Unsupervised Markovian Segmentation on Graphics Hardware

Pierre-Marc Jodoin Jean-François St-Amour Max Mignotte

Université de Montréal,
Département d'Informatique et de Recherche Opérationnelle (DIRO)
P.O. Box 6128, Studio Centre-Ville, Montréal, Québec, H3C 3J7.
E-mail: {jodoinp/stamourj/mignotte}@iro.umontreal.ca

Abstract. This contribution shows how unsupervised Markovian segmentation techniques can be accelerated when implemented on graphics hardware equipped with a Graphics Processing Unit (GPU). Our strategy exploits the intrinsic properties of local interactions between sites of a Markov Random Field model with the parallel computation ability of a GPU. This paper explains how classical iterative site-wise-update algorithms commonly used to optimize global Markovian cost functions can be efficiently implemented in parallel by *fragment shaders* driven by a *fragment processor*. This parallel programming strategy significantly accelerates optimization algorithms such as ICM and simulated annealing. Good acceleration are also achieved for parameter estimation procedures such as K-means and ICE. The experiments reported in this paper have been obtained with a mid-end, affordable graphics card available on the market.

1 Introduction

Image segmentation is generally understood as a mean of dividing an image into a set of uniform regions. Here, the concept of *uniformity* makes reference to image features such as color or lightness intensity. Among the existing classification approaches proposed in the literature, segmentation models can roughly be divided between *feature-space based* and *image-space based* families [1]. Because image-space based techniques incorporate information from the image to be segmented and the segmentation map, the results they produce are generally more precise, although at the cost of heavier computational loads.

Among the *image-space based* techniques are the Markovian algorithms [2, 3] which incorporate both image and spatial characteristics by using Markov Random Fields (MRF) as *a priori* models. The first contribution in that field came from Geman *et al.* [2] who proposed the concept of *Maximum a Posteriori* (MAP) as *image-space* criteria. While some authors proposed *ad-hoc* MAP energy-based functions, others used probabilistic functions to model the way the desired (hidden) label field is distributed. The shape of these probabilistic functions depends on parameters that are either supposed to be known (or manually adjusted) or estimated in a first step of processing. In the latter case, estimation algorithms such as Expectation Maximization (EM) or its stochastic Markovian extension called Iterative Conditional Estimation (ICE) [4, 5] have demonstrated their efficiency.

Markovian models are known to be flexible and precise. However, they are also known to be slow, especially when implemented along with a stochastic optimizer such as simulated annealing (SA) [6] and/or with a parameter estimation step. Although some deterministic optimization algorithms such as ICM [3] or HFC [7] dramatically reduce computation times, Markovian algorithms are still far from being real-time. In this contribution, we show how processing times of classical unsupervised Markovian segmentation algorithms can be significantly reduced when implemented on mid-end programmable graphics hardware equipped with a Graphical Processor Unit (GPU). Although such graphics hardware is built to process vertices, lights and textures in the context of image synthesis, many applications beyond traditional graphics have been demonstrated to run on GPUs [8–10]. Recently, some computer vision tasks, such as anisotropic diffusion, segmentation by level-set and motion estimation were successfully implemented on a GPU [10]. Parallel implementations of Markovian algorithms applied to motion detection [11] and picture restoration [12] have been already proposed in the past. Unfortunately, these methods were build upon dedicated, expensive and sometimes obsolete architectures.

The rest of the paper is organized as follows. In Section 2, a review of the Markovian segmentation theory is proposed while Section 3 and 4 present estimation and optimization algorithms. Section 5 gives a look to the graphics hardware architecture and presents how a Markovian segmentation algorithm can be implemented on such hardware. Finally, Section 6 and 7 show some experimental results and conclude.

