J Real-Time Image Proc (2012) 7:267–279
DOI 10.1007/s11554-011-0194-9

ORIGINAL RESEARCH PAPER

Real-time GPU color-based segmentation of football players

Miguel Angel Montañés Laborda · Enrique F. Torres Moreno · Jesús Martínez del Rincón · José Elías Herrero Jaraba

Received: 29 March 2010 / Accepted: 5 January 2011 / Published online: 3 February 2011

Abstract In this paper, we propose a multi-camera application capable of processing high resolution images and extracting features based on colors patterns over graphic processing units (*GPU*). The goal is to work in real time under the uncontrolled environment of a sport event like a football match. Since football players are composed for diverse and complex color patterns, a Gaussian Mixture Models (*GMM*) is applied as segmentation paradigm, in order to analyze sport live images and video. Optimization techniques have also been applied over the C++ implementation using profiling tools focused on high performance. Time consuming tasks were implemented over NVIDIA's *CUDA* platform, and later restructured and enhanced, speeding up the whole process significantly. Our resulting code is around 4–11 times faster on a low cost *GPU* than a highly optimized C++ version on a central processing unit (CPU) over the same data. Real time has been obtained processing until 64 frames per second. An important conclusion derived from our study is the scalability of the application to the number of cores on the *GPU*.

Keywords Image processing · Color segmentation · Real time · *GPU* · *CUDA*

M. A. Montañés Laborda (✉) · E. F. Torres Moreno · J. E. Herrero Jaraba
Maria de Luna, 1, Saragossa, Spain
e-mail: mmonla@unizar.es

E. F. Torres Moreno
e-mail: enrique.torres@unizar.es

J. E. Herrero Jaraba
e-mail: jelias@unizar.es

J. Martínez del Rincón
Penrhyn Road, Kingston Upon Thames, Surrey KT1 2EE, UK
e-mail: Jesus.Martinezdelrincon@kingston.ac.uk

1 Introduction

Professional sport is an extremely competitive world. Mass media coverage has contributed to the popularity of many sports, increasing its importance in our current society due to the money and fame that it generates. In this environment, in which any assistance is welcome, video-based applications have proliferated. Video-based approaches have shown themselves to be an important tool for analysis of athletic performance, especially in collaborative sports, where many hours of manual work are required to analyze tactics and collaborative strategies. Computer-vision-based methods can provide help in automating many of those tasks.

Real-time image processing systems are specially relevant in Computer Vision. Any advanced image processing application requires a previous extraction of significant features. These features could be used in recognition or tracking systems for several applications. Our proposal is oriented to improve drastically the performance of image segmentation systems. Concretely, we focus on feature extraction and object classification based on those features, not only over pre-recorded video sequences but also from live video streaming.

Our method to extract those features consists in an image segmentation according to color information. Segmentation systems are usually a first stage inside an image processing framework. Thus, for instance, results generated by segmentation techniques can be used as input for a tracking algorithm. In the literature, it exists a broad variety of methods for a reliable segmentation of objects in an image, being the most interesting ones, those capable of dealing with objects composed of various colors [3, 6, 7, 9, 15, 21]. One of the most popular approaches consists in a Gaussian mixture model (*GMM*) in which every object can be represented by one or more Gaussians. This is because most objects are composed not only of an unique color but also of a mixture of different tones associated with an unique color or even of several different colors.

Although *GMM* is a successful and broadly used method for feature extraction, its computational cost is a strong handicap for real time applications. The spectacular evolution that CPUs experimented in the past has provided a tool for mitigating the problem. Nevertheless, the progressive slowdown during the last years has stopped this progression, whereas it has promoted parallel architectures, such as multi-core, as a solution for increasing the computational power.

This novel style of multi-core design and programming acquires even a more relevant position thanks to the last technological developments. Return of these advances are the newest Graphics Processing Units or *GPU* containing up to hundred of simple processor cores. The *GPU* architecture is optimized for massively parallel processing with peaks up to hundreds of GFLOPS. But their most interesting features of these devices is that they can also be harnessed for general computing in a modality known as general-propose *GPU* (*GP-GPU*) [24]. Recently, in order to take advantage of these high performance computing devices, some extensions to well-known programming languages have been generated, such as *CUDA C* [8]. This language is a set of parallel extensions of the C/C++ programming languages and it is able to interact with a special hardware interface built into all current NVIDIA *GPUs* [22, 26].

In the last few years, the amount of scientific application tested over *GP-GPU* has increased [5]. Although generally those researches [23, 27, 28] are focused on specific calculations, they provide an initial idea about the intrinsic potential of this new platform [20]. Particularly, in our field of interest, several studies probe this capability in modern *GPUs* [14]. Traditional methodologies have been implemented, such as pattern recognition algorithms based on textures [11], Gaussian mixture models [19] or image feature extraction techniques [29, 31]. All these examples give an idea of the increase of efficiency that can be achieved thanks to these devices.

In our research, we have developed an application which is able to detect football players in a video sequence. Once they are extracted from background, each player is classified into any of the teams. For classification purposes, a color-based method is employed based on Expectation Maximization for Gaussian Mixture Models [3, 19, 21]. Since one of our main objectives is to process multi-high-resolution cameras, detection and classification processes must be applied on real time in an extremely efficient manner. In order to achieve that, we have adapted and implemented those tasks over *GPU* platform taking advantage of its high parallel computational capability (Sect. 5).

The evaluation of our implementation has been made over a set of different low cost *GPUs* with 16, 32 and 64 cores to study the scalability of the implementation. These tests have also been run under different CPUs, to clarify as much as possible the real contribution of our implementation.

The outline of the paper is as follows. In Sect. 2, the hardware infrastructure is described. Section 3 introduces the stages that compose our methodology and discusses their computational cost. Section 4 compares the computational cost between a version in C++ using Microsoft Visual Studio compiler and a version highly optimized using Intel C++ compiler, running in a conventional multicore CPU. In Sect. 5, the parallelization methodology is introduced and a CUDA implementation is detailed. Section 6 presents a comparison between CPU and *GPU* results and its scalability. Finally, conclusions and future work are presented in Sect. 7.

2 Infrastructure

Our goal consists in the processing and classification of football players in video sequences provided from one or multiple cameras installed in a real football stadium. The minimum number of cameras required for covering a football field depends on several factors, such as camera resolution, angle of vision and height of installation. In our infrastructure, we propose a system composed of eight static high definition digital cameras (1,388 $\times$ 1,036) positioned on the roof around the stadium. Thus, we obtain a detailed coverage of the two goalkeeper areas (2 cameras for each one) as well as the rest of the field which is covered by other four cameras. It is important to remark the importance of a multi-camera representation, since overlapping cameras are crucial to solve occlusions, specially in conflictive areas. On the other hand, the more cameras you have, the more increase of computational cost. For this reason the number of eight cameras has been chosen, since we consider it is the minimum number to make viable the processing of the match: it ensures the coverage of a player by at least two cameras at any point of the pitch and with an acceptable resolution level.

All the cameras are linked by ethernet optical fiber and shielded twisted pair with a computer set which has to process the received data and combine results. A distribution schema and its connection with the computing system can be seen in Fig. 2 and a overlapped zones schema can be seen in Fig. 1.

