Parallelization of a Video Segmentation Algorithm on CUDA–Enabled Graphics Processing Units

Juan Gómez-Luna[1], José María González-Linares[2], José Ignacio Benavides[1], and Nicolás Guil[2]

[1] Computer Architecture and Electronics Department, University of Córdoba, Campus de Rabanales, 14071 Córdoba, Spain
{el1goluj,el1bebej}@uco.es
[2] Computer Architecture Department, University of Málaga, Campus de Teatinos, 28071 Málaga, Spain
{gonzalez,nico}@ac.uma.es

Abstract. Nowadays, Graphics Processing Units (GPU) are emerging as SIMD coprocessors for general purpose computations, specially after the launch of nVIDIA CUDA. Since then, some libraries have been implemented for matrix computation and image processing. However, in real video applications some stages need irregular data distributions and the parallelism is not so inherent. This paper presents the parallelization of a video segmentation application on GPU hardware, which implements an algorithm for abrupt and gradual transitions detection. A critical part of the algorithm requires highly intensive computation for video frames features calculation. Results on three CUDA-enabled GPUs are encouraging, because of the significant speedup achieved. They are also compared with an OpenMP version of the algorithm, running on two platforms with multiples cores.

Keywords. Video Segmentation, CUDA, Canny, Generalized Hough Transform.

1 Introduction

A high number of video analysis applications, such as storyboard generation, video comparison, scene detection, video summarization, etc., apply temporal video segmentation as a previous step for further processing. Temporal video segmentation is a video processing technique, which is able to identify the shots appearing in a video, i.e., the sequences of frames captured from a single camera operation. Thus, a thorough study of video segmentation seems very convenient, in order to exploit their parallelism.

Two different types of transitions link the shots in a video. On the one hand, abrupt transitions or cuts occur when the change from one shot to the next one is performed in just one frame. On the other hand, gradual transitions involve several frames. Shot detection is performed through shot transition detection algorithms. This work focuses concretely on the on-line abrupt and gradual transitions detection algorithm presented in [1] and [2], which has achieved a high score in transitions detection.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 924–935, 2009.

The aforementioned applications are used in the digital video industry where, typically, cheap hardware computation platforms are available. Nowadays, these platforms may have a multi-core microprocessor, which is the trend in recent years. Applications should be parallelized in order to exploit the Thread Level Parallelism (TLP) inherent in these two- or four-core microprocessors. However, using the Graphics Processing Unit (GPU) as a SIMD coprocessor can also be considered.

Since the launch of nVIDIA CUDA [3], programming a GPU for general purpose computations is pretty much easier than before. The parallelizing effort with CUDA is equivalent to that with OpenMP [4] or any other application programming interface (API). Moreover, some libraries, such as OpenVidia [5] and GPUCV [6], have been implemented for matrix and image computations. However, the algorithm, scope of this work, is a typical video application, where some stages need irregular data distributions and the parallelism is not so inherent.

This paper presents the parallelization of the abrupt and gradual transitions detection algorithm, described in Section 2, on three CUDA–enabled GPUs: nVIDIA GeForce 8800 GTS, nVIDIA GeForce 9800 GX2, both with CUDA compute capability 1.1, and the recently released nVIDIA GeForce GTX 280, with CUDA compute capability 1.3. The performance of these implementations is compared with a data decomposition using OpenMP on a four-core Intel Core2Quad Q6600 2.40GHz and an eight-core Intel Xeon X5355 2.66 GHz multi-processor, in Section 5. Section 3 introduces CUDA and describes the decomposition of the algorithm in kernels. Section 4 explains the implementation of the most significant kernels.

2 Temporal Video Segmentation

This section presents a unified scheme for on-line video segmentation based on luminance and contour information from video sequences [1], [2]. It is able to detect abrupt as well as gradual shot transitions with high accuracy using neural networks for classification purposes. The process is summarized in Fig. 1.

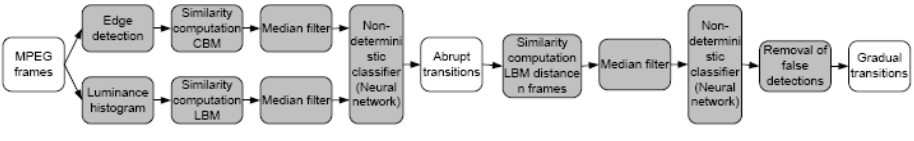

Fig. 1. Flow diagram to perform abrupt and gradual shot transitions detection. Shadowed stages indicate computing phases within the algorithm.