2 Unsupervised Markovian Segmentation

Let X and Y be respectively the *label field* (the segmentation map to be estimated) and the *observation field* (the input image to be segmented). Each field is defined on a rectangular lattice of size $\mathcal{N} \times \mathcal{M}$, represented by $S = \{s \,|\, 0 \leq s < \mathcal{N} \times \mathcal{M}\}$ where s is a site located at the Cartesian position (i, j). It is common to represent a *realization* of a field with a low-case variable such as x or y. For each site $s \in S$, x_s takes a value in $\Delta = \{e_1, e_2, ..., e_N\}$ and y_s takes a value in $\Gamma = \{\epsilon_1, \epsilon_2, ..., \epsilon_\zeta\}$ ($\epsilon_1 = 0$ and $\epsilon_\zeta = 255$ for grayscale images and ϵ_i is a 3D vector with a value contained between $(0, 0, 0)$ and $(255, 255, 255)$ for color images).

In the context of the *MAP* [2], the objective of a segmentation algorithm is to estimate *the best* label field x given y or equivalently the *optimal* solution $\hat{x}_{\text{MAP}}$ which maximizes the *posterior* probability function $P(X = x|Y = x)$ (written $P(x|y)$ to simplify notation). In accordance with Bayes theorem, the optimal label field is obtained when

$$\hat{x}_{\text{MAP}} = \arg\max_x \frac{P(y|x)P(x)}{P(y)} \tag{1}$$

where $P(y|x)$ is the likelihood, $P(x)$ the prior and $P(y)$ the evidence. Since $P(y)$ isn't related to x, without lost of generality, this equation can be simplified to $\hat{x}_{\text{MAP}} = \arg\max_x P(y|x)P(x)$.

If X and Y are MRFs, according to the Hammersley-Clifford theorem, the likelihood and prior probability functions have a Gibbsian shape, respectively, $P(y|x) \propto \exp\{-W(x,y)\}$ and $P(x) \propto \exp\{-V(x)\}$, where $W(x,y)$ and $V(x)$ are *energy* functions. Incorporating these two Equations to the MAP framework leads to the optimization formulation $\hat{x}_{\text{MAP}} = \arg\min_x\{W(x,y) + V(x)\}$. Assuming that the noise in the observed image y is uncorrelated, since X and Y are MRFs, the global energy functions $W(x,y)$ and $V(x)$ can be represented by a sum of *local* energy functions

$$\hat{x}_{\text{MAP}} = \arg\min_x \sum_{s\in S}\{W_s(x_s, y_s) + V_{\eta_s}(x_s)\} \tag{2}$$

where η_s stands for the neighborhood around site s (in this contribution, we use a second-order neighborhood). V_{η_s} is a sum of potential functions of the form $V_{\eta_s} = \sum_{c\in C_s} V_c(x_s)$, where C_s is the set of binary cliques linking s to sites $r \in \eta_s$. Here, the Potts model was used to represent V_{η_s}.

In the case of a *probabilistic* segmentation, input data y_s is related to a class x_s according to a distribution $P(y_s|x_s)$. Consequently, the energy function $W_s(x_s, y_s)$ has to be designed according to that distribution, namely $W_s(x_s, y_s) \propto -\ln P(y_s|x_s)$. A very popular function used to model $P(y_s|x_s)$ is the multidimensional Gaussian distribution

$$P(y_s|x_s) = \frac{1}{\sqrt{(2\pi)^d|\Sigma_{x_s}|}} \exp\{-\frac{1}{2}(y_s-\mu_{x_s})\Sigma_{x_s}^{-1}(y_s-\mu_{x_s})^{\text{T}}\}$$

where d is the dimensionality of y_s ($d = 3$ for color images and $d = 1$ for gray-scale images) and $(\mu_{x_s}, \Sigma_{x_s})$ are the mean and variance-covariance of class x_s. Thus, the energy function of Eq. (2) can be written as

$$\sum_{s\in S}\{\underbrace{\frac{1}{2}(\ln|\Sigma_{x_s}| + (y_s-\mu_{x_s})\Sigma_{x_s}^{-1}(y_s-\mu_{x_s})^{\text{T}})}_{W_s(x_s,y_s)} + V_{\eta_s}(x_s)\}.$$

In the case of *unsupervised* segmentation, the Gaussian parameters $\Phi = \{(\mu_i, \sigma_i)\,|\,1 \le i < N\}$ has to be estimated conjointly with x or preliminary to the segmentation step. Many parameter estimation algorithms are available among which EM, K-means and ICE [5] are the most popular.

