# Markovian segmentation and parameter estimation on graphics hardware

**Pierre-Marc Jodoin**
**Max Mignotte**
Université de Montréal
Département d'informatique et de recherche opérationnelle
C.P. 6128, succ. Centre-Ville
Montréal, Québec
Canada H3C 3J7
E-mail: jodoinp@iro.umontreal.ca

**Abstract.** *In this paper, we show how Markovian strategies used to solve well-known segmentation problems such as motion estimation, motion detection, motion segmentation, stereovision, and color segmentation can be significantly accelerated when implemented on programmable graphics hardware. More precisely, we expose how the parallel abilities of a standard graphics processing unit usually devoted to image synthesis can be used to infer the labels of a segmentation map. The problems we address are stated in the sense of the maximum a posteriori with an energy-based or probabilistic formulation, depending on the application. In every case, the label field is inferred with an optimization algorithm such as iterated conditional mode (ICM) or simulated annealing. In the case of probabilistic segmentation, mixture parameters are estimated with the K-means and the iterative conditional estimation (ICE) procedure. For both the optimization and the parameter estimation algorithms, the graphics processor unit's (GPU's) fragment processor is used to update in parallel every labels of the segmentation map, while rendering passes and graphics textures are used to simulate optimization iterations. The hardware results obtained with a mid-end graphics card, show that these Markovian applications can be accelerated by a factor of 4 to 200 without requiring any advanced skills in hardware programming. © 2006 SPIE and IS&T.* [DOI: 10.1117/1.2238881]

## 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, depth, or motion. Among the existing segmentation methods proposed in the literature, segmentation models can roughly be divided between feature-space-based and image-space-based families.[1] As the name suggests, feature-space algorithms segment images on the basis of a feature criteria, whereas space-based algorithms segment images on the basis of a feature-space criteria. Because image-space-based techniques incorporate informations from the image to be segmented and the segmentation map (typically likelihood and prior informations), the results they produce are often more precise, although at the cost of a heavier computational load.

Paper 05132R received Jul. 8, 2005; revised manuscript received Oct. 7, 2005; accepted for publication Jan. 6, 2006; published online Sep. 12, 2006.
1017-9909/2006/15(3)/033005/15/$22.00 © 2006 SPIE and IS&T.

Among the image-space-based techniques are the Markovian algorithms,[2,3] which incorporate spatial characteristics by using Markov random fields (MRFs) 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 an image-space probabilistic criterion. Shortly afterward, many Markovian models were adapted to solve all kinds of segmentation problems going from motion estimation to stereovision.[4] While some authors proposed *ad hoc* energy-based solutions, 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 *K*-means or the iterative conditional estimation (ICE)[5,6] have demonstrated their efficiency.

Although Markovian approaches have several appealing advantages, they are relatively slow algorithms and thus inappropriate to applications for which time is an important factor. This paper explains how Markovian segmentation algorithms used to solve segmentation problems such as motion detection,[7] motion estimation,[8] stereovision,[9] color segmentation,[1] and motion segmentation can be significantly accelerated when implemented on programmable graphics hardware. This is made possible by the use of the so-called graphics processor unit (GPU) embedded on most graphics hardware nowadays available on the market. This unit can load and execute in parallel general purpose programs independently of the CPU and the central memory. As the name suggests, the GPU architecture was optimized to solve typical graphics problems with the goal of rendering complex scenes in real time. Because of the very nature of conventional graphics scenes, graphics hardware have been designed to efficiently manipulate texture, vertices, and pixels. What makes these graphics processors so efficient is their fundamental ability to process vertices and fragments (see pixels) in parallel, involving interesting acceleration factors.

In spite of appearances, it is possible to take advantage of the parallel abilities of programmable graphics hardware to solve problems that goes beyond graphics. This is what people call general-purpose computation on GPUs

(GPGPUs).[10] Some authors have shown that applications such as fast Fourier transforms,[11] linear algebra,[12] motion estimation, and spatial segmentation with level sets could run on graphics hardware.[10,13] Even if these applications have little in common with traditional computer graphics, they all share a common denominator: they are problems that can be solved by parallel algorithms.