Two metrics are used to compare video frames, as it is explained below. The MPEG stream is decompressed and divided into frames, from which contour and luminance information is obtained:

– Contour information is extracted using the Canny edge detection algorithm [7]. Next, a Generalized Hough Transform (GHT, [8]) searches for couples of edge points whose gradient directions (θ) differ a certain angle (typically, two angles of 90º and 135º are used). When a pairing is found, a value α is computed. Then, the

(α, θ) element of a 2D histogram, called orientation table (OT), is updated (OT(α,θ) = OT(α,θ) + 1). The OT of an image can be used to detect if a given template, represented by its own OT, is present in the image. This algorithm uses a frame of the video as a template for the next frame. Thus, OTs of two frames are compared using a modified normalized 2D cross correlation [9] to obtain a similarity value (CBM, Contour Based Metric).

- A luminance histogram is computed for every frame. The histograms of two frames can be compared using a modified normalized 1D cross correlation function that obtains a similarity value (LBM, Luminance Based Metric).

Abrupt shot transitions are computed by comparing consecutive frames. The resulting similarity values of CBM and LBM are filtered by a third order median filter to minimize the effect of flashes and motion in the sequence. After that, a non-deterministic classifier identifies the abrupt transitions. The classifier is based on a multi-layer perceptron neural network with one hidden layer, 6 elements in the input layer, 10 neurons in the hidden layer, and 2 output elements.

Once abrupt transitions detection has been carried out, gradual shot transitions detection is performed by comparing frames at distance n (typically n=15). Candidate gradual transitions are identified by means of LBM over several frames and a new multi-layer perceptron neural network. This has 101 elements in the input layer, 80 neurons in the hidden layer, and 2 output elements.

Once candidate gradual transitions have been detected, it is required to discard false detections caused by camera operation. In order to do this, the algorithm implements a motion estimation procedure based on the GHT, which detects camera effects, such as zooms or displacements. Since displacements are the most usual effects, a Displacement Table DT is calculated for each frame. The displacement between a frame and the next ones in a window w (typically w=10) can be determined comparing the DT of each.

3 Implementing with CUDA

Since this algorithm is highly compute-intensive, its execution on a single CPU can take more than three seconds per frame. This value is too far from real-time, which would be a very desirable target for the applications it can be used. The scope of the algorithm is on-line video segmentation. Hence, parallelization should be applied to the computation of each frame. Some parts of this computation are really complex, such as the contour information extraction and the generation of the orientation table (OT). Moreover, some of them exhibit an inherent parallelism, because computation over all pixels of the image is regular. Thus, these parts present typical features of SIMD processing, what makes them suitable for parallelization on GPU. However, the parallelism is not so clear in other parts as explained next. Obtaining a performance improvement of these parts is the major challenge of this work.

3.1 CUDA Programming Interface

CUDA consists of a set of C language library functions and a compiler (nvcc), which generates the executable code for the GPU. CUDA offers a huge number of threads

running in parallel. Some elements should be described, because of their importance while programming:

- A **block** is a group of threads, which is mapped to a single multi-processor. All threads of a block can access 16 Kbytes of shared memory. The resources of the multi-processor are equally divided among the threads. The data is also divided among the threads in a SIMD fashion.
- The threads of a block run logically in parallel, but not always physically due to the limited hardware resources. A collection of threads running concurrently at the same time is called a **warp**. The size of a warp is 32.
- Every thread executes the same code called **kernel**. The number of blocks and the number of threads in a block should be defined for each kernel.

From the hardware point of view, threads execute on SIMD single multi-processors. The number of multi-processors depends on GPU model, varying from 1 in the first CUDA-capable models to 30 in the GeForce GTX 280.

All the threads of a kernel access the global memory or device memory. Data is transferred to global memory from the memory of the host, that is, the CPU. Moreover, each multi-processor contains a number of registers, which are split among the threads of a block.

The maximum efficiency of a kernel can be obtained when the occupancy of the multi-processors is as high as possible and the memory hierarchy is properly used.