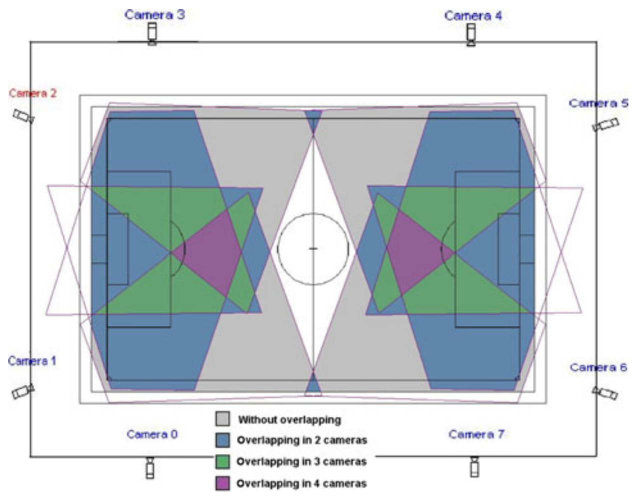


Fig. 1 Camera distribution on the roof

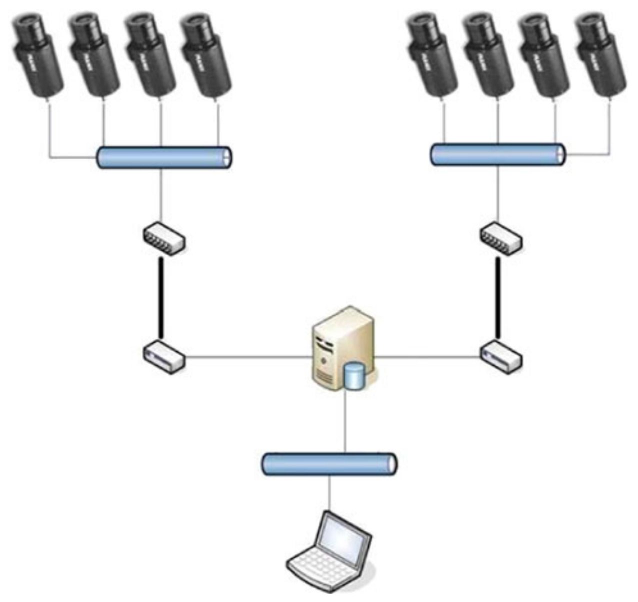


Fig. 2 Infrastructure schema

3 Methodology

Our system is composed of multiple and identical high definition cameras with a resolution 1,388 × 1,036. As requirement, this application must perform the capture of eight images per second, the processing of all frames (including extraction and classification processes), visualization tasks, communication and, finally, tracking.

The proposed classification algorithm can be decomposed into a set of steps. Most of them should be done per frame and per camera. The steps and input data that they require are described at following Sect. 3.1 and in Fig. 3 the processing flow per camera is detailed. Output generated from previous stages can be used as input for a tracking algorithm in order to ensure the temporal

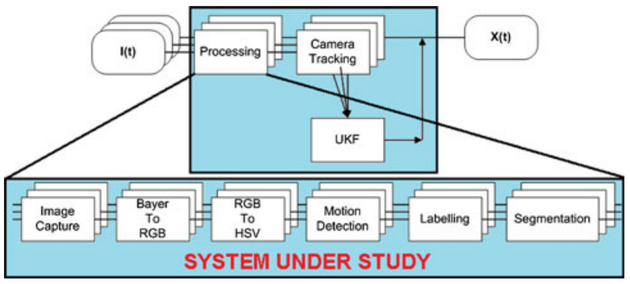


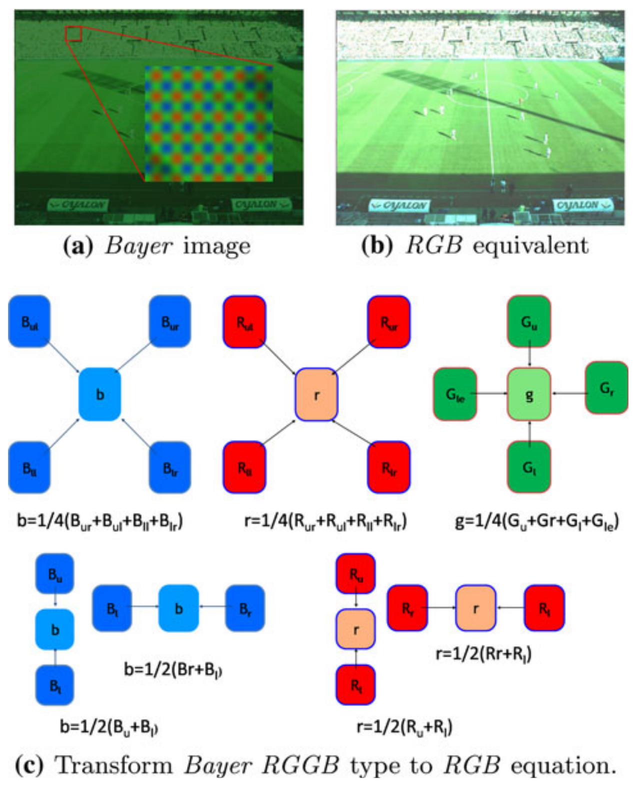
Fig. 3 Processing schema

coherence. Several different options can be found in the literature [4, 12, 13, 18]. Although it is out of the scope of this paper, a *Multi-Camera Uncensted Kalman Filter (MCUKF)* [16] has been used to demonstrate the global feasibility.

Empirical experiments allow us to conclude: A successful tracking can be obtained with a processing frame rate between 8 and 15 per each camera, i.e., a processing time per image per camera around 66–125 ms, and to cover the whole football field, at least eight cameras are needed to obtain enough overlapping. As conclusion, this requirement allows us to define the concept of real time and the scalability of the processing kernel for our particular needs.

3.1 Independent processing per camera

- Image capture: at this stage, images are retrieved on demand from each camera.
- Color space transformation from Bayer to RGB: high-resolution cameras usually provide images in raw format (also called Bayer-type RGGB [2]), i.e. 8 bits per pixels for color codification. This format only needs a third of a conventional RGB image size, but it is not suitable for our post-processing since all the channels are mixed. To obtain a RGB image, we need an intermediate transformation process called BayerToRGB, which is depicted in Fig. 4. The procedure to generate three channels from a Bayer sequence RGGB needs a particular calculation for every channel. RGB values which match up in the RGGB sequence are mapped directly, while other channels are calculated as an arithmetic mean of all neighbors corresponding to the same channel. For example, RGB value for a red position can be reconstructed as:
- *R* value is copied as the same value.
- *G* value is, as shown in Fig. 4c), the average of the four-neighbor pixels: left, right, up and low pixels.
- *B* value is, as shown in Fig. 4c), the average of the four-neighbor pixels in diagonal: up-left, up-right low-left, low-right pixels.



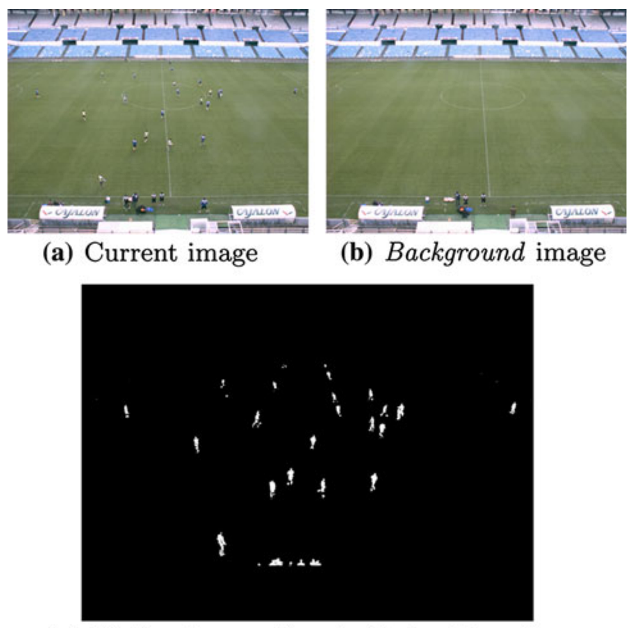
(a) *Bayer* image (b) *RGB* equivalent

(c) Transform *Bayer RGGB* type to *RGB* equation.