3 Parameter Estimation

The two parameter estimation algorithms we have implemented are K-means and ICE [5]. K-means is an iterative clustering method [4] that assumes input data $\{y_s\}$ are distributed within K spherical clusters of equal volume. At each iteration, every site s are assigned to the nearest cluster before a second step re-estimates the center of mass of every cluster. The resulting K-means clustering minimizes the sum-of-square error function $\sum_{i=1}^{N}\sum_{x_s=e_i}||y_s-\mu_i||^2$

1	$\mu_i \leftarrow$ random initialization , $\forall \mu_i \in \Phi_\mu$
2	For each site $s \in S$
2a*	$x_s \leftarrow \arg\min_{e_i \in \Gamma} \|\|y_s - \mu_{e_i}\|\|^2$
3	$\mu_i \leftarrow \frac{1}{N_i} \sum_{x_s = e_i} y_s \ , \forall \mu_i \in \Phi_\mu$
4	Repeat steps 2-3 until each mean μ_i no longer moves
5	$\Sigma_i^{nm} \leftarrow \frac{1}{N_i} \sum_{x_s = e_i} (y_s^n - \mu_{e_i}^n)(y_s^m - \mu_{e_i}^m) \forall \Sigma_i \in \Phi_\Sigma$

1	$\Phi \leftarrow K$-means
2	For each site $s \in S$
2a*	$P(e_i \| y_s) = \frac{1}{Z_s} \exp\{(W(e_i, y_s) + V_{\eta_s}(e_i))\} \forall e_i \in \Gamma$
2b*	$x_s \leftarrow$ according to $P(x_s \| y_s)$, randomly select $e_i \in \Gamma$
3a	$\mu_i \leftarrow \frac{1}{N_i} \sum_{x_s = e_i} y_s \ \forall \mu_i \in \Phi_\mu$
3b	$\Sigma_i^{nm} \leftarrow \frac{1}{N_i} \sum_{x_s = e_i} (y_s^n - \mu_{e_i}^n)(y_s^m - \mu_{e_i}^m) \forall \Sigma_i \in \Phi_\Sigma$
5	Repeat steps 2-3 until Φ no longer changes

Table 1. K-means and ICE algorithms. Here $n, m \in [1, d]$.

[4]. The variance-covariance of each cluster is estimated once the algorithm has converged.

Because K-means is a deterministic algorithm, it is sensitive to noise and is likely to converge toward local minima. Furthermore, its assumption that all clusters are spherical with equal volume is simplistic an often unsuited to some observed images. Thus, many authors suggest to refine Φ with a more realistic model, less sensitive to noise and local minima such as the stochastic ICE estimation algorithm. Details of this algorithm are presented in [5] while Table 1 presents a version adapted to this paper.

4 Optimization Procedures

Because Eq. (2) has no analytical solution, it has to be solved with an optimization algorithm such as simulated annealing (SA) [2] or ICM [3]. SA is a stochastic algorithm built upon a temperature variable that slowly decreases toward zero with time. If the cooling rate is small enough, this annealing schedule theoretically guarantees the convergence to the global MAP. The ICM algorithm is a hill climbing deterministic algorithm that isn't guarantied to converge toward global minima. However, it is drastically faster than SA and generates fairly good results when properly initialized. As Besag mentioned [3], it can be understood as an instantaneous freezing in SA. Both algorithms are presented in Table 2.

1	$T \leftarrow T_{\text{MAX}}$
2	For each site $s \in S$
2a*	$P(e_i \| y_s) = \frac{1}{Z_s} \exp\left\{\frac{1}{T}(W(e_i, y_s) + V_{\eta_s}(e_i))\right\}, \forall e_i \in \Gamma$
2b*	$x_s \leftarrow$ according to $P(x_s \| y_s)$, randomly select $e_i \in \Gamma$
3	$T \leftarrow T *$coolingRate
5	Repeat steps 2-3 until $T \leq T_{\text{MIN}}$

1	Initialize x (with ICE and/or K-means)
2	For each site $s \in S$
2a*	$x_s = \arg\min_{e_i \in \Gamma} (W(e_i, y_s) + V_{\eta_s}(e_i))$
3	Repeat steps 2 until x stabilizes

Table 2. Simulated annealing and ICM algorithms.