On the one hand, in order to determine the occupancy, the code is compiled using the nvcc and a special flag that outputs the amount of memory and the registers consumed by the kernel. Analyzing these values permits to determine the number of threads of a block for maximum efficiency. Usually, a block should contain 128-256 threads to minimize execution time. The number of blocks is set according to entry data size. On the other hand, global memory is a high-latency memory. Its bandwidth is used most efficiently when simultaneous memory accesses by threads can be *coalesced* into a single memory transaction. Coalescing occurs when the threads access words in sequence, i.e., the k^{th} thread access the k^{th} word, and the words lie in the same segment of 32, 64 or 128 bytes. Furthermore, a proper use of the low-latency shared memory can also improve performance. Threads of a block access shared memory simultaneously when the data resides in different memory banks. If it does not, simultaneous accesses are serialized.

3.2 Decomposition of the Algorithm in Kernels

This implementation with CUDA is focused on obtaining the similarity values of CBM and LBM, which are the most time-consuming stages of the algorithm. The execution time of the classifiers and the motion estimation is negligible. Thus, they are delegated to the CPU.

Both metrics are implemented with several kernels. Defining what tasks are coded into a kernel is a programmer's decision. In this work, most of the kernels are oriented to computation, but others prepare data for further processing by other kernel. Moreover,

some kernels are based on sample codes, included in the CUDA SDK, which implement commonly used functions, such as separable convolution [10] or histogram [11].

CBM and LBM require, first, a features extraction and, second, the comparison of these features. CBM is divided into edge detection, orientation table generation and correlation between consecutive orientation tables. Features extraction in LBM consists of generating a luminance histogram for each frame. After that, histograms are correlated using a 1D cross correlation. Table 1 presents the kernels of this implementation.

Table 1. Collection of kernels used in the implementation of CBM and LBM. Some of them are custom developed, while others are based on CUDA separable convolution and histogram.

Stage	Kernel	Description	Development
Canny Edge Detection	Convolution Row	Row filtering	Based on CUDA sample
	Convolution Column	Column filtering	Based on CUDA sample
	Gradient Calculation	Gradient computation	Custom
	Non-Maximum Suppression	Determines if a pixel belongs to a contour	Custom
Generalized Hough Transform	Compact List	Compacts sparse contour points matrix into a dense list	Custom
	OT Generation	Search pairings for each contour point. Generates the orientation table	Custom
2D Cross Correlation	Table of Terms in a Window	Adds an element of the OT and the elements in a square window	Based on CUDA sample
	Normalization Term	Generates the normalization term of an OT	Custom
	1D Correlation	Correlates an OT and a table of terms in a window. The result is the contour similarity value	Custom
Luminance Histogram	64 bins Histogram	Generates a 64 bins histogram of an image. Per-thread sub-histograms are reduced with global memory atomic functions	Based on CUDA sample
1D Cross Correlation	Vector of Terms in a Window	Generates a vector of terms in a window. Implemented by a row convolution with a 1D filter of size 3 elements equal to 1	Based on CUDA sample
	Normalization Term	Normalization term of a histogram	Custom
	1D Correlation	Correlates a histogram and a vector of terms in a window of the next frame. The result is the luminance similarity value	Custom

4 Parallelizing the Contour Based Metric

CBM requires much more execution time than LBM. In fact, the generation of the OT is the most time consuming step. Moreover, once the edge detection has been performed, computation turns irregular. The number of contour points obviously depends on the frame. Thus, the generation of the OT is workload dependent. The challenge of this work is finding an efficient implementation, which guarantees a constant performance improvement.

4.1 Canny Edge Detection

This work implements a version of the Canny algorithm without thresholding with hysteresis, which is usually its slowest part. This version of the Canny algorithm is adapted to the abrupt and gradual transitions detection algorithm, scope of this work, and is not an entire implementation as is the recently presented in [12].

The first part of the Canny algorithm convolves the image with two Gaussian filters, in order to reduce noise. Both filters are separable, what makes easier and more efficient its implementation. This is based on a separable convolution kernel included in the CUDA SDK. The second part obtains the intensity and direction of the gradient. This computation is applied over all pixels of the image. Finally, the last part is carried out to determine if the gradient magnitude assumes a local maximum in the gradient direction. A pixel belongs to a contour, if its gradient magnitude is greater than a threshold and the gradient magnitudes of the adjacent pixel in the gradient direction. Fig. 2 explains this process.