Fig. 4 Raw image (*Bayer*), *RGB* equivalent and transform equation

- Color space conversion *RGB* to *HSV*: under variable illumination conditions, better classification results can be obtained by applying a transformation in the color space [30]. Instead of RGB, HSV (Huge, Saturation, Value) has shown a better accuracy (Eqs. 1–3).
- Motion detection: it consists in a thresholded subtraction between the current image (Fig. 5a) of every camera and a pre-generated image of the scenario, called *background* (Fig. 5b). Process is shown in Fig. 5c. Motion detection image contains the dynamic areas, which will be used for posterior processing like distracter removal.

$$H = \begin{cases} \text{No defined} & \text{if} \quad MAX = MIN \\ 60^\circ * \dfrac{G-B}{MAX-MIN} + 0^\circ & \text{if} \quad MAX = R \\ & \text{and} \quad G \geq B \\ 60^\circ * \dfrac{G-B}{MAX-MIN} + 360^\circ & \text{if} \quad MAX = R \\ & \text{and} \quad G < B \\ 60^\circ * \dfrac{B-R}{MAX-MIN} + 120^\circ & \text{if} \quad MAX = G \\ 60^\circ * \dfrac{R-G}{MAX-MIN} + 240^\circ & \text{if} \quad MAX = B \end{cases} \tag{1}$$


(a) Current image (b) *Background* image

(c) Motion Image, thresholded subtraction

Fig. 5 Current image, *background* image and subtraction result image

$$S = \begin{cases} 0 & \text{if} \quad MAX = 0 \\ 1 - \dfrac{MIN}{MAX}, & \text{otherwise} \end{cases} \tag{2}$$

$$V = MAX \tag{3}$$

- Blob labeling: it is the algorithm that seeks connected areas, called *blobs*, in the resulting image of the previous step. By grouping pixels into *blobs* and assigning a common label we simplify the posterior tracking stage.
- Color segmentation: this procedure tackles the problem of identifying different areas of the image. *GMM* (*Gaussian Mixture Model*) has been chosen as paradigm, which implies a preliminar training by extracting color features from regions of interest. Thanks to this technique, a distinction into three groups is obtained: *player of team 1, player of team 2 and noise from the background.*

3.2 Gaussian mixture method for image segmentation

In collaborative sport applications, feature extraction and classification, although a difficult task, have an important advantage in comparison with more general approaches like video surveillance. It is known a priori that both teams, as well as background, are defined by clear and distinctive color patterns in their clothing. These color patterns can be easily modeled by parametric methods.

GMM is a method that allows a reliable object modeling and image classification even in presence of complex targets, which can be composed of multimodal appearance distributions. Since it is a parametric technique, it needs an off-line training phase to calculate those parameters. Training results are used afterwards in classification (*On-line* stage).

The simplest technique to model the appearance coefficients consists in assuming the target as a monochrome region and modeling it as a Gaussian using only two parameters: mean μ and covariance σ^2. Although this assumption limits the generality of the methodology, it can be easily extended by dividing the target into a predefined set of monochrome regions [25].

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{-(x-\mu)^2/2\sigma^2} \tag{4}$$

$$p(x) = \sum_{i=1}^{N} w_i \frac{1}{\sqrt{2\pi}\sigma_i} \mathrm{e}^{-(x-\mu_i)^2/2\sigma_i^2} \tag{5}$$

$$\tilde{s} = (\frac{s_1 - \bar{X}^1}{\sigma_1}, \ldots, \frac{s_m - \bar{X}^m}{\sigma_m}) \tag{6}$$

$$C_i^s = \sum_{j=1}^{m} \tilde{s}_j a_j^s \tag{7}$$

$$Prob_{\text{team0}} = \frac{1}{\frac{1+\text{distance}_{\text{team0}}}{\text{distance}_{\text{min}}}} \tag{8}$$

Both off-line training and on-line classification are composed of different phases:

1. Off-line computing
   - Sample selection: supervised sample selection for every group (*team 1, team 2* and *background*).
   - Parameter tuning: a crucial issue is the adequate number of Gaussians used to model every group. Given the special characteristics of several collaborative sports, like a football match, where colors are well-defined, but where the video compression can generated halos around the players, a deep study was made in order to optimize it as much as possible (Eqs. 4, 5).
   - Training: Expectation Maximization (EM) algorithm using Fuzzy C-Means as initialization [3] provides final model.
2. On-line computing
   - Classification: In this step, every pixel is classified into one of the different groups. For this, the distance between the pixel candidate and the different model of every group is computed (Eqs. 6, 7) and a final decision based on minimum distance are taken (Eq. 8). In addition, the membership degree to every group is computed inside a probabilistic framework giving, as result, probability images [10] that can be used to improve the tracking quality based on stochastic approaches.

As the offline stage is only applied once at the beginning of the match and under human supervision, it can be considered out of the real-time system and, therefore, has been implemented over CPU. However, this process is also amenable to be implemented using *GPU*, as it was demonstrated in [19], obtaining excellent results. Furthermore, and due to lighting conditions changes over the game, color models need to be updated every 2 or 3 min. This update does not require manual annotation at all, since a random sample of the classified pixels are feedback to the model for its update. Thus, models updating has to be implemented in real time and its computational cost has been taken into account in this paper.

For the selection of optimum parameters of the Gaussian mixture during the training, different experiments have been performed, as stated in Section 4.2 of [10]. For our application, two different color spaces were created, one for modeling the players of both teams and one to model the background. Likewise, it was decided to consider 2 Gaussians for each model, i.e., the model of each team consists of 2 Gaussians for each team and 2 additional Gaussians to model the background. These number are not arbitrary: whereas two Gaussians per team permits to model t-shirt and shorts independently, two Gaussians for the background enables to capture the variability introduced by shadows and saturated areas of the pitch. The reader could argue that many sport equipments contain more complex color patterns, such as vertical strips, but in the reality, the distance to the camera mixes that patterns into a single one given the current technology of HD cameras. In the same way, shadows or saturated areas could be modeled as a single model in an appropriated color space such as HSV. However, this is plausible only for an optimal setup of the camera parameter, which is not practical and evolves during the game.

