Considering that real-time image/video processing systems usually consist of thousands of lines of code, proper design principles should be practiced from the start in order to ensure maintainability, extensibility, and flexibility in response to changes in the hardware or the algorithm.

One key method of dealing with this problem is to make the software design modular from the start, which involves abstracting out algorithmic details and creating standard interfaces or application programming interfaces (APIs) to provide easy switching among different specific implementations of an algorithm. Also beneficial is to create a hierarchical, layered architecture where standard interfaces exist between the upper layers and the hardware layer to allow ease in switching out different types of hardware so that if a hardware component is changed, only minor modifications to the upper layers will be needed.

It is important to mention also that in a real-time image/video processing system, certain tasks or procedures have strict real time deadlines, while other tasks have firm or soft real-time deadlines. In order to be able to manage the deadlines and ensure a smoothly running system, it is useful to utilize a real time operating system. Real-time operating systems allow the assignment of different levels of priorities to different tasks. With such an assignment capability, it becomes possible to assign higher priorities to hard real-time deadline tasks and lower priorities to other firm or soft real-time tasks. For portable embedded devices such as digital cameras, a real-time operating system can be used to free the upper layer application from managing the timing and scheduling of tasks, and handling file input/output operations. Therefore, a real-time operating system is an important key component of the software of any practical real-time image/video processing system since it can be used to guarantee meeting real-time deadlines and thus ensuring deterministic behavior to a certain extent.

4. Structural organization of the platform and jobs scheduling

4.1. Parallel platform model

The proposed model for the platform consists of P processor units. Each processor p_i has capacity $c_i > 0$, $i = 1,2,\ldots, P$. The capacity of a processor is defined as its speed relative to a reference processor with unit-capacity. We assume for the general case that $c_1 \le c_2 \le \ldots \le c_P$. The *total capacity* C of the system is defined as $C = \sum_{i=1}^{P} c_i$. A system is called *homogeneous* when $c_1 = c_2 \ldots = c_P$. The platform is conceived as a distributed system. Each machine is equipped with a single processor. In other words, we do not consider interconnections of multiprocessors. The main difference with multiprocessor systems is that in a distributed system, information about the system state is spread across the different processors. In many cases, migrating a job from one processor to another is very costly in terms of network bandwidth and service delay, and that the reason that we have considered for the beginning only the case of data parallelism for a homogenous system. The intention was to test the general case of image processing with both data and task parallelism, by developing a scheduling policy with two components [7]. The *global* scheduling policy decides to which processor an arriving job must be sent, and when to migrate some jobs. At each processor, the *local* scheduling policy decides when the processor serves which of the jobs present in its queue.

Jobs arrive at the system according to one or more interarrival-time processes. These processes determine the time between the arrivals of two consecutive jobs. The *arrival time* of job j is denoted by A_j. Once a job j is completed, it leaves the system at its *departure time* D_j.

The *response time* R_j of job j is defined as $R_j = D_j - A_j$. The *service time* S_j of job j is its response time on a unit-capacity processor serving no other jobs; by definition, the response time of a job with service time s on a processor with capacity c' is s/c'.

We define the *job set J*(*t*) *at time t* as the set of jobs present in the system at time *t*:

$$J(t) = \{ j \mid A_j \le t < D_j \}$$

For each job $j \in J(t)$, we define the *remaining work* $W_j^r(t)$ *at time t* as the time it would take to serve the job to completion on a unit-capacity processor. The *service rate* $\sigma_j^r(t)$ *of job j at time t* ($A_j \le t < D_j$) is defined as: $\sigma_j^r(t) = \lim_{\tau \to t} \frac{dW_j^r(\tau)}{d\tau}$. The *obtained share* $\omega_j^s(t)$ *of job j at time t* ($A_j \le t < D_j$) is defined as: $\omega_j^s(t) = \sigma_j^r(t) / C$. So, $\omega_j^s(t)$ is the fraction of the total system capacity *C* used to serve job *j*, but only if we assume that $W_j^r(t)$ is always a piecewise-linear, continuous function of *t*. Considering $W_j^r(A_j) = S_j$ and $W_j^r(D_j) = 0$ we have $\int_{A_j}^{D_j} \omega_j^s(t)dt = \int_{A_j}^{D_j} \sigma_j^r(t)dt = S_j / C \cdot$