Convolution separable and gradient calculation are regular as they are applied to all pixels in the same way. Hence, their adaptation to GPU is easy and achieves an important improvement. The last part, non-maximum suppression, presents some undesirable features for GPU, because computation depends on the gradient direction. This is implemented with conditional clauses, which involve different alternatives for the threads in a warp. Thus, some threads in the warp are not executed simultaneously, i.e. thread execution is serialized. Performance can be improved by loading data, which is being reused several times during the kernel, into shared memory. Each thread block loads a row tile, the upper row tile and the lower row tile into its shared memory, as is shown in Fig. 2.

4.2 Generalized Hough Transform

As it has been seen, Canny algorithm returns a sparse matrix of contour points. After that, computation turns very irregular along the pixels of the image, turning the parallelization on GPU pretty much difficult.

A straightforward implementation can use one thread per pixel. If the pixel is a contour point, the thread will search the rest of contour points, in order to check whether there is a pairing or not. This approach does not achieve a good performance, since it requires conditional clauses, which prevent the threads in a warp from executing simultaneously. Thus, computation will be serialized and the computing power of the GPU, wasted. In fact, most of the threads will turn idle, because only a small amount of threads correspond to a contour point. Moreover, these active threads should examine the whole image, in order to find a reduced number of contour points.

A better implementation should reorganize the input data, in order to maintain every thread active. In this paper, contour points are compacted and then assigned to the threads. Thus, the generation of the OT consists of two kernels.

Compacting the Contour Points. The first kernel compacts the contour points in a single dense list, which contains the coordinates of the contour points and their gradient directions. Each thread evaluates one pixel. If it is a contour point, the thread increments an accumulator, lied in global memory, using an atomic addition. The

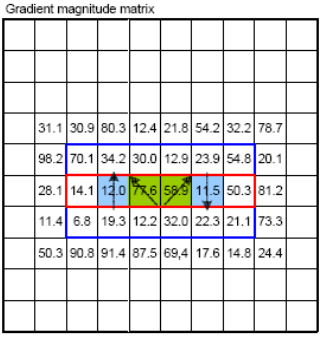

Fig. 2. Non-maximum suppression. A thread block is focused on a row tile (red), but also loads to shared memory the upper and the lower row tiles (blue). A pixel belongs to a contour if its gradient magnitude is greater in the direction of the gradient (lime). If it is not, the pixel is discarded as contour point (pale blue).

current value of the accumulator is used as index for the compact list, as is showed in Fig. 3. Thus, the size of this list is the number of contour points, which is dependent on the frame. This conversion from sparse matrix to dense list makes more efficient the subsequent data management and computation.

Generating the Orientation Table. The compact list is the input data to the second kernel. It uses a number of thread blocks, which depends on the number of contour points. While the algorithm is processing a video, the size of the thread block is fixed. Instead, the number of blocks is varying in each call to the kernel, because it is the number of contour point divided by the size of the thread block. This makes the kernel adaptable to any image. Each thread block takes a part of the compact list and loads it in shared memory. Then, it searches pairings for each contour point, comparing the gradients within its own sub-list during the first iteration. Next iterations require loading in shared memory the sub-list correspondent to other blocks, in order to compare the gradient of a contour point with any other in the list. Fig. 4 explains this process. Each time a pairing is found, the corresponding element of the OT, which resides in global memory, is incremented by using an atomic addition. Thus, the OT is generated as a square histogram.

An alternative to the use of atomic additions, which unavoidably serialize the execution, could be generating an OT per thread in shared memory. Each thread would be assigned to one contour point and would search pairings for it, in order to create its own OT. Finally, all OTs would be added by a reduction. The major drawback of this strategy is that the size of the OT is 45x45. Thus, when more than two threads are assigned to a block (typically, the number of threads per block is 64) the private OTs do not fit into shared memory and these tables need to be mapped into global memory. Taking into account that the accumulation operation generates non-coalesced memory access pattern, the high latency of the global memory will not allows to obtain good performance values.