Parallel implementations of Markovian algorithms applied to motion detection[14] and picture restoration[15] have been proposed in the past. Unfortunately, these methods were build on dedicated, expensive, and sometimes obsolete architectures. This paper shows how reasonably priced and widely distributed graphics hardware can be used to significantly accelerate Markovian segmentation. Even if GPUs are cutting edge technologies made for graphics rendering, this contribution shows that implementing a MAP segmentation algorithm on a fragment processor is not much more difficult than writing it in a C-like procedural language.

The remainder of the paper is organized as follows. Section 2 reviews the proposed Markovian segmentation theory before Sec. 3 presents three energy-based segmentation problems. Section 4 follows with the probabilistic (and unsupervised) problem of motion and color segmentation before Secs. 5 and 6 present the iterated conditional mode (ICM) and simulated annealing (SA) optimization algorithms and the parameter estimation algorithms $K$-means and ICE. Section 6 gives an in-depth look to the graphics hardware architecture and exposes how the algorithms presented so far can be implemented on a graphics hardware. Finally, Sec. 8 shows experimental results and Sec. 9 concludes the paper.

## 2 Markovian Segmentation

The applications this contribution tackles aim at subdividing observed input images into uniform regions by grouping pixels having features in common such as color, motion, or depth. Starting with some observed data $Y$ (which is typically one or more input images), the goal of a segmentation program is to infer a label field $X$ containing class labels (i.e., labels indicating whether a pixel belongs or not to a moving area or a certain depth for instance). Here $X$ and $Y$ are generally defined over 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 $(i,j)$ (for simplicity, $s$ is sometimes defined as a pixel). It is common to represent by a lower-case variable such as $x$ or $y$, a realization of the label field or the observation field. For each site $s\in S$, its corresponding element $x_s$ in the label field takes a value in $\Gamma=\{e_1,e_2,\ldots,e_N\}$, where $N$ is the total number of classes. In the case of motion detection for example, $N$ can be set to 2 and $\Gamma=\{\text{StaticClass},\text{Mobileclass}\}$. Similarly, the observed value $y_s$ takes a value in $\Lambda=\{\epsilon_1,\epsilon_2,\ldots,\epsilon_\zeta\}$, where $\zeta$ can be set, for instance, to $2^8$ for gray-scale images and $2^{24}$ for color images.

In short, a typical segmentation model is made of two fields $x$ and $y$, i.e., an observation field $(y)$ that is to be decomposed into $N$ classes by inferring a label field $(x)$.

In the context of this paper, the goal is to find an optimal labeling $\hat{x}$ that maximizes the *a posteriori* probability $P(X=x|Y=y)$ [represented by $P(x|y)$ for simplicity], also called the MAP estimate:[2] $\hat{x}_{\text{MAP}}=\arg\max_x P(x|y)$. With Bayes theorem, this equation can be rewritten as

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

or equivalently $\hat{x}_{\text{MAP}}=\arg\max_x P(y|x)P(x)$, since $P(y)$ is not related to $x$. Assuming that $X$ and $Y$ are MRF, and according to the Hammersley-Clifford theorem,[2] the *a posteriori* probability $P(x|y)$—as well as the likelihood $P(y|x)$ and the prior $P(x)$—follows a Gibbs distribution, namely,

$$P(x|y)=\frac{1}{\lambda_{x|y}}\exp\left[-U(x,y)\right], \tag{2}$$

where $\lambda_{x|y}$ is a normalizing constant, and $U(x,y)$ is an energy function. Combining Eqs. (1) and (2), the optimization problem at hand can be formulated as an energy minimization problem, i.e.,

$$\hat{x}_{\text{MAP}}=\arg\min_x\left[W(x,y)+V(x)\right], \tag{3}$$

where $W(x,y)$ and $V(x)$ are, respectively, the likelihood and prior energy functions. If we assume that the noise in $y$ is not correlated, the global energy function $U(x,y)$ can be represented by a sum of local energy functions

$$\hat{x}_{\text{MAP}}=\arg\min_x\sum_{s\in S}\left[W_s(x_s,y_s)+V_{\eta_s}(x_s)\right]. \tag{4}$$

Here, $\eta_s$ is the neighborhood around site $s$, and $V_{\eta_s}(x_s)=\sum_{c\in\eta_s}V_c(x_s)$ is a sum of potential functions defined on so-called cliques $c$. Notice that function $V_c(x_s)$ defines the relationship between two neighbors in $c$, a binary clique linking a site $s$ to a neighbor $r$.