5 Graphics Hardware Architecture

Graphics hardware is highly optimized to solve traditional computer graphics problems. Nowadays, graphics hardware is most of the time embedded on a graphics card which can receive/send data from/to the CPU or the main memory via the system bus, be it PCI, AGP or PCIe. Most graphics hardware are designed to fit the so-called *graphics processing pipeline* [13, 14]. This pipeline is made of various stages which sequentially transform images and geometric input data into an output image stored in a section of graphics memory called the *framebuffer*. Part of the framebuffer (the front buffer) is meant to be visible on the display device.

During the past few years, the major breakthrough came when the vertex processing and fragment processing stages have been made *programmable*. These two stages can now be programmed using C-like languages to process vertex and fragments in parallel. Let us mention that a fragment is a per-pixel data structure created at the rasterization stage and containing data such as color, texture coordinates and depth. A fragment is meant to update a unique location in the framebuffer. Because the GPU is a *streaming processor* (i.e. a processor with inherent parallel processing abilities) mapping general computation problems to its unique architecture becomes very interesting [10].

1 Copy the input image y into texture memory.
2 Compile, link and load the ICM shader on the GPU.
3 Specify shader parameters ($N, \mathcal{N}, \mathcal{M}, \Phi$ for example).
4 Render a rectangle covering a window of size $\mathcal{N} \times \mathcal{M}$
5 Copy the framebuffer into texture memory
6 Repeat steps 4 and 5 until convergence
7 Copy the framebuffer into a C/C++ array if needed

1 $\hat{x}_s \leftarrow \arg\min_{e_i \in \Gamma} W(e_i, y_s) + V_{\eta_s}(e_i)$
2 framebuffer$_s \leftarrow \hat{x}_s$

1 Copy the input image y into texture memory.
2 Compile, link and load the K-means shader on the GPU.
3 $\Phi \leftarrow$ Init Gaussian parameters.
4 Specify shader parameters ($N, \mathcal{N}, \mathcal{M}$ and Φ_μ).
5 Render a rectangle covering a window of size $\mathcal{N} \times \mathcal{M}$
6 $E \leftarrow$ Copy the framebuffer into a C/C++ array.
7 $\mu_i \leftarrow \frac{1}{N_i} \sum_{E_s = e_i} y_s, \quad \forall \mu_i \in \Phi_\mu$
8 Repeat steps 4 to 7 until convergence.
9 $\Sigma_i^{nm} \leftarrow \sum_{E_s = e_i} (y_s^n - \mu_{e_i}^n)(y_s^m - \mu_{e_i}^m), \quad \forall \Sigma_i \in \Phi_\Sigma$
10 Copy the framebuffer into a C/C++ array if needed

1 $x_s \leftarrow \arg\min_{e_i \in \Gamma} \|\|y_s - \mu_{e_i}\|\|^2$
2 framebuffer$_s \leftarrow x_s$

Table 3. *High level representation of ICM and K-means hardware programs. The upper sections (line* 1 *to* 7 *and* 1 *to* 10*) are C/C++ CPU programs used to load the shader, render the scene and manage textures. The lower sections (line* 1-2*) are the fragment shaders launched on every fragment (pixel) when the scene is rendered (line* 4 *and* 5*).*

A fragment processor is designed to load and execute in parallel a program (also called a *shader*) on each fragment generated during the rasterization stage

