Scalability of Color-Based Segmentation of Football Players over GPUs

Miguel Ángel Montañés
University of Zaragoza
1 Maria de Luna
Zaragoza, Spain
mmonla@unizar.es

Enrique F. Torres*
Univ. of Zaragoza, HiPEAC
1 Maria de Luna
Zaragoza, Spain
enrique.torres@unizar.es

Jesús Martínez
University of Kingston
Penrhyn Road, Kingston Upon Thames
KT1 2EE, UK
Jesus.Martinezdelrincon@kingston.ac.uk

J. Elías Herrero
University of Zaragoza
1 Maria de Luna
Zaragoza, Spain
jelias@unizar.es

ABSTRACT

In this paper, we study the scalability of a real application to the available number of cores in the *GPU*. Our application is a real-time image processing in which a football player feature extractor based in color patterns obtain feasible measures for tracking system. Since football players are composed for diverse and complex color patterns, a Gaussian Mixture Models *(GMM)* is applied as segmentation paradigm. 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. The optimized application has been benchmarked over different *GPUs* with different number of cores. Due to data dependencies performance increase 1.4x when doubling number of cores.

1. INTRODUCTION

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 prerecorded 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. 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 of a mixture of different tones associated to a 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. Unfortunately, most programs are conceived using a serial philosophy. Serial code cannot automatically take advantage of multiple cores to execute itself faster, so that code must be redesigned from a newer parallel point of view.

The *GPU* architecture is optimized for massively parallel processing with peaks up to hundreds of GFLOPS. 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* [4]. This language is a set of parallel extensions of the C/C++ programming languages and it is able to interact with a special

*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/ICT 217068).

hardware interface built into all current NVIDIA *GPUs*.

In the last few years, the amount of scientific application tested over *GP-GPU* has increased [3]. Although generally those researches are focused on specific calculations, they provide an initial idea about the intrinsic potential of this new platform [10]. Particularly, in our field of interest, several studies probe this capability in modern *GPUs* [7]. Traditional methodologies have been implemented, such as pattern recognition algorithms based on textures [6], Gaussian mixture models [9] or image feature extraction techniques [13, 15]. 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. Our election has been Expectation Maximization for Gaussian Mixture Models [9]. Since one of our main objectives is to process multi high-resolution cameras, detection and classification processes must be applied on real time in a 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 (Section 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 Section 2 the hardware infrastructure is described. Section 3 introduces the stages that compose our application and discusses their computational cost. Section 4 explains the computational cost of a prototype implemented in an optimized version in C++ running in a conventional CPU. In section 5, the parallelization as well as the CUDA implementation are detailed. Section 6 presents a comparison between CPU and *GPU* results and its scalability. Finally, conclusions are presented in Section 7.

2. INFRASTRUCTURE

Our approach consists of the processing and classification algorithms for football players in sequences provided from one or multiple cameras, which are installed in a real football stadium. In our infrastructure, we propose a system composed of 8 static high definition digital cameras (resolution 1388x1036) with overlapping fields of view. The cameras are positioned around the stadium as is shown in fig. 1.

This camera distribution has been done in this way because the minimum number of cameras for covering the football field with enough resolution is 8 and the overlapping cameras are crucial to solve occlusions, specially in conflictive areas.

In order to check the performance improvement and scalability that our implementation achieves, we have tested the algorithm over different types of processors and *GPUs*. Thus, three different types of PCs are available for scalability study and a fourth PC is used to confirm results. These

	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	2.8	Geforce GTX260	216	111.9