An *ad hoc* definition of $W_s(x_s,y_s)$ and $V_{\eta_s}(x_s)$ leads to a so-called energy-based segmentation. By opposition, when $W_s(x_s,y_s)$ and $V_{\eta_s}(x_s)$ are defined by a probabilistic law linking $x_s$ to $y_s$, the segmentation is called probabilistic. In both cases, though, $\hat{x}_{\text{MAP}}$ is estimated with an optimization procedure such as SA or ICM, which are typically slow algorithms. Details of these algorithms are discussed in Sec. 5.

## 3 Energy-Based Segmentation

This section shows how typical computer vision problems can be expressed as the minimum of a global energy function made of a likelihood and a prior term.

### 3.1 *Motion Detection*

The goal of motion detection is to segment a video sequence into mobile and static regions. For this kind of application, moving pixels are typically the ones with a nonzero displacement, no matter what direction or speed they might have. Motion detection is thus a particular case of the more general motion segmentation problem. The solution presented in this section was inspired by the work of Bouthemy and Lalande,[7] which proposed one of the first energy-based Markovian solution to that problem.

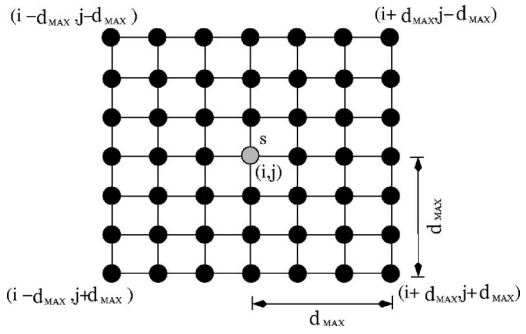


**Fig. 1** Total number of possible displacement vector for a site $s \in S$ is $(2d_{\max}+1)^2$.

**Fig. 2** Motion detection, motion estimation, and stereovision label fields obtained after a Markovian optimization.

Note that their paper influenced many subsequent contributions including the one by Dumontier *et al.*,[14] who proposed a parallel hardware architecture to detect motion in real time. Unfortunately, the hardware they used was specifically designed and is not, to our knowledge, available on the market. In addition, the design of their hardware architecture is different from standard graphics hardware.

The solution proposed here is based on the concept of temporal gradient and does not require the estimation of an optical flow. From two successive frames $f(t)$ and $f(t+1)$, the observation field $y$ is defined as the temporal derivative of the intensity function $\mathrm{d}f/\mathrm{d}t$, namely, $y=\|f(t+1)-f(t)\|$. Assuming that scene illumination variation is small, the likelihood energy function linking the observation field to the label field is defined by

$$W(x_s,y_s)=\frac{1}{\sigma^2}(y_s-m_p x_s)^2, \tag{5}$$

where $m_p$ is a constant, and $\sigma$ is the variance of the Gaussian noise. Because of the very nature of the problem, $N=2$ and $x_s \in \{0,1\}$, where 0 and 1 correspond to static and moving labels. As for the prior energy term, as in Refs. 7 and 14, the following Potts model was implemented

$$V_c(x_s)=\begin{cases}1 & \text{if } x_s \neq x_r\\ 0 & \text{otherwise.}\end{cases} \tag{6}$$

The overall energy function to be minimized is thus defined by

$$U(x,y)=\sum_{s\in S}\left[\underbrace{\frac{1}{\sigma^2}(y_s-m_p x_s)^2}_{W(x_s,y_s)}+\beta_{\mathrm{MD}}\underbrace{\sum_{c\in\eta_s}V_c(x_s)}_{V_{\eta_s}(x_s)}\right], \tag{7}$$

where $\eta_s$ is a second-order neighborhood (eight neighbors), and $\beta_{\mathrm{MD}}$ is a constant. Note that this solution makes the implicit assumption that the camera is still and that moving objects were shot in front of a static background. To help smooth out interframe changes, one can add a temporal prior energy term $V_\tau(x_s)$ linking label $x_s$ estimated at time $t$ and the one estimated at time $t-1$.