[13, 15]. Thus, a fragment shader is executed whenever a graphics primitive such as a polygon or a line is rendered. To be effective though, the shader must be initially loaded, compiled and linked on the GPU. This is illustrated by the two C/C++ programs of Table 3[1]. The first algorithm represents an ICM program whereas the second one represents a K-means program. The first section of these programs (line 1 to 7 and line 1 to 10) is written in C/C++ and runs on the CPU. This section essentially compiles, links and loads the shader, renders a graphics primitive and manipulates texture memory. Its crucial to understand that the shader (as opposed to traditional CPU programs) is loaded, compiled, linked and executed *during the runtime* execution of the C/C++ program. The shader (second section of Table 3) is launched on every fragment when the primitive –here a rectangle polygon– is rendered (line 4 and 5). After the primitive has been rendered, the results returned by the shaders is located in the framebuffer. This buffer can be copied in another section of the graphics memory (line 6 of the ICM code) or transfered back into central memory (line 7 of ICM code and lines 6 and 10 of K-means code). This last operation involves data traffic on the system bus and thus induces significant latency.

5.1 General-Purpose Computation on the GPU

The fragment processor is better suited for image processing problems than the vertex processor, simply because it is the only part of the graphics pipeline that has access to both input memory (texture memory) and output memory (the framebuffer). Although fragment shaders can be written in C-like languages [13, 15], they have some specificities as compared to ordinary C/C++ programs. The most important ones are the following:

1. a fragment shader is made to process every fragment in parallel;
2. the only memory in which a fragment shader can write into is the write-only *framebuffer* and *depthbuffer*;
3. the only data a fragment shader can read is contained in the texture memory, in built-in variables or in user-defined variables. As such, it cannot read the content of the framebuffer or the depthbuffer;
4. since fragments are processed in parallel, fragment shaders cannot exchange information. GPUs do not provide its shaders with access to general-purpose memory.

With such specificities, minimizing a global Markovian energy function such as Eq. (2) can be tricky. In fact, three main problems have to be overcome. The first problem is to make sure the rasterization stage generates one fragment for each pixel of the input image y. Such one-to-one mapping from the input pixels to the output buffer is achieved by rendering a screen-aligned rectangle covering a *window* with exactly the same size than the input image (see line 4 and 5 of

[1] Although other CPU programming languages such as JAVA could be used, C and C++ are by far the most widely utilized at the moment.

Table 3). In this way, the rasterization stage generates $\mathcal{N} \times \mathcal{M}$ fragments, one for each input pixel y_s.

The second problem comes from the fourth limitation. Since GPUs provide no general-purpose memory, one might wonder how can the prior energy function V_{η_s} have access to the labels x_t contained in the (write-only) framebuffer. This situation is handled by coping the framebuffer (i.e. the section of texture memory containing the label field x computed after an iteration) into texture memory (line 5 of ICM, Table 3). In this way, at the next iteration, the texture memory (which can be read by the fragment shader) will contain the label field x computed during the previous iteration. Thus, V_{η_s} is computed with labels iteratively updated and not sequentially updated as it is generally the case. Such strategy was already proposed by Besag [3]. As observed by some authors [11], the difference between these two updating schemes is very narrow, although the former might infer some small energy oscillations.

The last problem with shaders comes with their inability to generate random numbers such as needed by the stochastic algorithms SA and ICE. As a workaround, we generate an image containing random values at the beginning of the CPU application. This random image is then copied in texture memory where the shader can access it. Although this strategy isn't as efficient as a good random number generator, the results generated are very close to the ones obtained with standard CPU programs.

5.2 ICM and SA on Graphics Hardware

As shown in Table 3, y is first copied into texture memory. A fragment program is then launched on every pixel in order to solve Eq. (2) (line 4 of ICM, Table 3). The output labels are then copied in the framebuffer. Because the next ICM iteration needs the newly computed label field to proceed, the framebuffer content is copied back to the texture containing the label field information. This operation is extremely efficient because no data needs to be transmitted between the GPU and the CPU. The SA method is implemented in a manner very similar to the ICM algorithm, the only difference being that it requires a random function inside the fragment program. This situation is handled with the workaround presented in the previous Section. Notice that because fragment programs can only write in the framebuffer during a rendering pass, multiple rendering passes are used to simulate ICM/SA iterations.