One can define an upper bound on the sum of the obtained job shares of any set of jobs {1, … , J} as:

$$\omega_{\max}(t) = C^{-1} \sum_{i=1}^{\min(J,P)} c_i \tag{1}$$

Since equation (1) imposes upper bonds on the total share, the maximum obtainable total share at time *t* is defined as:

$$m^T(t) = c^{-1} \sum_{p=1}^{\min(|J(t)|,P)} c_p \tag{2}$$

In a similar manner, for a group *g* of processors (*g = 1, 2, ... , G*), the maximum obtainable group share at time *t* is defined as:

$$m^G(t) = c^{-1} \sum_{p=1}^{\min(|J_g(t)|,P)} c_p \tag{3}$$

4.2. Share-scheduling policy for parallel processing

Let consider the above mentioned model of a parallel processing platform with P processors, and assume that each job *j* has a weight w_{jp} on processor *p*, representing the fraction of time job *j* spends on processor *p*. We consider also that job *j* can switch between processors such that it appears to be served by more than one processor at the same time, at total service rate of $\sum_{p=1}^{P} w_{jp} c_p$ and the following conditions are respected:

$$0 \le w_{jp} \le 1, \text{for all j, p} ; \sum_j w_{jp} \le 1 \text{ for all p} ; \sum_p w_{jp} \le 1 \text{ for all j}$$

Under these assumptions, the proposed policy defined as job-priority multiprocessor sharing (JPMPS) can be applied without difference for both multiprocessors and distributed systems with free job migration.

In [8] is proven that under JPMPS, jobs *1, . . . , J (J ≥ 1)* can be given service rates $\sigma_1^J \ge \sigma_2^J \ge ... \ge \sigma_J^J$, *J > 0*, if and only if, for *j = 1, . . . , J*,

$$\sum_{k=1}^{j} \sigma_k^J \le \sum_{p=1}^{\min(k,P)} c_p \tag{4}$$

When we compare this result to the definition of $m^T(t)$ in (2) and $m^G(t)$ in (3) it is obvious that, under JPMPS, the set of jobs present in the system can always be provided with a share of $m^T(t)$ and also all jobs of group *g* can be provided with $m^G(t)$.

The objective of share scheduling is to provide groups with their feasible group shares. For each group one can define a constant share, the required group share r_g^G, as the fraction of the total system capacity *c* that group *g* is entitled to, with $\sum_g r_g^G = 1$. The required group shares are assumed to be constant over time and to be known in advance to the system. When the required group shares of two groups are the same, the system should treat them equally, and when one exceeds the other, the system should give preferential treatment to the group with the r_g^G. The required group share plays an important role in the definition of the feasible group share, because the feasible group share depends only on the required group share, the number of jobs present of the group and the average processor capacity.

We state the following three requirements for the definition of the feasible group share $f_g^G(t)$, *g = 1, . . .,G*:

1. $0 \le f_g^G(t) \le m_g^G(t)$
2. $f_g^G(t)$ depends only on $|J_g(t)|, r_g^G, P, c_i (i = 1,..., P)$
3. if $|J_g(t)| \ge P$ then $f_g^G(t) = r_g^G$

The first requirement means that we should not promise to a group more than the maximum we can provide.

The second requirement means that the feasible group share only depends on the number of jobs of the group and not on the coincidental presence of jobs of other groups (i.e. groups are promised a share of the system that does not depend on the activity of other groups).

The third requirement states that when the number of jobs in a group exceeds a threshold, the feasible group share equals the required group share. This threshold may be different for different groups, provided they have different required group shares.

From requirements 2 and 3, it follows that:

$$f_g^G(t) \le r_g^G \text{, and} \sum_{\mathrm{g}} f_g^G(t) \le 1 \qquad (5)$$

This means one can never promise more than the total system capacity to all groups together.