### 3.2 *Motion Estimation*

The goal of motion estimation is to estimate the direction and magnitude of the optical flow observed over each site $s \in S$ of an animated sequence.[16–18] Among the solutions proposed in the literature, many are based on an hypothesis called lightness consistency. This hypothesis stipulates that a site $s \in S$ at time $t$ keeps its intensity after it moved to site $s+\boldsymbol{v}_s$ at time $t+1$. Although this hypothesis excludes noise, scene illumination variation, and occlusions (and thus is an extreme simplification of the true physical nature of the scene) it enables simple energy functions to generate fairly accurate results. Under the terms of this hypothesis, the goal of motion estimation is to find, for each site $s \in S$, an optical displacement vector $\boldsymbol{v}_s$ for which $f_s(t) \approx f_{s+\boldsymbol{v}_s}(t+1)$. In other words, the goal is to find a vector field $\hat{v}$ for which

$$\hat{\boldsymbol{v}}_s=\arg\min_{\boldsymbol{v}_s}\|f_s(t)-f_{s+\boldsymbol{v}_s}(t+1)\|, \quad \forall\, s\in S. \tag{8}$$

Notice that the absolute difference could be replaced by a cross-correlation distance for more robustness. Such strategy is sometimes called in the literature "region-based matching."[18] When estimating motion, the observation field $y$ is the input image sequence $f$ and $y(t)$ is a frame at time $t$. Furthermore, the label field $x$ is a vector field made of 2-D vectors defined as $x_s=\boldsymbol{v}_s=(\zeta_i,\zeta_j)$, where $\zeta_i$ and $\zeta_j$ are integers taken between $-d_{\max}$ and $d_{\max}$, as shown in Fig. 1.

However, Eq. (8) has one major limitation which comes from the fact that real-world sequences contain textureless


**Fig. 3** Difference between a label field $x$ estimated with ICM (7 iterations) and SA (247 iterations).

**Table 1** SA and ICM algorithms.

| | |
|---|---|
| 1 | $T \leftarrow T_{\max}$ |
| 2 | For each site $s \in S$ |
| 2a | $P(x_s = e_k \mid y_s) = \frac{1}{\lambda} \exp\left[\frac{-1}{T} U(e_k, y_s)\right], \quad \forall e_k \in \Gamma$ |
| 2b | $x_s \leftarrow$ according to $P(x_s \mid y_s)$, randomly select $e_k \in \Gamma$ |
| 3 | $T \leftarrow T \times$ cooling rate |
| 5 | Repeat lines 2 and 3 until $T \leq T_{\min}$ |
| 1 | Initialize $x$ |
| 2 | For each site $s \in S$ |
| 3 | $x_s = \arg\min_{e_k \in \Gamma} U(e_k, y_s)$ |
| 4 | Repeat lines 2 and 3 until $x$ stabilizes |

areas and/or areas with occlusions. Typically, over these areas several vectors $x_s$ can have a minimum energy, although only one is valid. This is the well-known aperture problem.[19] To guarantee the uniqueness of a consistent solution, some approaches have been proposed.[16] Among these approaches, many opt for a regularization term (or smoothness constraints) whose essential role is to rightly constrain the ill-posed nature of this inverse problem. These constraints typically encourage neighboring vectors to point in the same direction with the same magnitude. In the context of the MAP, these constraints can be expressed as a prior energy function such as the Potts model of Eq. (6). However, since the number of labels can be large [here $(2d_{\max}+1)^2$], we empirically observed that a smoother function is better suited. In fact, the following linear function was implemented

$$V_{\eta_s}(x_s) = \sum_{c \in \eta_s} [|x_s(0) - x_r(0)| + |x_s(1) - x_r(1)|], \tag{9}$$

where $c$ is a binary clique linking site $s$ to site $r$. Notice that other smoothing functions are available.[19] The global energy function $U(x,y)$ to be minimized is obtained by combining Eqs. (8) and (9) as follows:

$$U(x,y) = \sum_{s \in S} \left\{ \underbrace{\|y_s(t) - y_{s+x_s}(t+1)\|}_{W_s(x_s,y_s)} + \beta_{\mathrm{ME}} \underbrace{\sum_{c \in \eta_s} [|x_s(0) - x_r(0)| + |x_s(1) - x_r(1)|]}_{V_{\eta_s}(x_s)} \right\}, \tag{10}$$

where $\beta_{\mathrm{ME}}$ is a constant. Note that Konrad and Dubois[8] proposed a similar solution involving a line process to help preserve edges.