Table 1: Different types of CPUs and GPUs for testing

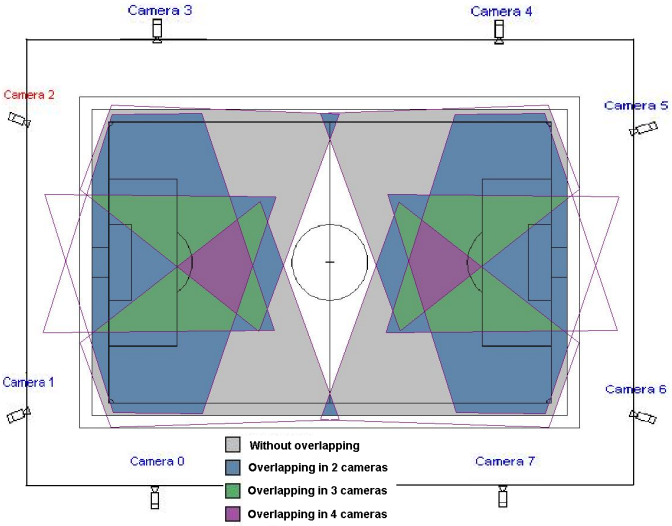


Figure 1: Camera distribution on the roof

equipments are shown in table 1. These equipments are close to the average current processors, giving us a significative sampling of the market. On the other hand, 4 different *GPUs* have been tested too keeping same philosophy. They also fulfill another requirement: since our implementation employs atomic functions to obtain synchronism, we need video cards compatible with *CUDA Compute Capability 1.1* or higher. For every possible combination of both platforms (CPUs and *GPUs*), a scalability study was made.

3. DESCRIPTION OF APPLICATION

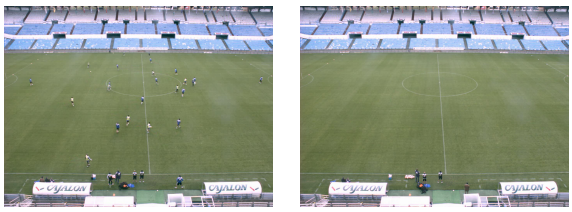
The proposed classification algorithm can be decomposed into a set of steps. Most of them should be done per camera. The steps and input data that they require are described next.

3.1 Independent processing per camera

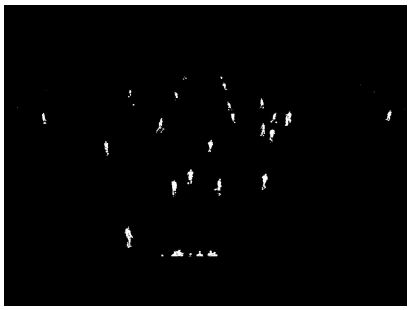
- Image Capture: at this stage, images are retrieved on demand from each camera.
- Color Space Transformation from Bayer to RGB: high-resolution cameras usually provide images in raw format (also called Bayer-type RGGB [2]), i.e. 8 bits per pixels for color codification. To obtain a RGB image, we need an intermediate transformation process

called BayerToRGB. 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.

- Motion Detection: it consists in a thresholded subtraction between the current image (fig. 2 a)) of every camera and a pre-generated image of the scenario, called *background* (fig. 2 b)). Process is shown in fig. 2 c). Motion detection image contains the dynamic areas, which will be used for posterior processing like distracter removal.
- Color Space Conversion *RGB* to *HSV*: under variable illumination conditions, better segmentation results can be obtained by applying a transformation in the color space [14]. Instead of RGB, HSV (Huge, Saturation, Value) has shown a better accuracy.
- Blob Labelling: 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.*



a) Current image b) *Background* image



c) Motion Image, thresholded subtraction between the current image and *background.*

Figure 2: Current image, *background* image and substraction result image.

Fig. 3 details the processing flow per camera. Output generated from previous stages is used as input for a tracking algorithm in order to ensure the temporal coherence. Although it is out of the scope of this paper, a *Multi-Camera Uncensted Kalman Filter (MCUKF)* [8] has been used to

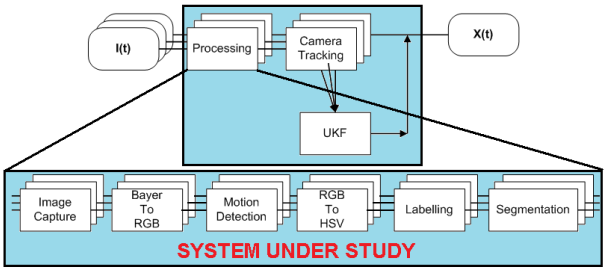


Figure 3: Processing schema

demonstrate the global feasibility. Empirical experiments allow us to conclude that a successful tracking can be obtained with a processing frame rate between 8 and 15 per each camera and to process 8 cameras we need more than 64 images per second for real time.