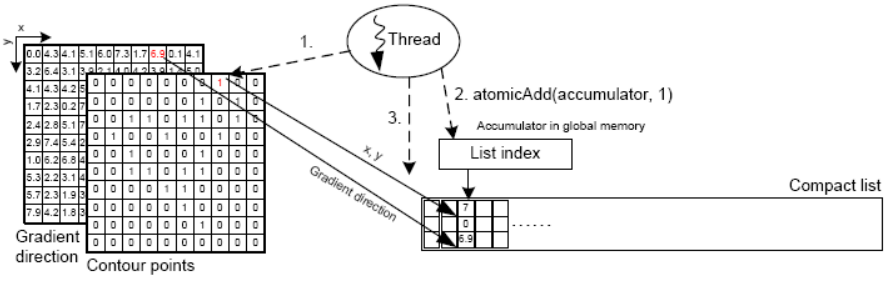

Fig. 3. Sparse matrix to dense list conversion: 1. The thread checks if the pixel is a contour point; 2. If it is, the thread increments an accumulator in global memory; 3. Copies the contour point coordinates and its gradient direction into the compact list.

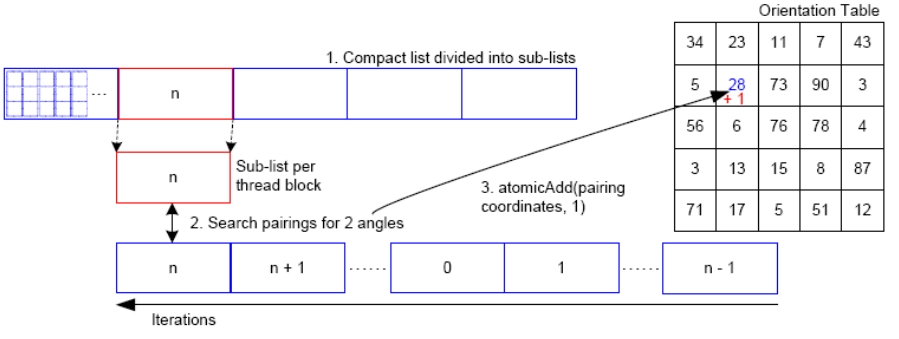

Fig. 4. OT generation: 1. Compact list is divided into sub-lists and these are assigned to each thread block; 2. The block search pairings in all the sub-lists; 3. If a pairing is found, the corresponding element of the OT in global memory is incremented using an atomic addition.

4.3 2D Cross Correlation

The preceding stage calculates the OT of an image, which is a dense matrix. Computation is regular again. During the first part of this stage, each element of the OT is windowed, i.e. it is added to the elements of a square window centered in it. This generates a table of terms in a window of the same size than the OT. This part can also be implemented using the separable convolution and two one-dimensional filters with all the elements equal to 1. The table of terms is a window of a frame is then correlated with the OT of the preceding frame and the same OT rotated clockwise and counterclockwise, resulting three similarity values. The highest of these values is the result of the kernel.

5 Experimental Results

This section presents the evaluation of the performance of the implemented algorithm. First, results on three CUDA-enables GPUs are analyzed. Then, the performance of a multi-core implementation with OpenMP on two platforms is showed, in order to compare with the results on the GPUs.

5.1 Performance on CUDA-Enabled GPUs

This implementation has been tested on three CUDA-enabled GPUs, whose features are showed in Table 2. They permit to compare the performance of the algorithm on the three more recent nVIDIA GPU generations. As it can be seen, the major difference between GeForce 8 and GeForce 9 series is the number of multiprocessors. Both 8800 GTS and 9800 GX2 have compute capability 1.1. This permits global atomic functions on 32-bit words in global memory. GeForce 200 series has compute capability 1.3, which improves features significantly. It allows atomic functions in global and in shared memory and supports double-precision floating-point numbers.

The workloads for the tests are 500 frames fragments of four MPEG videos from the MPEG-7 Content Set. Table 3 shows the average number of contour points and pairings per frame. These values have a clear influence on execution time, as it is explained below. Fig. 5 presents the execution time for the four fragments of the videos. GeForce GTX 280 improves significantly the performance of its predecessors. The main reason is the greater number of multi-processors. GeForce 9800 GX2 does not outperform significantly the results of GeForce 8800 GTS, since only one core of 16 multi-processors is used.