**Table 2** *K*-means algorithm.

| | |
|---|---|
| 1 | $\mu_k \leftarrow$ random initialization $\forall \mu_k \in \Phi_\mu$ |
| 2 | For each site $s \in S$ |
| 2a | $x_s \leftarrow \arg\min_{e_k \in \Gamma} \|y_s - \mu_{e_k}\|^2$ |
| 3 | $\mu_k \leftarrow \frac{1}{N_k} \Sigma_{x_s = e_k} y_s \quad \forall \mu_k \in \Phi_\mu$ |
| 4 | Repeat lines 2 and 3 until each mean $\mu_k$ no longer moves |
| 5 | $\Sigma_k^{nm} \rightarrow \frac{1}{N_k} \Sigma_{x_s = e_k} (y_s^n - \mu_k^n)(y_s^m - \mu_k^m), \quad \forall \Sigma_k \in \Phi_\Sigma$ |

### 3.3 *Stereovision*

The goal of stereovision is to estimate the relative depth of 3-D objects from two (or more) images of a scene. For simplicity purposes, many stereovision methods use two images taken by cameras aligned on a linear path with parallel optical axis (this setup is explained in detail in Scharstein and Szelisky's review paper).[9] Stereovision algorithms often make some assumptions on the true nature of the scene. One common assumption (which is similar to motion estimation the lightness consistency assumption) states that every point visible in one image is also visible (with the same color) in the second image. Based on that assumption, the goal of a stereovision algorithm is to estimate the distance between each site $s$—with coordinate $(i,j)$—in one image to its corresponding site $t$—with coordinate $(i+d_s,j)$—in the second image. Such distance is called disparity and is proportional to the inverse depth of the object projected on site $s$. In this contribution, $d_s \in [0, D_{\mathrm{MAX}}]$. This gives rise to a matching cost function that measures how good a disparity $d_s$ is for a given reference image $y_{\mathrm{ref}}$. This is expressed mathematically by

$$C(s,d,y) = \|y_{\mathrm{ref}}(i,j) - y_{\mathrm{mat}}(i+d_s,j)\|, \tag{11}$$

where $y = \{y_{\mathrm{mat}}, y_{\mathrm{ref}}\}$, and $y_{\mathrm{mat}}$ is the second image familiarly called the matching image. Notice that the function $\|.\|$ can be replaced by a robust function[9] if necessary. In the context of the MAP, $C(.)$ stands for the likelihood energy func-

**Table 3** ICE algorithm.

| | |
|---|---|
| 1 | $\Phi \leftarrow K$-means |
| 2 | For each site $s \in S$ |
| 2a | $P(e_k \mid y_s) = \frac{1}{Z_s} \exp[W(e_k, y_s) + V_{\eta_s}], \quad \forall e_k \in \Gamma$ |
| 2b | $x_s \leftarrow$ according to $P(x_s \mid y_s)$, randomly select $e_k \in \Gamma$ |
| 3a | $\mu_k \leftarrow \frac{1}{N_k} \Sigma_{x_s = e_k} y_s \quad \forall \mu_k \in \Phi_\mu$ |
| 3b | $\Sigma_k^{nm} \leftarrow \frac{1}{N_k} \Sigma_{x_s = e_k} (y_s^n - \mu_k^n)(y_s^m - \mu_k^m), \quad \forall \Sigma_k \in \Phi_\Sigma$ |
| 5 | Repeat lines 2 and 3 until $\Phi$ stabilizes |

tion, and the disparity map $d$ is the label field to be estimated. Thus, to ensure uniformity with the notation of Sec. 2, the cost function of Eq. (11) can be referred to as $C(s,x,y)$.