3.2 Gaussian Mixture Method for Image Segmentation

In collaborative sport applications, 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 segmentation even in presence of complex targets, which can be composed of multimodal appearance distributions. Since it is a parametric technique, it needs a 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 [12].

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

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

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

1. Off-Line Computing consist of a sample selection in order to supervise sample selection for every group (*team 1, team 2 and background*), parameter tuning to adequate number of Gaussians used to model every group and training.

2. On-Line Computing is composed of a classification step in which every pixel is classified into one of the different groups. For this, it is required to HSV conversion, computation of the distance between the pixel candidate and the different model of every group, final decision based on minimum distance and probability computation to measure the membership degree to every group. This last process generates, as result, probability images [5] that can be used to improve the tracking quality based on stochastic approaches.

As the offline stage is only applied once at the beginning and under supervision, it can be considered out of the real-time system and, therefore, its implementation is not required over a *GPU* platform for our goals. However, this process is also amenable to be implemented using the *GPU*, as it was demonstrated in [9], obtaining excellent results.

4. CPU IMPLEMENTATION

A single thread implementation of the algorithm was made in C++ language running under Windows.

For this optimization process, performance analysis tools, such as *Intel VTune Performance Analyzer* were applied to identify the possible *hotspots*. This tool aimed at increasing performance, in addition to the location of hotspots, allowed us to perform a deep analysis of them. Thus, *Intel VTune Performance Analyzer* let us to detect, to re-code and to optimize our implementation, improving the performance substantially. An important difference must be noticed between those general optimizations and those that act on specific parts of the code.

- General optimizations: they improve the global performance of the application. Our main optimizations consist in:

  1. Usage of a specific optimizing compiler, such as *Intel C++ compiler*. Full optimization and specific architecture compilation flags are both used in this implementation.
  2. Classical Code Optimizations [1]. It is crucial to take into account the memory mapping of data structures. In this way, we ensure a high success rate in the access to cache memories. For that, input data must be stored consecutively in memory as often as possible, i.e. data are stored in memory in raw order. Therefore, in loops, image data have to be accessed by rows. Thus, data access obtains high cache hit. Access by columns fails in cache because data are not consecutively in memory.

- Specific optimizations: they improve the performance of given functions. The most important of them 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 all possible result for any input combination. This resulting matrix is called *Look-up Table* (*LUT*). An example is color space conversion. For each RGB value, the classification result is calculated and stored in LUT. After its generation, the expensive calculation is replaced for a memory access to the right memory slot, which implies a substantial boost of the efficiency. The more complex the operation is, the more efficient this technique is. For our particular case, calculations for determining the segmentation of a pixel costs 39.96ns, whereas the memory access to check the value in the LUT is 6.02 ns, which implies a speed-up x6.63. Although generating the LUT implies a fixed cost of 670.48 ms, it can be done off-line since the color model is usually static.

Stages	Time (ms)	% of total
Conversion Bayer to RGB	15.7	10.34
Motion Detection	9.83	6.47
Conversion RGB to HSV	35.61	23.45
Labelling	5.3	3.49
Segmentation	85.39	56.24

Table 2: Time of different stages over optimized Intel C++ compilation [ICC] and percentage time of them over total time.

In table 2 is shown that any implementation using Intel C++ [ICC] achieves good results.

$$Rate_{Intel}(fps) = 6.58 fps$$

In spite of this considerable improvement, *Conversion RGB to HSV* and *Segmentation* stages still remain as critical bottle necks. Therefore, in order to raise the performance and to be able to achieve our goals, a more powerful tool is required. Moreover, other stages that were not so critical *a priori*, like *Conversion RGB to HSV*, have acquired now a more important role. It is because of this reason that stages shown in Table 2 have been implemented on the *GPU* platform, with a special focus on the *Segmentation* stage.