5.3 *K*-means and ICE on Graphics Hardware

Unlike the optimization methods, K-means and ICE are not perfectly suited to a mapping to the GPU. While the first step of these algorithms (assigning *the best* label x_s to each image pixel, line $2a$ and $2a, 2b$ of Table 1) is perfectly implementable in parallel, the second step (Gaussian parameters computation, line 3 and $3a, 3b$ of Table 1) is not. As such, we have to take a simple hybrid approach: execute line 2 on the GPU (parallel processing) and line 3 on the CPU (sequential processing).

To do so, the input image y is first copied in texture memory so it is accessible by the fragment processor. A fragment program is then activated for each pixel that determines in parallel the *best* class e_i for that pixel. The result is then outputted in the framebuffer. Once every pixel have been assigned a label (line 2), the Gaussian parameters for every class now need to be recomputed (line 3). Because this operation can't be parallelized, the framebuffer image containing the current class of each pixel is read back to CPU memory, where the computation takes place. Once the parameters are re-estimated, they are passed back to the GPU after which a new iteration can begin. This hybrid approach is illustrated by the K-means algorithm of Table 3.

6 Experimental Results

We first implemented the four algorithms presented in Section 3 and 4 in C++. Then, we adapted these programs to the graphics hardware architecture by replacing with Cg code[2] the instructions identified with a star ($*$) in Table 1 and 2. We used OpenGL to render the polygon and manage texture memory, and used the Cg Runtime Library [15] to load, compile and link the fragment shader. The software and hardware version of these programs run in the same C++ environment and thus, can be fairly compared.

The performances of each implementation was evaluated by varying the number of segmentation classes and the size of the images to be segmented. Processing times have been obtained by averaging results obtained after segmenting several grayscale and color images. The acceleration factor between the software and hardware version of the programs is presented in Fig. 1. In the leftmost graphics, the programs were launch over images of size ranging from 64×64 to 1024×1024 with a number of classes set to 4. In the other graphics, results were obtained after segmenting images of size 256×256 with different number of classes.

The SA parameters T_{MAX}, T_{MIN} and the cooling rate were respectively set to 10.0, 0.05 and 0.99. This setting corresponds to a total of 500 iterations as opposed to 10 iterations for K-means, ICE, and ICM. Let us note that the number of iterations for the software and hardware implementations is exactly the same. The results were obtained on a computer equipped with AMD Athlon 64 Processor 3200+ and an NVIDIA GeForce 6800 GT graphics card.

Notice that the speedup factor between hardware and software version of ICM and SA (between 20 and 120) is more important than the one for K-means and ICE (between 2 and 8). This can be explained by the fact that both K-means and ICE algorithms have to exchange information (for the Gaussian parameter estimation) with the CPU which is major bottleneck for such hardware programs. Also, the speedup factor for K-means is larger than for ICE because ICE has to estimate and invert the variance-covariance matrix which isn't required for K-means. This extra load on the CPU makes ICE less efficient than K-means. Similarly, the speedup factor for SA and ICM is more important on color images

[2] Cg is a C-like hardware language program developed by NVIDIA.

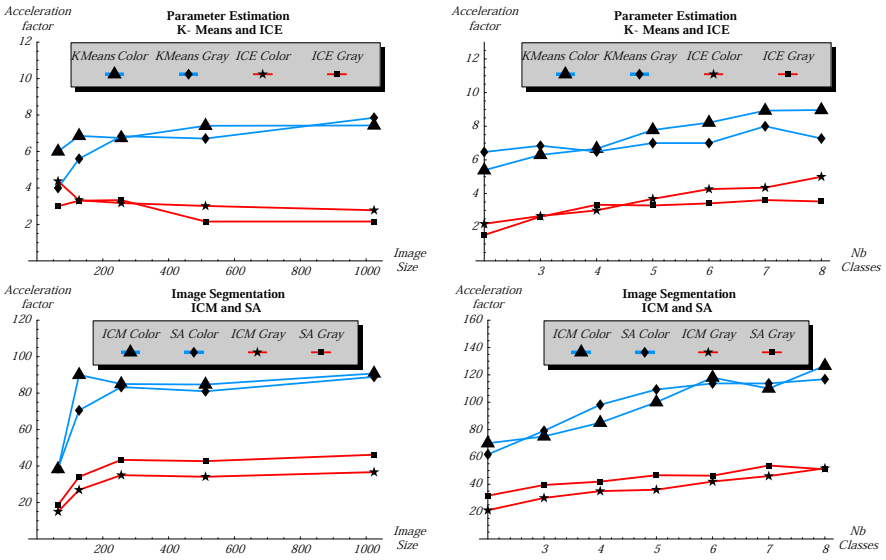

Fig. 1. Acceleration factor for K-means, ICE, SA and ICM obtained on grayscale and color images.