In the classification stage (*on-line computing*), a certain number of mathematical operations are performed per pixel (Eqs. 6–8). The results depend on the pixel values *HSV* and the color models. As the maximum number of possible combinations of HSV values is not large (maximum 256 × 256 × 256 values) and models do not change often, an optimization in both CPU and *GPU* implementations is the use of *Look-up tables or LUTs*. Those functions with a clear and repetitive pattern, such as color classification, can be replaced for a storage in memory of every possible result for any input combination. This resulting matrix is called segmentation *Look-up Table* (*LUT*) and there is one per camera. When the color models change, the LUT is

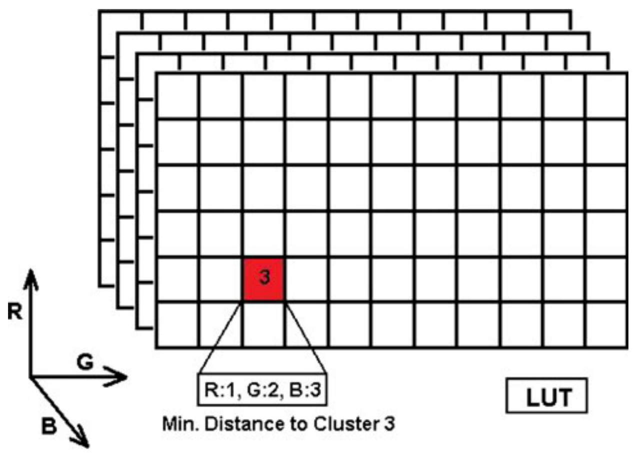


Fig. 6 Calculation of the segmentation value for color [H,S,V] = [1, 2, 3]. These data are stored in the segmentation Look-up Table

re-calculated for every possible HSV values. An example is depicted in Fig. 6. For every HSV value, the classification result is pre-computed and stored in the LUT. After its generation, the expensive calculation is replaced for a memory access to the right memory slot, which implies a substantial boost in efficiency. For example, the calculation result for a pixel HSV with values [H, S, V] = [1, 2, 3], is stored in row 1, column 2 and plain 3 as Fig. 6. The more complex the operation is, the more efficient this technique proves itself.

3.3 Performance evaluation

All image processing operations described in this section have been implemented in the corresponding CPU and GPU versions. For validating them, results were compared with a prototype modeled in Matlab, confirming that insignificant differences are only due to typical rounding errors.

In order to check the performance improvement that our implementation achieves, we have tested the algorithm over different types of processors and *GPUs*. Thus, four different types of PCs are available: Core 2 Duo 2.2 GHz 3GB Ram, Core 2 Duo 2.4 GHz 3.5 GB Ram, Core 2 Quad 2.83 GHz 3 GB Ram, and Core i7 Quad 2.66 GHz 4GB Ram. These equipments are close to the average current processors, giving us a significant sampling of the market. On the other hand, four different *GPUs* have been tested too: *GeForce 8600M GS, Quadro FX 1600M and Quadro FX 1800*. All of them can be considered low-cost *GPUs* containing 16, 32 and 64 cores, respectively. The fourth *GPU*, GTX260 with 216 cores, has been chosen to confirm the tendency.

For every possible combination of both platforms (CPUs and *GPUs*), a scalability study was made. A scalability study aims to assess the performance of our algorithm as a function of the number of images, the number of cameras or the computational power. To this end, we have processed the algorithms on several computers as it is shown in Table 1.

Table 1 Different types of CPUs and GPUs for testing

	Micro	GHz	nVidia	Cores	Bandwidth (GB/s)
PC1	Core 2 T7500	2.2	Geforce 8600M GS	16	6.7
PC2	Core 2 T8900	2.4	Quadro FX 1600M	32	11.2
PC3	Core 2 Quad	2.83	Quadro FX 1800	64	38.4
PC4	Core i7 Quad	2.66	Geforce GTX260	216	111.9

In the next section the implementation on C++ and CPU optimizations are described. Section 5 does the same for the implementation on *GPU* and in the last section, a scalability test is performed.

4 CPU implementation

Our first implementation of the algorithm was made in C++ language running under Windows. Once the accuracy of the results were validated with a Matlab prototype, a set of optimizations was included to obtain an improved C++ version.

For this optimization process, performance analysis tools, such as *Intel VTune Performance Analyzer* [17] were applied to identify the possible *hotspots*. This tool aimed at increasing performance, as well as the location of hotspots, allowing us to perform a deep analysis of them. Thus, *VTune* lets us detect, re-code and optimize our implementation, improving the performance substantially.

A comparative studio between the default *Microsoft Visual Studio compiler* and *Intel C++* was made for our application, showing that the usage of this last one was always beneficial with a general speedup of almost 4*x*. Full optimization and specific architecture compilation flags are both used in this implementation. These specific flags perform aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, loop blocking to allow more efficient use of cache and additional data prefetching.

Intensive use of *SIMD* and code modifications have been also done to allow the compiler to automatically apply *SIMD* instructions. Special care has been taken in the alignment of data in memory, and *vector* and *simd pragmas* has been used. Classical Code Optimizations [1] as *Loop-invariant code motion*, *Strength reduction*, and *Arithmetic pointers* have been used to clear loops. Compiler generated code has been analyzed following the compiler High level

Table 2 Time comparison between optimized C++ using Microsoft Visual Studio [MVCC] and optimized C++ using Intel C++ compiler [ICC], running in PC3

Stages in PC3	MVCC (ms)	ICC (ms)	Speed-up
Conversion BayerToRGB	71.62	15.7	4.56
Conversion RGBToHSV	71.58	35.61	2.01
Motion detection	31.67	9.83	3.22
Segmentation	297.5	90.69	3.28

Loop Optimizations (HLO) and vectorization reports (/Qopt-report), Vtune, and in some cases studying the generated assembler code and comparing performance with a not-vectorized version.

As a result, we obtain the differences between an optimized single threaded implementation in C++ using Microsoft Visual Studio compiler [MVCC] versus the same code compiled with Intel C++ compiler [ICC]. Results are depicted in Table 1 (obtained using PC3 described in Sect. 3.3). As can be seen, there are stages with low speed-up (like *RGBToHSV*), while *Conversion BayerToRGB* get a boosts in performance of 4.56×. The main gain comes from *Segmentation* that goes from 297.5 down to 90.69 ms.

In Table 2 it is shown that confronting [MVCC] and [ICC] implementations a big difference in performance exists just by compiling the code. The results prove that, as expected, using an optimizing compiler increases performance considerably.

These measurements have been obtained using the evaluation metric shown in Eq. 9, where sp_{fi} is the speed-up for stage i, $t_{fi,\mathrm{mvcc}}$ is the execution time of stage i optimized using *Visual Studio* and $t_{fi,\mathrm{icc}}$ the execution time of stage i optimized using *Intel C++*.

$$sp_{fi} = \frac{t_{fi,\mathrm{mvcc}}}{t_{fi,\mathrm{icc}}} \tag{9}$$

Another metric usually employed to evaluate the computing capability of a real-time oriented system is the processing rate or *rate*. Rate measures how many frames are processed per second. Rate equation can be described as follows:

$$\mathrm{Rate}\,(\mathrm{fps}) = \frac{1{,}000\,(\mathrm{ms})}{t_{\mathrm{total}}\,(\mathrm{ms})}\,(\mathrm{fps}) \tag{10}$$

Using [MVCC] implementation, the processing rate would be around 2.11 frames per second, while if [ICC] optimization is used, rate increases around 6.58 fps. We should remember that, as discussed in the introduction, a minimum rate between 8 and 15 fps is necessary for the correct operation of the subsequent tracking stage.

Table 3 Comparison of the processing time of each stage between an implementation for a single frame and one for four frames using a machine with four cores (*PC3*)

Stage	Time (ms)		Increment (%)
	CPU3 (single thread)	CPU3 (four instances)	
BayerToRGB	15.7	16.77	4.84
RGBToHSV	35.61	36.00	1.1
Motion detection	9.83	10.30	4.78
Segmentation	85.39	87.75	2.76
Total	146.53	150.82	2.92

$$\begin{aligned} &\mathrm{Rate}_{\mathrm{Visual}}\,(\mathrm{fps}) = \frac{1{,}000\,\mathrm{ms}}{(71.62 + 71.58 + 31.67 + 297.5)} \\ &\Rightarrow \mathrm{Rate}_{\mathrm{Visual}}\,(\mathrm{fps}) = 2.11\,\mathrm{fps} \\ &\Rightarrow \mathrm{Rate}_{\mathrm{Intel}}\,(\mathrm{fps}) = 6.58\,\mathrm{fps} \end{aligned} \tag{11}$$

In a multicore processor we could have more than one core doing image processing. As image processing is composed of many pipelined stages, we could assign each stage to a different thread or we could have many cores working on the same frame. Due to load balancing problems between threads and the added synchronization and communication, we found that it was much better to have each core working on a different frame (from the same camera or from another camera).