5. GPU IMPLEMENTATION

The hardware architecture of a system with a *GPU* can be seen in fig. 4. 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 allowing to take advantage over the current CPUs.

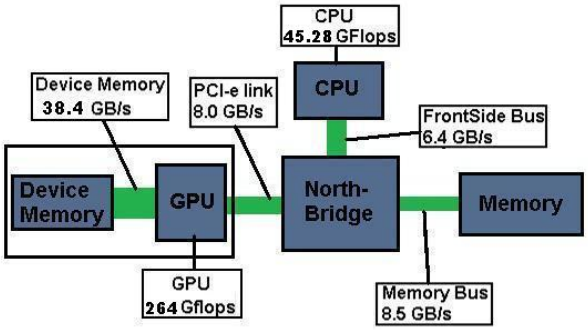


Figure 4: Hardware architecture of a system with *GPU*

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

1. Hardware composed of several computing functional units and several multicores.

2. In single precision floating point, a *GPU* can reach up to 500 Gflops owed to the 30-50 Gflops of conventional CPUs.

3. It has a high bandwidth to the internal memory of up to one order of magnitude higher than the bandwidth of a CPU and system memory (about 86.4 GB/s in a *GPU* versus 8.5 GB/s in a CPU).

4. In order to take advantage of such high bandwidth, *GPUs* allow several memory access operations to run simultaneously.

5. Paradigm *Single Instruction Multiple Thread*, SIMT, is used by the *GPU*. This specific execution allows and needs many independent and simultaneous active threads that execute the same instructions over different data. All of them are running into a unique kernel at the same time.

Attending *GPU* characteristics and SIMT paradigm, a preliminary study of our application is needed. Different criteria have been used in this analysis: Computational cost and redesign of several algorithms for massively parallel computing.

5.1 Preliminary Study

In this section, the adequacy of each stage to be implemented on *GPU* has been analyzed. The *CUDA* implementation was tested using PC3 (see Section 2 table 1) obtaining the results shown in fig. 5, where we can conclude:

- **Conversion Bayer to *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. Therefore, there is no easily adaptable and massively parallelizable implementation due to a dependency among the instructions data. In spite of pixelwise calculation, *Conversion Bayer a RGB* stage presents several dependencies in its data. *CUDA* implementation has to be carefully studied because time is higher in *CUDA* implementation as is depicted in fig. 5.
- **Motion detection:** Since it is basically a pixelwise subtraction, there is not dependency with the neighbor pixels and a new thread per pixel can be launched independently. *Motion Detection* and *Conversion RGB to HSV* stages prove a good behavior when they are implemented over *CUDA*. This results are obtained because in this phases the computation is realized pixel by pixel and dependency data is very scarce. Time cost is reduced considerably.
- **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.
- **Blob Labelling:** 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. *Labelling* stage is not parallelizable and our designed algorithm for *GPU* has a deficient behavior. Its computation time has increased.
- **Color Segmentation:** it is also a good candidate to be implemented on *GPU* as computation does not have dependency with the neighbors and it implies a substantial part of the total time. It can be decomposed into three substages: resulting image calculation by consulting the corresponging *LUT*, *LUT* update for the next frame and noise filtering by morphological operators. *CUDA* implementation of *Segmentation* stage presents a significative improvement.

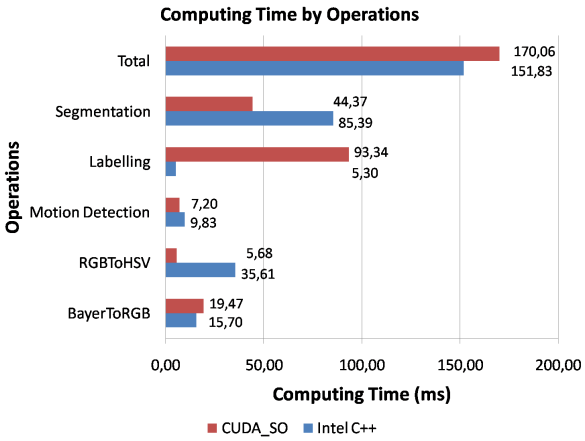


Figure 5: Computational cost for [ICC] and *CUDA* (over PC3) implementations.