Table 2. Hardware and software features in nVIDIA GeForce GPUs

Parameter	8800 GTS	9800 GX2	GTX 280
Multi-Processors per GPU	12	2x16	30
Processors/Multi-processor	8	8	8
Threads/Warp	32	32	32
Threads/Block	512	512	512
Threads/Multi-Processor	768	1024	1024
Warps/Multi-procesor	24	24	32
32-bit registers/Multi-Processor	8192	8192	16384
Shared memory/Multi-Processor	16 Kbytes	16 Kbytes	16 Kbytes
Global memory	512 Mbytes	1 Gbyte	1 Gbyte

A previous analysis of the execution time of each kernel concluded that the generation of the Orientation Tables takes more than 90% of the execution time of the algorithm. This is clearly conditioned by the number of contour points and pairings. This is the main bottleneck for achieving a better performance in this application. The reason is the high latency of the global memory, which is accessed continuously in order to load sub-lists into shared memory, while searching for pairings. Moreover, once a pairing has been found, the corresponding element of the OT is incremented by using an atomic addition, which serializes the work of the threads. Although other parts of the algorithm, such as Canny, achieve a speedup of 15, the global improvement is limited by the pretty much slower OT generation.

In line with the efforts of accelerating the algorithm, *stream management* capability of CUDA has been used in order to overlap the transference of the frames from host memory to device memory. Unfortunately, this transference is less than 1% of the execution time of the kernels. Thus, the improvement is negligible.

Table 3. Test workloads characteristics. The four videos have a resolution of 352x288 pixels. Number of contour points and pairings are average values per frame, in the 500 frames fragments.

Video	Description	Contour points	Pairings
Basket	Basketball match: Images of the court and people in the stands	24902	22348724
Cycling	Cycling race: A cyclist on the road, surrounded by public	10850	5354556
Drama	Television series: Some actors in an indoor scenario	21714	19167915
Movie	Beginning of a movie: Credits on a black background	2833	615818

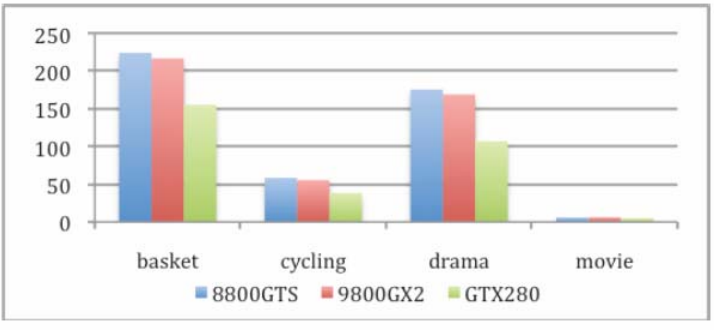

Fig. 5. Execution times of the CUDA implementation on the GPUs for the four videos. Results are represented in seconds.

5.2 An OpenMP Implementation

The abrupt and gradual transitions detection algorithm has also been implemented with OpenMP, in order to compare the performance on the GPUs and on two multiple cores platforms. Table 4 shows some features of both platforms.

The OpenMP implementation takes advantage of the Thread Level Parallelism in both platforms by following a data decomposition strategy. Parallel-for pragmas are used to assign equally the workload to the threads. Fig. 6 presents the results on both platforms for the videos in Table 3. As in the CUDA implementation, the execution time is conditioned by the number of contour points and pairings. Speedup, referred to the 1 thread version, varies in a similar way in both platforms, as shown in Table 5. It scales well for 2 and 4 threads, but falls when the contour points and the pairings decrease or when the number of threads increases, that is, the implementation performs worse when the workload per thread decreases.

Table 4. Hardware features of both multiple core platforms

Platform	8-way Intel Xeon	Intel Core2Quad
# Sockets	2	1
Cores/Socket	4	4
Clock speed	2.66 GHz	2.40 GHz
L2 Cache	4 Mbytes/2 cores	4 Mbytes/2 cores

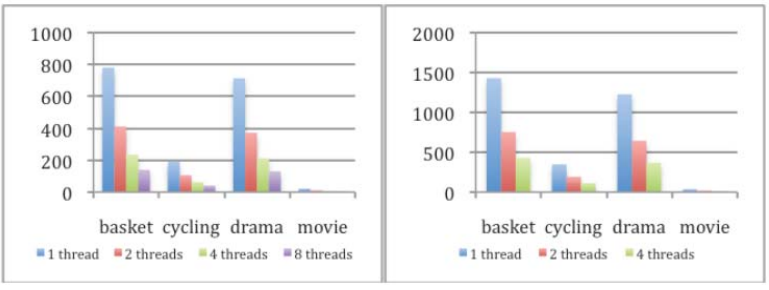

Fig. 6. Execution times of the OpenMP implementation on 8-way Intel Xeon (left) and on Intel Core2Quad (right) for the four videos. Results are represented in seconds.