To ensure spatial smoothness, two strategies have been traditionally proposed. The first one is to convolute $C(s,x,y)$ with a low-pass filter or a so-called aggregation filter $w$ (see Ref. 9 for details on aggregation). Although a prefiltering step slows down the segmentation process, it can significantly reduce the effect of noise and thus enhance result quality. The second strategy to ensure spatial smoothness is to take advantage of a prior energy term $V_{\eta_s}(x)$ of the form

$$V_{\eta_s}(x) = \sum_{c \in \eta_s} |x_s - x_r|, \tag{12}$$

where the absolute value could be replaced by another cost function if necessary. The global energy function $U(x,y)$ can thus be written as

$$U(x,y) = \sum_{s \in S} \left[ \underbrace{(w * C)(s,x,y)}_{W_s(x_s,y_s)} + \beta_s \underbrace{\sum_{c \in \eta_s} |x_s - x_r|}_{V_{\eta_s}(x_s)} \right], \tag{13}$$

where $\beta_s$ is a constant.

Minimizing the energy function of Eq. (13) can be time consuming, especially when the number of disparities $D_{\mathrm{MAX}}$ is large. To save on processing time, two simple strategies are conceivable. The first one is to minimize the likelihood function only and ignore the prior term: $\hat{x}_s = \arg\min_{x_s}(w*C)(s,x,y)$. In this way, the filter $w$ is assumed to be good enough to ensure spatial smoothness. This simple greedy strategy is called winner take all (WTA) and converges after only one iteration. Another way to reduce processing time is to precompute $C(s,x,y)$ in a 3-D table. Such a table–called the disparity space integration (DSI) table—contains $\mathcal{N} \times \mathcal{M} \times D_{\mathrm{MAX}}$ cost values. Once the DSI table has been computed, is can be filtered by $w$, after which the optimization process can be launched.

## 4 Probabilistic Segmentation

In this paper, both the color and the motion segmentation are based on a probabilistic criterion, which relates the observed data $\boldsymbol{y}_s$ to its label $x_s$ with a distribution $P(y_s|x_s)$. For the color segmentation, $\boldsymbol{y}_s$ takes a value between 0 and 255 for gray-scale images and between (0,0,0) and (255,255,255) for color images. As for motion segmentation, $\boldsymbol{y}_s$ is a 2-D vector represented by $\boldsymbol{y}_s = \boldsymbol{v}_s = (u_s, v_s)$, where $u_s$ and $v_s$ are real values. Notice that since $\boldsymbol{y}_s$ is an observation, the motion vector $\boldsymbol{v}_s \forall s \in S$ is assumed to have been previously estimated.

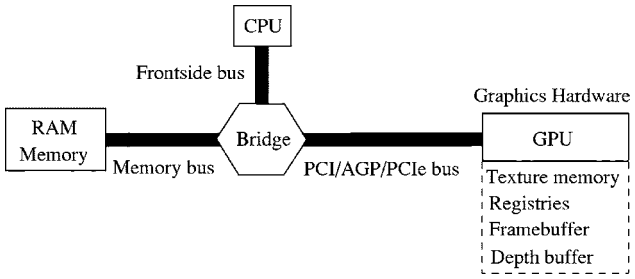


**Fig. 4** Diagram showing physical relation between the CPU, the main memory and the graphics hardware.

Because $\boldsymbol{y}_s$ is related to $\boldsymbol{x}_s$ by a probability distribution, the energy function $W_s(\boldsymbol{x}_s, \boldsymbol{y}_s)$ is designed according to that distribution, namely, $W_s(\boldsymbol{x}_s, \boldsymbol{y}_s) \propto -\ln P(\boldsymbol{y}_s|\boldsymbol{x}_s)$. A very popular function used to model $P(\boldsymbol{y}_s|\boldsymbol{x}_s)$ is the multidimensional Gaussian distribution

$$P(\boldsymbol{y}_s|\boldsymbol{x}_s) = \frac{1}{[(2\pi)^d |\Sigma_{x_s}|]^{1/2}} \exp\left[ -\frac{1}{2} (\boldsymbol{y}_s - \mu_{\boldsymbol{x}_s}) \Sigma_{\boldsymbol{x}_s}^{-1} (\boldsymbol{y}_s - \mu_{\boldsymbol{x}_s})^T \right], \tag{14}$$