Because of data shown in fig. 5, relevant decisions can be taken. Analyzing fig. 5, it can be observed that *segmentation*, for being the most expensive stage, must be analyzed carefully. This stage takes between 58.27% to 57.8% of computing time (without taking into account labelling time) in [ICC] or *CUDA* implementations respectively. Since *Conversion Bayer to RGB* stage takes between 25.37% (in PC3) of the total time in *CUDA* implementation and its data dependencies detected, it needs a special optimization. For *Motion Detection* and *Conversion RGB to HSV* stages, data independence provides margin to get better.

A critical design phase is the labelling computing, since it is not parallelizable. Since labelling becomes a expensive stage in *GPU* as observed in fig. 5, it is worthy to take special care in aspects as kernel context switch or data transfer with CPU, avoiding unnecessary waste of time. Three solutions must be studied:

- **Option 1:** All the stages are run on the *GPU*: Labelling allows identifying active areas in the image, reducing the segmentation to those areas and making unnecessary segmenting the rest of the image. Total computational cost would be $T_{total_1} = T_p + t_{e_{gpu}} + t_{s_{blob}}$, where T_p is the time due to the pre-labelling stages, $t_{e_{gpu}}$ is the labelling cost in *GPU* and $t_{s_{blob}}$ is the segmentation cost on the active areas.
- **Option 2:** Previous stages to labelling are run on *GPU*, results are transferred to the host, which runs the labelling and returns the result to the *GPU*, where the segmentation is done on the active areas. $T_{total_2} = T_p + t_{totaltrans} + t_{e_{cpu}} + t_{totaltrans} + t_{s_{blob}}$, being $t_{totaltrans}$ the transference cost + kernel commutation cost + driver access cost.
- **Option 3:** All the stages are run on *GPU* an the labelling is eliminated. This implies that segmentation is applied to the whole image and not only over active areas. $T_{total_3} = T_p + t_{s_{image}}$.

In each kernel switch or data transference, the *CPU* needs to access the *GPU* driver to complete the operation, which implies an additional time.

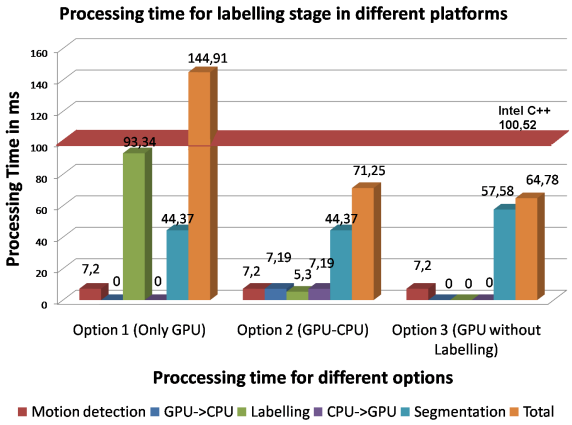


Figure 6: Comparison: First *CUDA* (over PC3) implementation versus optimized C++

1. The computational cost of transferring data $CPU \Rightarrow GPU$ *or* $GPU \Rightarrow CPU$ is around 7.19 ms. By running as many instructions as possible inside the *GPU*, just two transfers should be needed: to introduce input data and to obtain the results.
2. For each kernel switch, the *GPU* requires extra time for changing the context. By grouping different stages in a shared *kernel*, we save this extra time.

Therefore, every single operation of any type that we could run into the *GPU* will avoid to waste time unnecessarily. Thus, it is a good practise to design stages which could run into the same kernel.