Image processing is clearly CPU bound, but as the different cores share the last level cache and the memory bandwidth, we expect a certain performance penalty. We have run multiple instances over different frames to observe the effect on each of the stages. The results are presented in Table 3 on a given run of four threads over the four cores of *PC3*.

Data in Table 3 show that, while in the single thread implementation we are able to process around 6.58 fps, running one instance per core we reach about 25 fps, so it follows that there is a minimal overhead for each stage at around 2.5 % for this particular execution. *Conversion BayerToRGB* is the stage that more variability supports with a 4.84 % penalty due mainly to the increased L3 cache miss ratio.

5 *GPU* implementation

The hardware architecture of a system with a *GPU* can be seen in Fig. 7. A *GPU* is a hardware device connected to the main system through a fast bus, second-generation PCI Express currently. It has some very specific processing features regarding the current CPUs.

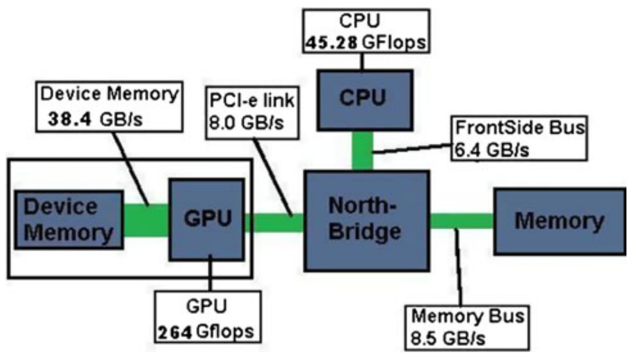


Fig. 7 Hardware architecture of a system with *GPU*

Specifically, the features that make *GPUs* specially powerful in massively parallel computing are:

– Hardware composed of several computing functional units and several multicores.
– In single precision floating point, a *GPU* can reach up to 500 Gflops owed to the 30–50 Gflops of conventional CPUs.
– High bandwidth for the internal memory up to one order of magnitude higher than the bandwidth of a CPU and system memory (up 111.9 GB/s in *GPU4*).
– In order to take advantage of such high bandwidth, *GPUs* allow several memory access operations to run simultaneously.
– *GPU* use *Single Instruction Multiple Thread* (SIMT) paradigm. This specific execution allows and needs many independent and simultaneous active threads that execute the same instructions over different data. All of them running as an unique kernel.

Below, a brief introduction to the main techniques in CUDA optimizations are described attending *GPU* characteristics and SIMT paradigm. Next, a preliminary study of our application is needed taking in mind these techniques as well as different criteria such as computational cost or massively parallel computing redesign. Finally, the optimized results for CPU and *GPU* implementations are shown and discussed.

5.1 Techniques for optimizing *GPU* code

Several techniques are at our disposal for an optimum use of *GPU* capacities according to recommended methodologies [22, 23, 26, 27]. Across all the stages these techniques have been evaluated. A *GPU* is a device designed for highly parallel computation having a very high number of functional units and a high memory bandwidth. Therefore, the main techniques for increasing performance are based on keeping up the occupation of functional units (known as occupancy), maximizing the use of effective bandwidth to memory (using techniques like coalescence) and minimizing branch divergency.

Occupancy: Occupancy is defined as the number of threads assigned to each processor. Maintaining a high occupancy in the *GPU* is important in order to mask the high latency of memory accesses. It can be achieved by means of three different ways: taking care with data-independent instructions, maintaining the number of registers per thread as low as possible and/or obtaining the best compromise *occupancy-shared memory size per thread*. Therefore, it is important to fully exploit the parallelism available in the application.

Coalescence: Coalescence is a technique for optimizing memory accesses. Memory accesses from different threads can be merged into a single access to the device memory if the required conditions are fulfilled [8]. This fusion process is known as coalescence and it is defined as a mean to gather several simultaneous memory accesses in parallel. It is promoting during the global memory accesses and it consists in a mechanism that fuses into an unique operation all the read/write accesses from the running threads in the current active block. *GPUs* have specific hardware that detects and makes this fusion, hiding the high latency of threads accessing to local or global memory when cache is not available.

Divergency: In the *SIMT* paradigm implementation of CUDA *GPUs*, high performance is obtained when all the thread in the same active block are executing identical instruction. In conditional execution code (i.e. conditional branches) several threads could take different paths. The result could be the serialized executions of diverging threads within a block, and therefore, increasing the cost for every divergent thread.

In order to evaluate and achieve high performance over *GPU*, several tools have been used to refine code: *CUDA Visual Profiler, CUDA Occupancy Calculator*, and *Decompiler*. This last one is a tool for disassemble code generated by a *CUDA* project. It provides the exact register mapping of the *GPU*, so bottlenecks in terms of number of registers used by the kernel can be checked. We used a specific decompiler named *decuda* that is available at http://www.wiki.github.com/laanwj/decuda/ [32]

5.2 Preliminary study

In this section, the adequacy of each stage to be implemented as a *GPU* kernel has been analyzed. Stages are independently implemented in different kernels in order to check their behavior using *GPU* paradigm. This test has been performed over *PC3* and the results are presented below.

Conversion BayerTo*RGB*: this stage requires, for every pixel, access to the neighbor pixels in order to calculate the resulting RGB. The processing is made per pixel independently, although the final result also depends on the adjacent input values such as Fig. 4 shown. Therefore, there is no coalescence in reading or writing, it has a high grade of divergency (each pixel is computed in a different way) and because it is the first stage, it supports the driver overhead (data has to be send to the *GPU*). Resulting RGB data are saved in memory as planar form to take advantage of coalescence in the following stages. Evaluation: suitable. Computational cost: 19.47 ms (≈11.45%).

Conversion *RGB*To*HSV*: in the same way as the previous stage, processing is pixelwise but there is no data dependency regarding the neighbor pixels. There is no divergence and as RGB data is kept in memory in planar form accesses are fully coalesced. Evaluation: suitable. Computational cost: 5.68 ms (≈3.31%).

Motion detection: Since it is basically a pixelwise subtraction, there is not dependency. As in the previous stage, Motion detection has a high coalescence degree and there is no divergence. Evaluation: suitable. Computational cost: 7.2 ms (≈4.23%).

Color segmentation: Segmentation consists basically in 2 substages, *blob labeling* and *color classification*. We are going to study them independently.

- **Blob Labeling:** this algorithm searches for connected zones in the image. The nature of the connectivity search produces a strong dependency among neighbors. There is not a simple parallel solution and a new algorithm should be developed to take advantage of the available features. We have tried many different algorithms and implementations. The more parallel code is, the more synchronization between *CUDA* blocks is needed, so more performance lost. Evaluation: not suitable. Computational cost: 93.34 ms (≈54.88%).
- **Color classification:** it is also a good candidate to be implemented on *GPU* as computation does not have dependencies with the neighbors and it implies a substantial part of the total time in the CPU implementation. It can be decomposed into three substages: resulting image calculation by consulting the corresponding *LUT* entry, *LUT* update for the next frame and noise filtering by morphological operators. The coalescence ratio for reading is low because *LUT* accesses are not regular. Divergence is minimal or none. Again, since this is the final stage, it supports the driver overhead of returning data results to the CPU. Evaluation: suitable. Computational cost: 44.37 ms (≈26.09%).