than on grayscale images. This is explained by the fact that the global energy function of Eq. (2) is more expensive to compute for color images than for grayscale images. Thus, parallelizing this costly CPU operation leads to a more important acceleration factor. Notice that the acceleration factor is larger when segmenting large images and/or segmenting images with many classes.

With our actual hardware implementation, a color image of size 128×128 is segmented in 4 classes at a rate of 76 fps for ICM, 1.4 fps for SA, 2.5 fps for ICE and 14 fps for K-means. These frame rates do not however include the time needed to load, compile and link the shaders which vary between 0.1 second and 5 seconds. Although this might seem prohibitive, the initialization step is done only once at the beginning of the program. In this way, when segmenting more than one image (or segmenting an image of size larger than 128×128), this initialization time soon gets negligible as compared to the acceleration factor.

7 Conclusion

This paper exposed how Markovian algorithms devoted to image segmentation can be significantly accelerated when implemented on programmable graphics hardware. Even if GPUs were built to process traditional graphics primitives, we demonstrated how fragment programs can be adapted to the context of Markovian estimation and optimization algorithms such as K-means, ICE, ICM and SA. The acceleration factor between software and hardware implementation was more impressive for the optimization algorithms (between 20 and 120) than the estimation ones (between 2 and 8). Results have shown that remarkably fast

optimization was achievable, especially over large images and/or with a large number of classes.

As future work, we plan to implement on graphics hardware energy-based computer vision tasks such as motion estimation, motion segmentation and stereovision. Because these tasks can be defined on a Markovian framework similar to the one presented in this paper, we have good reasons to believe that the hardware version of these algorithms will be more efficient that its software counterpart. We also look forward to implement and compare the most popular optical flow techniques on graphics hardware (Horn and Schunck, Lucas Kanade, Anandan, etc.[16]).

References

1. Lucchese L. and Mitra S. Color image segmentation: A state-of-the-art survey. In *Proc. of INSA-A*, 2003.
2. Geman S. and Geman D. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Machine Intell.*, 6(6):721–741, 1984.
3. Besag J. On the statistical analysis of dirty pictures. *J. Roy. Stat. Soc.*, 48(3):259–302, 1986.
4. Bishop C. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.
5. Pieczynski W. Statistical image segmentation. *Machine Graphics and Vision*, 1(1):261–268, 1992.
6. Kirkpatrick S., Gelatt C., and Vecchi M. Optimization by simulated annealing. *Science*, 220, 4598:671–680, 1983.
7. Chou P. and Brown C. The theory and practice of bayesian image labeling. In *ICCV*, pages 185–210, 1990.
8. Kruger J. and Westermann R. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.
9. Moreland K. and Angel E. The fft on a gpu. In *proc. of Workshop on Graphics Hardware*, pages 112–119, 2003.
10. http://www.gpgpu.org/.
11. Dumontier C., Luthon F., and Charras J-P. Real-time dsp implementation for mfr-based video motion detection. *IEEE Trans. on Img. Proc.*, 8(10):1341–1347, 1999.
12. Murray D., Kashko A., and Buxton H. A parallel approach to the picture restoration algorithm of geman and geman on a simd machine. *Image and Vision Computing*, 4:141–152, 1986.
13. Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley, 1st edition, 2004.
14. Tomas Akenine-Möller and Eric Haines. *Real-time Rendering*. AK Peters, 2e edition, 2002.
15. Fernando R. and Kilgard M. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.
16. Barron J., Fleet D., and Beauchemin S. Performance of optical flow techniques. *Int. J. Comput. Vision*, 12(1):43–77, 1994.