Previous options have been tested and results are shown in fig. 6 from PC3. By minimizing the computational cost (T_{total_1}, T_{total_2} and T_{total_3}), the optimum decision can be taken. As fig. 6 shown, option 3 provides the optimum solution (64.78 ms) in comparison with the other alternatives whose costs are 144.91 and 71.25 ms. As fig. 6 shows, 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 segmented in the last one. In the light of previous results, we can conclude that *Blob Labelling* is not efficient for parallel computing and, in case it would be necessary for the 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.Therefore option 3 has been selected, *Labelling* stage is relegated to CPU (if it is needed) and *segmentation* is applied over the whole image.

5.2 Techniques for optimizing GPU code

Several techniques are at our disposal for an optimum use of *GPU* capacities according to recommended methodologies [11]. 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 large memory bandwidth. Therefore, the main techniques for increasing performance are based on keeping up the occupation of functional units (known as occupancy) and maximizing the use of effective bandwidth to memory. Next, the most effective ones are described.

5.2.1 *Occupancy*

Occupancy is measured by the number of threads assigned to each processor. Maintaining a high occupancy in the *GPU* is important to performance due to it can be achieved by means of two different ways: through the number of registers and through the amount of *shared* memory employed.

As a general rule, the less the number of register used per kernel, the higher occupancy. However, it is worthy to note that this modification is not always easy since it strongly depends on the algorithm and could imply a deep restructuring.

By analyzing one of the segmentation substages and restructuring an indexing instruction for memory allocation, we were able to save 2 registers per kernel. This complex reduction implies the core occupancy has gone from 66% to 100%

5.2.2 *Coalescence*

Coalescence is a technique for optimizing memory accesses. Memory accesses from different threads can be merged into a single access if the required conditions are fulfilled [4]. This fusion process is known as coalescence. Coalescence is defined as a mean to gather several simultaneous memory accesses in parallel. It is promoting during the global memory accesses.

Coalescence is, without doubt, the most powerful method for optimization in *GPU*. It consists in a mechanism that fuses into a unique operation all read/write accesses from the running threads in the current active block. *GPUs* have specific hardware that detects and makes this fusion, allowing to hide the high latency of threads accessing to local or global memory when cache is not available, and improving the speed-up above two orders of magnitude for these operations.

This technique is specially relevant in the following stages, although it has been applied across the whole system: conversion Bayer to RGB: 100% coalescent on writing and on some reading, conversion RGB to HSV: 100% coalescent on both writing and reading, motion detection: 100% coalescent on both writing and reading and color segmentation: $\approx$ 10% coalescent in substage 1, $\approx$ 30% coalescent in substage 2 and 100% coalescent in substage 3.

5.2.3 *Others Techniques*

Other techniques to achieve improvement in *GPU* are:

- Masking of high latency memory accesses: this can be achieved by sending non data-dependant instructions to the processing units during the transference cycles.
- Avoiding branch divergence: when several threads should take different paths, it is called divergence and the execution times of all the branches become serialized, increasing the cost for every divergent thread.

	CPU	GPU	Speedup
PC1	4.11	8.66	2.11
PC2	5.33	12.37	2.32
PC3	6.58	22.45	3.41
PC4	5.94	63.38	10.67

Table 3: Final results (in number of frames)

6. RESULTS AND SCALABILITY TEST

As our system is composed of identical high definition cameras (1388x1036), we will only analyze the processing time for one of them. Later, we could extrapolate results to work out the scalability of our processing kernel.

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 that have been selected on the basis of different criteria:

1. The CPUs will be of mid-high range because it seeks a significant increase in computational power.
2. The first 3 GPUs have been chosen with the criteria of having a number of cores that is a multiple of the number of cores of the previous GPU. The aim is to study the evolution of the cost of processing each of the phases and the global system.
3. The fourth *GPU* is chosen to confirm the tendency showed in the previous tests as this section describes.

Thus, the chosen configuration for each experiment is as is shown in Section 2 table 1.