As a summary, main characteristics of every stage are shown in Table 4. The *CUDA* implementation was tested over PC3, obtaining the results shown in Fig. 8. We can conclude:

Table 4 Study of main parameters to improve the performance in every stages

Stage	Occ	Coal	Div	DO
BayerToRGB	66	$\bar{R}/\bar{W}$	High	Yes
Motion detection	100	*R*/*W*	Not	Not
RGBToHSV	100	*R*/*W*	Not	Not
Classification	66–100	$\bar{R}/\bar{W}$	Low or none	Yes

Occ Occupancy, *Coal* coalescence, *Div* divergence, *DO* driver overload, *R*/*W* Coalescence in read and write, $\bar{R}/\bar{W}$ non-coalescence in read and write

- Most stages are performed per pixel, so there is plenty of parallelism. Consecutive stages could be grouped and executed invoking a single kernel, reducing driver and synchronization overheads.
- *Motion Detection* and *Conversion RGBToHSV* stages prove a good behavior when they are implemented over *CUDA*. When comparing the CPU and the GPU implementation, times goes from 9.83 to 7.20 ms and from 35.61 to 5.68 ms, respectively.
- In spite of pixelwise calculation, *Conversion BayerToRGB* stage presents several dependencies in its data and divergence in the operations. *CUDA* implementation has to be carefully studied because time is higher in the *CUDA* implementation (19.47 vs. 15.7 ms in the CPU).
- *Labeling* is not parallelizable and our designed algorithm for *GPU* has a deficient behavior. Its computation time has increased almost 20×.
- *CUDA* implementation of the *classification* stage presents a significant improvement in performance, representing around 58% of the total time (if we do not account labeling).

The critical design phase is the labeling computing, since it is not parallelizable. The CPU version is much faster than the GPU version, as observed in Fig. 8, so an hybrid implementation of the *segmentation* stage could be implemented with *Labeling* done in the CPU. It is worth to take special care in aspects as kernel context switch or data transfer with CPU, avoiding unnecessary waste of time as they needs to access the *GPU* driver to complete the operation. The computational cost of transferring data *CPU* ⇒ *GPU or GPU* ⇒ *CPU* is around 7.19 ms. Three solutions have been studied:

- **Option 1:** All the stages are run over *GPU*: Labeling allows identifying active areas in the image, reducing the segmentation to those areas and making unnecessary segmenting the rest of the image. Total computational

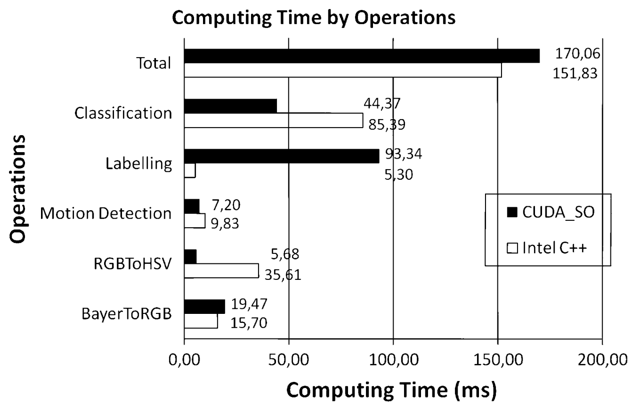


Fig. 8 Computational cost for [ICC] and *CUDA* (over PC3) implementations

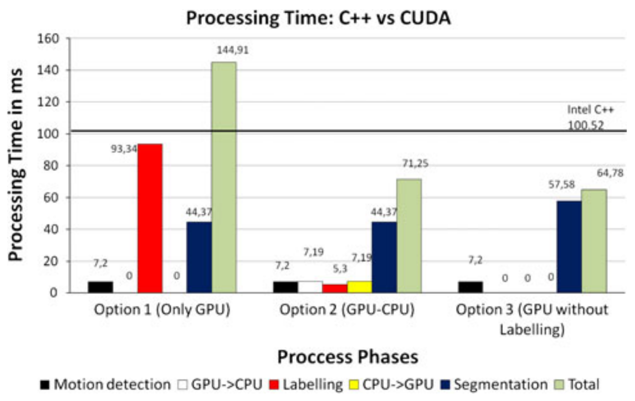


Fig. 9 Comparison: First *CUDA* (over PC3) implementation versus optimized C++

cost would be $T_{\text{total}_1} = T_p + t_{e_{gpu}} + t_{s_{blob}}$, where T_p is the time due to the pre-labeling stages, $t_{e_{gpu}}$ is the labeling cost in *GPU* and $t_{s_{blob}}$ is the segmentation cost on the active areas.

– **Option 2:** Previous stages to labeling are run on *GPU*, results are transferred to the host, which runs the labeling and returns the result to the *GPU*, where the segmentation is done on the active areas. $T_{\text{total}_2} = T_p + t_{\text{totaltrans}} + t_{e_{cpu}} + t_{\text{totaltrans}} + t_{s_{blob}}$, being $t_{\text{total}trans}$ the transference cost + kernel commutation cost + driver access cost.
– **Option 3:** Classification is applied to the whole image and not only over active areas. $T_{total_3} = T_p + t_{s_{image}}$. In this hybrid solution, labeling and final segmentation are relegated to CPU because its performance for these stages is more efficient than the corresponding over *GPU*. So one GPU kernel is invoked for processing all the pixelwise operations (from Bayer to classification) and then the CPU ends with the segmentation stage.

Previous options have been tested and results are shown in Fig. 9 over PC3. By minimizing the computational cost T_{total_1}, T_{total_2} and T_{total_3}, the optimum decision can be taken. As Fig. 9 shows, option 3 provides the optimum solution (64.78 ms) in comparison with the other alternatives whose costs are 144.91 and 71.25 ms. Option 1 is even more expensive than [ICC] implementation whose processing time is about 100.52 ms. Because the extra data transfers and the kernel context switching, option 2 is worse than option 3 although the whole image is classified in this last one.

In the light of previous results, we can conclude that *Blob Labeling* is not efficient for parallel computing and, in case of necessity for posterior stages such as tracking or distracter removal (football field lines), must be relegated to the CPU. Taking this decision as a new starting point, the next step consists in the optimization of all the stages.

Table 5 Comparison among CPU3 (single thread) and *GPU3* implementations

Stage	Time (ms)		Speed-up
	CPU3	GPU3	
BayerToRGB	15.7	16.48	0.95
RGBToHSV	35.61	2.57	13.85
Motion detection	9.83	1.86	5.27
Labeling	5.3	Not used	
Classification	85.39	23.63	3.61

5.3 Results

The preliminary study of the GPU execution concludes that on-line processing are composed of four stages (*BayerToRGB conversion, RGBToHSV conversion, Motion detection*, and *classification*), all of them are done per pixel. In addition, our implementation over *GPU* consists of an unique kernel, avoiding thus the extra time introduced by context changes or driver overload. This kernel receives frame data and runs the four pixelwise processes, and ends transferring the resulting data from the classification to the CPU.

A comparison between implementations on PC3 over the Intel C++ [CPU3] and over the CUDA [GPU3] applying all the optimizations is shown in Table 5.

– Since transfer time is a non-negligible limitation, a detailed study for minimizing the number of data transfer operations and kernels invocations has to be done
– *BayerToRGB* performance accounts for the driver overhead and its time is worse than the CPU implementation.
– *Motion Detection* has a good behavior since processing is pixelwise. High speed-up has been obtained, being 5.27 times faster.