Table 5. Speedup of the OpenMP implementation for both platforms

Platform	Video	2 threads	4 threads	8 threads
8-way Intel Xeon	Basket	1.90	3.32	5.57
	Cycling	1.81	3.07	4.65
	Drama	1.92	3.37	5.47
	Movie	1.83	2.54	3.61
Intel Core2Quad	Basket	1.90	3.32	-
	Cycling	1.81	3.09	-
	Drama	1.90	3.32	-
	Movie	1.83	2.53	-

5.3 Comparison between CUDA and OpenMP Implementations

The main goal of this work is the implementation of the algorithm in CUDA, but it also has been developed an implementation in OpenMP, in order to compare their performances. If the GPUs are being used as SIMD coprocessors, they should improve the results of current multi-core CPUs. In this way, if the results on the GeForce GTX 280 are compared with the ones on Intel Core2Quad and Intel Xeon, the conclusions are very encouraging.

Performance on GeForce GTX 280 achieves a speedup from 7.6 to 11.3 versus the 1 thread implementation on Intel Core2Quad, depending on the video. Moreover, the higher speedup corresponds to a greater number of pairings in the frames. Comparing with the 4 threads implementation on the same CPU, the speedup varies between 2.8 and 3.5. This is almost four times faster than one of the most recently released desktop processors.

Only the 8 threads implementation on Intel Xeon is in the same order of magnitude than GeForce GTX 280. The 8 threads implementation slightly beats the GTX 280 results for the basket video, but not for the rest. The GTX 280 improves up to 22% the execution time of the 8 threads implementation for drama and movie.

6 Conclusions

This paper has presented the parallelization of an abrupt and gradual transitions detection algorithm on CUDA-enabled GPUs. The implementation of some stages with

irregular computation and the conversion between storing formats, i.e., sparse matrix to dense list, have been the major challenge. Its evaluation on the three more recent nVIDIA GPU generations has revealed a very good performance and has shown the possibility of using GPUs for accelerating video segmentation applications.

The implementation on the recently released nVIDIA GeForce GTX 280 achieves an acceleration up to 11.3 comparing to a 1 thread implementation of the algorithm on a current desktop processor. Moreover, it is up to 3.5 times faster than a 4 thread OpenMP implementation, which has also been developed. Only an implementation on an 8-way multi-processor approaches the performance on GeForce GTX 280. This is a sample of the computing power of the GPUs and their applicability for general-purpose processing. Its use is also clearly justified in terms of costs, since the price of a current GPU is 10% of an 8-core Intel Xeon.

References

1. Sáez, E., Benavides, J.I., Guil, N.: Reliable Real Time Scene Change Detection in MPEG Compressed Video. In: IEEE International Conference on Multimedia and Expo. (2004)
2. Sáez, E., Palomares, J.M., Benavides, J.I., Guil, N.: Global Motion Estimation Algorithm for Video Segmentation. In: IS&T/SPIE Visual Communications and Image Processing (2003)
3. Compute Unified Device Architecture (CUDA), `http://www.nvidia.com/cuda`
4. Open SMP Programming (OpenMP), `http://www.openmp.org`
5. OpenVIDIA: Parallel GPU Computer Vision, `http://openvidia.sourceforge.net`
6. Farrugia, J.P., Horain, P., Guehenneux, E., Alusse, Y.: GPUCV: A Framework for Image Processing Acceleration with Graphics Processors. In: IEEE International Conference on Multimedia and Expo. (2006)
7. Canny, J.F.: A Computational Approach to Edge Detection. IEEE Trans Pattern Analysis and Machine Intelligence 8, 679–698 (1986)
8. Guil, N., González, J.M., Zapata, E.L.: Bidimensional Shape Detection using an Invariant Approach. Pattern Recognition 32, 1025–1038 (1999)
9. Sáez, E., González, J.M., Palomares, J.M., Benavides, J.I., Guil, N.: New Edge-Based Feature Extraction Algorithm for Video Segmentation. In: IS&T/SPIE Symposium, Image and Video Communications and Processing (2003)
10. Podlozhnyuk, V.: Image Convolution with CUDA. nVIDIA white paper (2007)
11. Podlozhnyuk, V.: Histogram Calculation in CUDA. nVIDIA white paper (2007)
12. Luo, Y., Duraiswami, R.: Canny Edge Detection on NVIDIA CUDA. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (2008)