Analyzing results from the application point of view (see Table 3), a considerable speed increase has been obtained (10.67x), being possible to process 63.38 frames per second with a Geforce GTX 260 versus the 5.94 that CPU4 could. A comparison GPU - CPU in PC1 shows that achieved improvement is around 2.11x, since this CPU processed 4.11 fps and its GPU processed 8.66 fps. Values in comparison GPU - CPU in PC2 achieve 5.33 fps in CPU2 and 12.37 fps in GPU2 resulting in a speed up of 2.32. Values in comparison GPU - CPU in PC3 achieve 6.59 fps in CPU3 and 22.45 fps in GPU3 resulting in a speed up of 3.41.

In the same manner, a comparison among the time cost evolution of different stages and process ratio in fps over different equipments has been extracted (see fig. 7 and 8).

In these figures, results using the 4 *GPUs* with 16, 32, 64 and 216 cores are depicted. Two comparative analysis can be done: evaluating the time cost for every stage for each GPU or comparing the global performance of the application using the 4 different CPUs against the GPUs measured in frames per second, fps.

Analyzing in the stage level (fig. 7), it is important to note that improvement increase with *GPU* performance, almost always proportional to the number of cores. The only exceptions are the conversion Bayer to RGB and segmentation stages, where input data dependence produces a slightly lower rate (see fig. 7). Global improvement has an almost

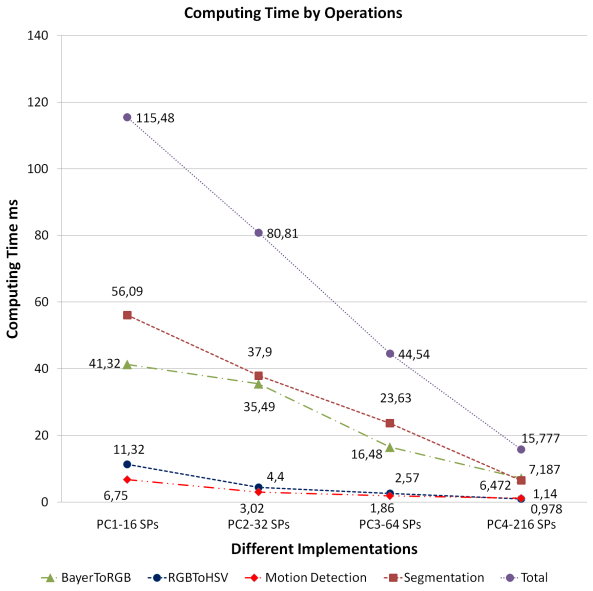


Figure 7: Stage computing time using different *GPU* models.

linear tendency achieving an execution code 7.32 time faster in GPU4 than in GPU1.

In fig. 8, results are compared in the application level between three 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 tracking stage (8 fps or more). Nevertheless a highly optimized implementation in a medium PC as PC4 is not able to do that without the *GPU*.

It is also worthy to note some characteristics of the three different equipments under test. 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 two years is really smaller in comparison with GPUs in the same period. This can be explained due to the maturity of both technologies and the improvement margin.

It has to be noticed how a low-cost *GPU* as Geforce 8600M GS with only 16 cores takes advantage over a medium-high CPU as Core i7 (8.66 fps versus 5.94 fps, respectively), being 1.46 times faster in global processing.

Finally, note that the speed-up increase with the number of cores is constant although not in the same proportion. This difference is mainly due to the overhead of the transference time CPU $\Leftrightarrow$ *GPU*.

Given that a minimum processing rate of 8 fps is required for a posterior tracking stage and that we need to process 8 cameras, it is necessary a minimum processing rate of about $8 * 8 = 64 fps$. Thus, the scalability can be obtained as:

- PC type 3 processes 6.58 fps per camera using CPU3, so we need 10 medium-high PC.
- A *GPU Quadro FX 1800* (*GPU3*) processes 22.45 fps per camera, so we need 3 low-cost *GPUs*.