– *Conversion RGBToHSV* stage also achieves high speed-up. This computing is boosted 13.85×.
– Finally, *Classification*, the most expensive stage, has achieved an speed-up of 3.61×, being comparable in time to other stages like *Conversion BayerToRGB*.
– Finally, the optimized version is 42.96% better than the first implementation. The gain comes mainly from the optimized version of *classification*. *BayerToRGB* gets almost no improvement because it supports the data transfer and driver overhead.

6 Scalability test

The performance of a GPU system is mainly determined by the number of cores and the memory bandwidth. To verify this, we have selected different systems with different resources (shown in Table 1) to test the performance. The aim is to study the cost evolution per stage and globally. The first 3 GPUs have been chosen with a consistent growing criterion in the number of *GPU* cores (16, 32 and 64). Memory bandwidth almost doubles from GPU1 to GPU2, and GPU3 has almost six times more than GPU1. The fourth *GPU*, with 216 cores and 112 GB/s, is chosen to confirm the tendency showed in the previous tests.

Two comparative analyses have been done. The first one, at the stage level, evaluating the time cost for every stage for each GPU (Fig. 10). The second one, comparing the global performance of the application using the four different CPUs against the GPUs measured in frames per second fps (Fig. 11).

Analyzing at the stage level (Fig. 10), it is important to note that improvement increase with *GPU* power, almost always proportional to the number of cores. The only exceptions are the *conversion BayerToRGB* and *Classification* stages, where driver overhead, input data dependence, and memory bandwidth produces a slightly lower rate (see Fig. 10).

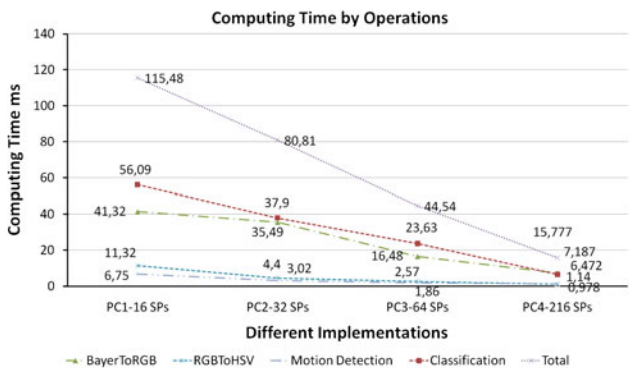


Fig. 10 Stage computing time using different *GPU* models

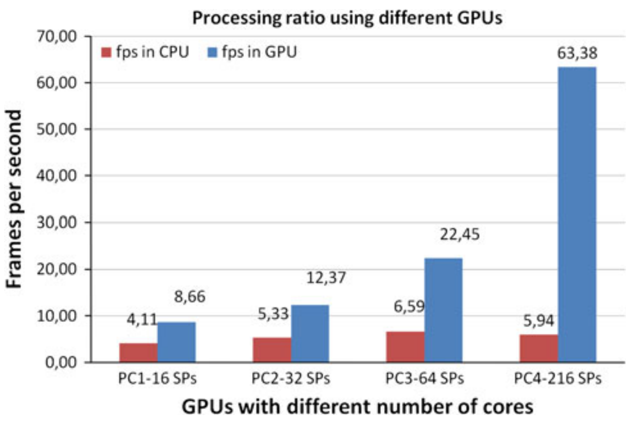


Fig. 11 Ratio in frames per second for different *GPUs* in comparison with the three available CPUs

In Fig. 11, results are compared at the application level between the GPU-CPU configurations, and the same tendency can be appreciated. A very low-cost laptop equipped with GPU1 is able to obtain enough processing ratio in fps to connect a camera to the tracking stage (8 fps or more). Nevertheless a highly optimized single threaded implementation over a medium PC as CPU4 is not able to do that. A comparison GPU-CPU in PC1 shows that achieved improvement is around 2.11× , 2.32× in PC2, and 3.41× in PC3. A considerable speedup has been obtained (10.67×) with GPU4, a Geforce GTX 260, processing 63.38 frames per second versus the 5.94 from CPU4, and 7.32× if we compare it with GPU1.

A remark about the architectures and characteristics of the different equipments under test can also be extracted. Despite the fact that the pair CPU-GPU are contemporary, the evolution of both architectures are not equal over time. CPU power increase in the last 2 years is negligible in comparison with GPUs in the same period. This can be explained due to the maturity of both technologies and the improvement margin. CPU1 is able to process 4.11 fps while CPU3 only goes up to 6,59 fps and CPU4 only achieves 5,94 fps even slower than the previous generation.

In a dual core processor (CPU1 and CPU2), while one core is doing image processing the other is used by the application for doing the other tasks (imagen capture, tracking, control and visualization). In machines with additional cores, more frames could be processed in parallel. As shown previously, CPU3 is able to run four threads, processing around 25 fps, almost three frames more than GPU3. CPU4 is also a 4-core processor but has simultaneous multithreading (Hyperthreading in Intel terminology), so it appears as eight CPUs to the Operating System. When running four threads, CPU4 achieves 21.29 fps and goes up to 30.62 fps when running eight threads, getting more throughput but slowing down each tread.

Given that we establish a minimum processing rate of at least 8 fps as requirement for a successful posterior tracking stage and that we need to process 8 cameras, it is necessary a minimum processing rate of about $8 \times 8 = 64$ *fps*. Thus, real-time can be obtained as:

- PC3 processes 6.59 fps in single thread mode or $\simeq$ 25 fps using multicore execution, so we need 3 medium-high PCs.
- A *GPU Quadro FX 1800* (*GPU3*) processes 22.45 fps, so we need at least 3 low-cost *GPUs*.
- In a hybrid implementation using a *PC3* and a *GPU3* it was possible to process $\simeq$45 fps.
- A Geforce GTX 260, while its price is around 150 dollars, shows a processing ratio of around 64 fps.

7 Conclusions and future work

7.1 Conclusions

In the light of these results, we can assert a set of interesting conclusions:

- High-capability computing devices, such as current *GPUs*, have an enormous potential for video processing applications. As proof, segmenting football players in real time have been possible by making an efficient use of these platforms.
- The usage of *GPUs* has meant a significant success for our application. We are able to improve all the processing stages, with the exception of labeling, with speed-ups up to 40$\times$ and using medium-cost hardware.
- An hybrid *segmentation* implementation, where *classifications* is done for the whole image in the GPU and *labeling* is later done by the CPU without any penalty, gives us better performance.
- The global performance improvement is 10.67$\times$ over a single thread implementation, making possible a processing rate of 63.38 fps over a single GPU.
- Over a 4-core processor we are able to process almost 25 fps in a multithreaded implementation.

7.2 Future work

Given the good performance achieved which confirms the initial promising idea, we consider this paper as a first step in a future research line. For that, we propose several ideas that, due to lack of time, resources or for being out of the scope of the paper have not been studied properly. Future lines of research can use this increase not only for increasing the processing rate but also for an intrinsic improvement of the processing stage.

- To study the evolution of processing rate according to image resolution.
- Feature modeling has been assumed as known. We propose to study the scalability according to variation in the target model (number of Gaussians, non-parametric models, etc.).
- To study how the classification metric (Euclidean distance, Mahalanobis, etc.) or even the classification methodology (neural networks, SOM, etc.) can affect to the final results.
- To extend the application field to other compatible disciplines such as facial recognition or human tracking, to name a few.

Acknowledgments This work was supported in part by grants TIN2010-21291-C02-01, TIN2007-66423 and TIN2007-60625 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ICT217068).