5. Real time image processing applications based on CUDA platforms

Parallel processing applications have specific processing requirements compared to sequential programming. Due to its complex architecture, the GPU meets most of them. While the CPU addresses the requests of a general purpose processor dealing with a large number of instructions and arbitrary operations (e.g. scheduling, transfers or computing), the GPU was conceived to execute just a few instructions, but really fast. The GPU was designed as a number of separate processing units that apply in parallel the same instruction set on different data points. Considering these facts and trying to explore the full performance of the GPU, an increased number of researching communities have focused their attention on the concept of general purpose computing, also known as “GPGPU”. GPUs became attractive essentially because they offer extensive resources for massive parallelism, high memory bandwidth and a general purpose instruction set including support for both, single and double – precision floating point.

Recognizing the value of GPUs for GPGPU, vendors have designed specific hardware and software support for developers to use the highly parallel GPU architecture with-out the need to proceed through the entire graphics pipeline. The NVIDIA’s solution is the CUDA platform.

CUDA stands for Compute Unified Device Architecture and is a parallel computing architecture which comes with a free software environment allowing developers to use C, C++, FORTRAN, OpenCL or DirectCompute as a high or low – level programming language. Actually, CUDA language can be seen as an extension to C based on a few easy to learn abstractions for parallel programming and math-coprocessor offload. The challenge is to develop application software that transparently scales its parallelism in order to leverage the increasing number of processor cores just as 3D graphics scale their parallelism to many core GPUs.

CUDA's parallel programming model was designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages. The minimal set of language extensions involves at least three key abstractions: a hierarchy of thread groups, barrier synchronization and shared memory.

While developing parallel applications CUDA programmers should consider the CPU and the GPU as a single entity. The actual system consists of a host (the traditional CPU) and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units (such as GPU). Therefore, a CUDA program is a collection of code sequences executed either by a host or a device. Usually, the code sequences that exhibit little or no data parallelism are implemented in host code, while those that exhibit rich amount of data parallelism are implemented in device code.

CUDA platform configuration allows simultaneous computation on both CPU and GPU without contention for memory resources. CUDA enabled GPUs have hundreds of cores able to run thousands of computing threads. CUDA architecture is built around a matrix of multithreaded Streaming Multiprocessors (SMs). Any SM may have up 8 scalar processors (SP) cores, a multithreaded instruction unit and shared memory. To manage parallel threads, NVIDIA implements SIMT (single-instruction, multiple-thread). The SM maps each thread to one SP core and the SIMT unit schedules threads in groups of 32. These groups are also known as warps. The SIMT unit selects a warp (that is ready to execute) and it launches the instruction to the active threads of the warp. A warp executes one instruction at a time, therefore all 32 threads of a warp must agree on their execution path to optimize the execution. Even if all threads from the same warp start together, at the same program address, they can branch and execute independently. If threads within a warp diverge because of a conditional branch, the warp serially executes each branch path, disabling threads that are not in the path. When all paths complete, the threads converge back to the same execution path. This situation may occur only within a warp; different warps will always be executed independently. Considering that a single instruction controls multiple processing elements, SIMT and SIMD (Single Instruction, Multiple Data) architectures are similar. Although, a key difference is that SIMT enables programmers to write thread-level parallel code for independent threads, as well as data-parallel code for coordinated threads. The SIMT behavior seems to be the key to substantial performance improvements in CUDA architecture, the same way the cached lines are the key to performance in traditional code. The programmer can ignore the SIMT behavior for an easier design process, but he should consider it in order to achieve top performance.

CUDA provides a hierarchy of thread groups, barrier synchronization and shared memory, three key abstractions that are able to offer a clear parallel structure to conventional C code. Multiple levels of threads, memory, and synchronization provide fine-grained data and thread parallelism, all these nested within coarse-grained data and task parallelism. These abstractions guide the programmer to partition the problem into coarse sub-problems that can be independently solved in parallel. The sub-problems can be split into smaller entities that will be the subject of parallel cooperative computing.