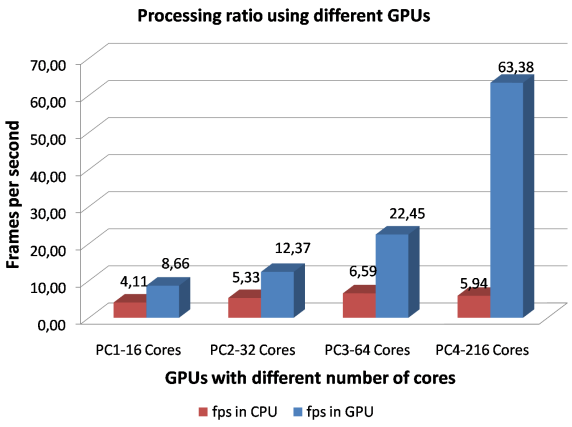


Figure 8: Ratio in frames per second for different *GPUs* in comparison with the three available CPUs.

In addition, since the performance increase is ∼1.4x when the number of cores doubles, we could extrapolate that a machine with a *GPU* with a triple number of cores, could process the 64 fps needed over only one equipment.

This extrapolation has been confirmed in a experiment over a Geforce GTX 260 while price is around 150 dollars. Results show a processing ratio around 64 fps proving that our scalability study is correct.

7. CONCLUSIONS

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

- Usage of high-capability computing devices, such as *GPUs*, have a potential for this kind of applications. It has been possible to segment football players in real time by making an efficient use of these platforms. We are able to improve all the processing stages, with the exception of labelling, with speed-ups up to 40x and using medium-cost hardware. The global performance improvement is x10.67 making possible a processing rate of 63.38 fps instead of the 4.11 fps in low-medium PC (PC1), 5.33 fps in medium PC (PC2), 6.58 fps in medium-high PC (PC3) or 5.94 fps in medium-high PC (PC4).
- We have been able to make the functionality independent of the scalability. Therefore, we have proved that a single but more powerful card would be able to process our 8 cameras.
- Even optimizing the *GPU* occupancy and the effective memory bandwidth using coalescence, scalability is affected by the data dependencies.

8. REFERENCES

[1] D. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, 1993.

[2] B. E. Bayer. Bayer. United States Patent, 1975. http://en.wikipedia.org/wiki/Bayer_filter.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[4] N. Corp. *CUDA 2.0 Programming Guide*. NVIDIA, 2008.

[5] J. M. del Rincón and C. O. Uruñuela. *Feature-based human tracking: from coarse to fine*. PhD thesis, Zaragoza, University of Zaragoza, Zaragoza, Dic 2008. Presented: December 2008.

[6] J. Fung and S. Mann. 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) Volume 1*, pages 805–808, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, September 2008.

[8] J. R. Gómez, J. E. Herrero, C. Medrano, and C. Orrite. Multi-sensor system based on unscented kalman filter. In *IASTED*, pages 13–18. In Proc. Image Processing (VIIP), 2006.

[9] N. S. L. P. Kumar, S. Satoor, and I. Buck. Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:103–109, 2009.

[10] P. Lu, H. Oki, C. Frey, G. Chamitoff, L. Chiao, E. Fincke, C. Foale, S. Magnus, W. McArthur, D. Tani, P. Whitson, J. Williams, W. Meyer, R. Sicker, B. Au, M. Christiansen, A. Schofield, and D.Weitz. Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 2009.

[11] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[12] P. Pérez, C. Hue, J. Vermaak, and M. Gangnet. Color-based probabilistic tracking. In *ECCV '02*, pages 661–675, London, UK, 2002. Springer-Verlag.

[13] S. N. Sinha, J. Frahm, M. Pollefeys, and Y. Genc. Gpu-based video feature tracking and matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[14] A. R. Smith. Color gamut transform pairs. In *SIGGRAPH '78: Proc. of the 5th annual conference on Computer graphics and interactive techniques*, pages 12–19, New York, NY, USA, 1978. ACM.

[15] T. Tuytelaars and K. Mikolajczyk. Local invariant feature detectors: a survey. *Found. Trends. Comput. Graph. Vis.*, 3(3):177–280, 2008.