References

1. Bacon, D., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**, 45–420 (1993)
2. Bayer, B.E.: Bayer. United States Patent num. 3971065 (1975). http://patent.ipexl.com/US/3971065.html
3. Bilmes, J.: A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Technical report (1998)
4. Buckley, K., Vaddiraju, A., Perry, R.: A new pruning/merging algorithm for mht multitarget tracking. In: *Radar-2000* (2000)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using cuda. J. Parallel Distributed Comput. **68**(10), 1370–1380 (2008). ISSN 0743-7315. doi:10.1016/j.jpdc.2008.05.014. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.4849
6. Chen, T.Q., Lu, Y.: Color image segmentation: an innovate approach. Pattern Recogn. **35**, 395–405 (2001)
7. Cheng, H.D., Sun, Y.: A hierarchical approach to color image segmentation using homogeneity. IEEE Trans. Image Process. **9**, 2071–2082 (2000)
8. NVIDIA Corp. CUDA 2.0 Programming Guide. NVIDIA, 2008. http://www.nvidia.es
9. Martínez del Rincón, J., Herrero-Jaraba, J.E., Gómez, J.R., Orrite-Uruñuela, C., Medrano, C., Montañés, M.A.: Multi-camera sport player tracking with bayesian estimation of measurements. Comput. Vision Image Understanding (2007)
10. Martínez del Rincón, J., Orrite Uruñuela, C.: Feature-based human tracking: from coarse to fine. PhD thesis. Zaragoza, University of Zaragoza, Zaragoza, Dic 2008. Presented: December 2008
11. Fung, J., Mann, S.: Using multiple graphics cards as a general purpose parallel computer: applications to computer vision. In: ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04), vol. 1, pp. 805–808, IEEE Computer Society, Washington, DC, USA (2004). ISBN 0-7695-2128-2. doi:10.1109/ICPR.2004.968

12. Funk, N.: A study of the kalman filter applied to visual tracking. Technical report, University of Alberta (2003)
13. Gad, A., Farooq, M., Serdula, J., Peters, D.: Multitarget tracking in a multisensor multiplatform environment. In: The Seventh International Conference on Information Fusion, pp. 206–213, Stockholm, Sweden (2004)
14. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with cuda. Micro, IEEE **28**(4), 13–27 (2008). doi:10.1109/MM.2008.57
15. Gavrila, D., Philonim, V.: Real time object detection for smart vehicles. In: Proceedings of Seventh International Conference on Computer Vision, pp. 87–93 (1999)
16. Gómez, J.R., Herrero, J.E., Medrano, C., Orrite, C.: Multi-sensor system based on unscented kalman filter. In: Proceedings of Image Processing (VIIP), IASTED International Conference on Visualization, pp. 13–18 (2006)
17. Software development products Intel® *Intel® VTune Analyzer*. Intel Corporation (2009)
18. Isard, M., Blake, A.: Condensation conditional density propagation for visual tracking. Int. J. Comput. Vision, **29**(1), 5–28 (1998). ISSN 0920-5691. doi:10.1023/A:1008078328650
19. Kumar, N.S.L.P., Satoor, S., Buck, I.: Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. In: 10th IEEE International Conference on High Performance Computing and Communications, pp. 103–109 (2009). doi: 10.1109/HPCC.2009.45
20. Lu, P., Oki, H., Frey, C., Chamitoff, G., Chiao, L., Fincke, E., Foale, C., Magnus, S., McArthur, W., Tani, D., Whitson, P., Williams, J., Meyer, W., Sicker, R., Au, B., Christiansen, M., Schofield, A., Weitz, D.: Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. J. Real-Time Image Process. (2009). doi:10.1007/s11554-009-0133-1
21. McLachlan, G.J., Krishnan, T.: The EM Algorithm and Extensions. 2 edn. Wiley Series in Probability and Statistics. Wiley, New York, March 2008. ISBN 0471201707. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471201707
22. Nguyen, H.: *GPU Gems 3*. Addison-Wesley, Professional, Reading, August 2007. ISBN 0321515269
23. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: Gpu computing. In; Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899 (2008). doi:10.1109/JPROC.2008.917757
24. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput. Graphics Forum **26**(1), 80–113, March 2007. ISSN 1467-8659. doi:10.1111/j.1467-8659.2007.01012.x
25. Pérez, P., Hue, C., Vermaak, J., Gangnet, M.: Color-based probabilistic tracking. In: ECCV '02: Proceedings of the 7th European Conference on Computer Vision-Part I, pp. 661–675, Springer, London (2002). ISBN 3-540-43745-2
26. Pharr, M., Fernando, R.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, Reading, March 2005. ISBN 0321335597. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321335597
27. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73–82. ACM, New York (2008). ISBN 978-1-59593-795-7. doi:10.1145/1345206.1345220
28. Schneider, S., Yeom, J., Rose, B., Linford, J.C., Sandu, A., Nikolopoulos, D.S.: A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. *SIGPLAN Not.*, **44**(4), 131–140 (2009). ISSN 0362-1340. doi: 10.1145/1594835.1504197
29. Sinha, S.N., Frahm, J., Pollefeys, M., Genc, Y.: Gpu-based video feature tracking and matching. Technical report, In: Workshop on Edge Computing Using New Commodity Architectures (2006)
30. Smith, A.R.: Color gamut transform pairs. In: SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques, pp. 12–19. ACM, New York (1978). doi:10.1145/800248.807361
31. Tuytelaars, T., Mikolajczyk. K.: Local invariant feature detectors: a survey. Found. Trends. Comput. Graph. Vis. **3**(3):177–280 (2008). ISSN 1572-2740. doi:10.1561/0600000017
32. van der Laan, W.J.: Decuda and cudasm, the cubin utilities package. GIThub (2009)

Author Biographies

Miguel Angel Montañés Laborda received his M.Sc. degree in 2010 from the University of Zaragoza specializing in systems engineering and computing. He previously graduated in electronic engineering in 2001 and he is currently completing the second cycle of electronic and automatic engineering, both from the University of Zaragoza. In September 2004, he joined the Computer Vision Laboratory of the Aragon Institute of Engineering Research (I3A) as a scientific developer.

Enrique F. Torres Moreno received the MS degree in computer science from the Polytechnic University of Catalunya in 1993, and the Ph.D. degree in computing science from the University of Zaragoza in 2005. He was an assistant professor in the Polytechnic Schools of the University of Girona. He is an assistant professor in the Computer Science and Systems Engineering Department (DIIS) at the University of Zaragoza, Spain. He is also on sabbatical leave for study and research at the University of California in Berkeley, where he is a member of the International Computer Science Institute (ICSI). His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He is a member of the IEEE Computer Society. He is also a member of the Aragón Institute of Engineering Research (I3A) and the European HiPEAC NoE. More details about his research and background can be found at http://webdiis.unizar.es/gaz/miembros.html.

Jesús Martínez del Rincón received the Ph.D. degree from the University of Zaragoza specializing in Biomedical Engineering in 2008. He previously graduated from the University of Zaragoza in Telecommunication in 2003. He is currently a research fellow in the Faculty of Computing, Information Systems and Mathematics, Kingston University, London. His current research interests include aspects of computer vision such as human motion analysis, activity recognition and multi-target tracking in real time.

José Elías Herrero Jaraba received his Ph.D. degree in 2005 from the University of Zaragoza, Spain. He joined the Centro Politécnico Superior of the University of Zaragoza as a researcher in March 2001. In February 2003 he became an assistant professor, and since May 2007 he has been an associate professor at the same university. His current research interests include image processing, multicamera and multitarget tracking, three-dimensional vision, and measurement processes. Dr. Herrero is an associate member of the IEEE.