This means that threads are not always independent, sometimes they must cooperate in order to complete the task. Each of the threads that execute a function (usually defined by the programmer and known as a kernel) has a unique threadID accessible within the kernel. The kernels can be parallelized, hence they will be executed *n* times in parallel by *n* different CUDA threads, as opposed to "only once" like regular C functions. The programmer organizes these threads into a hierarchy of grids of thread blocks [9]. A thread block is a set of concurrent threads that can cooperate through barrier synchronization and shared access to a memory space. Blocks can be executed in any order, in parallel or in series. Furthermore, a set of blocks may be executed independently in which case they are seen as a grid.

As we previously pointed out, threads within a block can cooperate by sharing data through shared memory. To synchronize their execution and to coordinate memory access, the SM implements the CUDA__syncthreads() barrier. Mainly, this function acts as a barrier at which all threads within a block must wait before any of them is allowed to proceed. This guarantees that no thread can proceed until all participating threads have reached this point. Since threads in a block may share local memory and synchronize via barriers, they will reside on the same SM. The number of thread blocks can exceed the number of processors within a SM, thus it virtualizes the processing elements and gives the programmer the flexibility to parallelize at whatever granularity is most convenient. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the GPU. Therefore, the scheduling policy analyzed in Section 4 can be applied successfully for CUDA based applications.

6. Conclusion

This paper shows how to use parallelizable patterns, obtained for typical low level image processing operations, on the basis of a parallel processing platform model. Given the experimental results we are confident in that the proposed software architecture forms a powerful basis for automatic parallelization and optimization of a wide range of image processing applications.

The scheduling solution can be used for both multiprocessors (GPUs) and distributed (clusters) computing for real-time video processing. It is true that distributed computing has mainly been applied to applications in which data could be processed in non-real-time, but one can perform visual communication, if real-time constraints that give additional requirements to data processing in distributed computing are considered.

GPU-based developments in the field of real-time image/video processing are fairly new, but can be observed in typical examples including stereo depth map computation and subpixel motion estimation. The power of the GPU allowed the use of advanced features, including multiresolution matching, adaptive windowing, and cross-checking.

Regarding the potential of the parallel platform model for image processing, in the near future we will focus our attention on the improvement of the scheduling component, by using processor units with different processing capacities and also other service policy for the queue of jobs, trying to improve the performances by supporting the execution of a sequence of algorithms on the same block and by dynamical reconstruction of the post processed image.

R E F E R E N C E S

[1] *H. Hunter*, *J. A. Moreno*. New Look at Exploiting Data Parallelism in Embedded Systems, Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, pp. 159–169, **2003**.

[2] *NVidia*, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications, **2014**.

[3] *S. Mittal*, *J. S. Vetter*. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. ACM Computing Surveys, Vol. 47, Iss. 2, pp. 19.1-19.23, **2015**.

[4] *A. Bovik*, Handbook of Image & Video Processing, Elsevier, **2005**.

[5] *S. Kyo, S. Okazaki, and T. Arai*, An Integrated Memory Array Processor Architecture for Embedded Image Recognition Systems, Proc. Int. Symp. on Computer Architecture, pp. 134-145, **2005**.

[6] *C. Soviany*. Embedding Data and Task Parallelism in Image Processing Applications, Ph.D. Dissertation, Delft University of Technology, **2003**.

[7] *R. Dobrescu, D. Popescu, M. Nicolae, H. Humaila,* Real time dependable communication infrastructure for a collaborative groupware system, Proc. of the 1st Int. Conf. MEQAPS’09, vol.1, pp. 207-212, 2009.

[8] *J.F.C.M, de Jongh,* Share scheduling in distributed systems, PhD Thesis, Technische Universiteit Delft, **2002**.

[9] *Dobrescu, M. Dobrescu, S. Mocanu, S. Taralunga*, Client-Server Architecture for Parallel Image Processing, WSEAS Transactions on Signal Processing, Issue 9, vol. 2, pp. 1181-1188, **2006**.