where $d$ is the dimensionality of $\boldsymbol{y}_s$ ($d=1$ or 3 for grayscale/color segmentation and $d=2$ for motion segmentation) and $(\mu_{\boldsymbol{x}_s}, \Sigma_{\boldsymbol{x}_s})$ are the mean and variance-covariance matrices of class $\boldsymbol{x}_s$. Using a classical Potts function as prior model, the global energy function of Eq. (4) can be written as

$$\sum_{s \in S} \left\{ \frac{1}{2} \left[ \ln |\Sigma_{\boldsymbol{x}_s}| + (\boldsymbol{y}_s - \mu_{\boldsymbol{x}_s}) \Sigma_{\boldsymbol{x}_s}^{-1} (\boldsymbol{y}_s - \mu_{\boldsymbol{x}_s})^{\mathrm{T}} \right] + \beta_p V_{\eta_s}(\boldsymbol{x}_s) \right\}. \tag{15}$$

In the case of unsupervised segmentation, the Gaussian parameters $\Phi = \{(\mu_k, \Sigma_k) \,|\, 1 \le k \le N\}$ must be estimated conjointly with $x$ or preliminary to the segmentation step. To

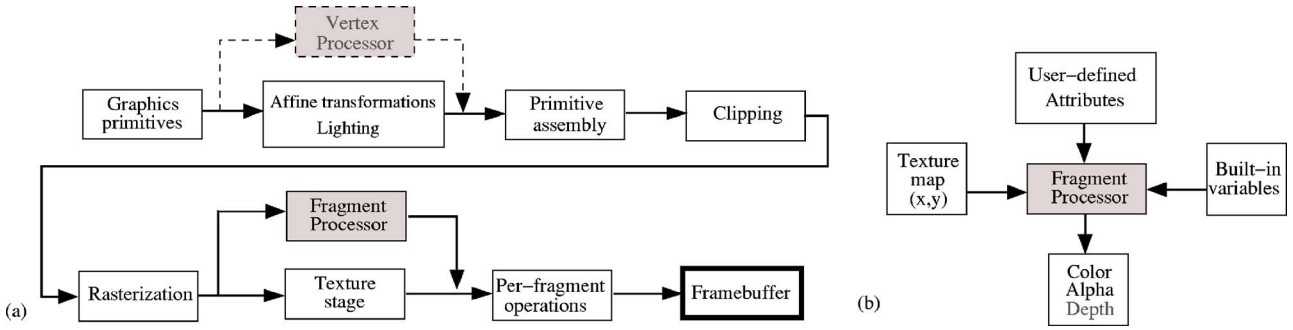


**Fig. 5** (a) Logical diagram of a typical processing pipeline with programmable vertex and fragment processor and (b) fragment processor inputs and outputs.

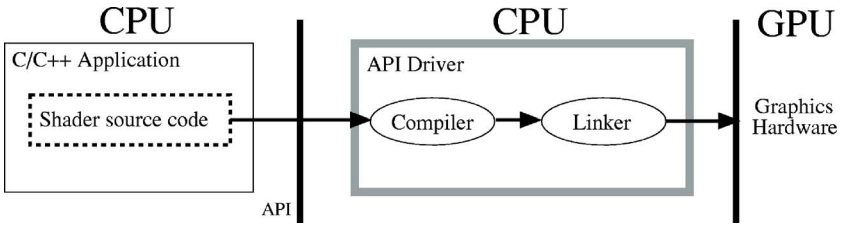


**Fig. 6** Execution model of a typical shader.

do so, many parameter estimation algorithms are available among which *K*-means[5] and ICE (Ref. 6) are commonly used. For further details on probabilistic Markovian segmentation, consider the following books.[4,20]

## 5 Optimization Procedures

Since Eq. (4) has no analytical solution, $\hat{x}$ must be estimated with an optimization procedure. The first optimization procedure we have implemented is the SA, which is a stochastic relaxation algorithm based on a Gibbs sampler. The concept of SA is based on the manner in which some material recrystallizes when slowly cooled down after being heated at a high temperature. The final state (called the frozen ground state) is reached when temperature gets down to zero. Similarly, SA searches for the global minima by cooling down a temperature factor[21] $T$ from an initial temperature $T_{\mathrm{MAX}}$ down to zero.

The major advantage with SA is its ability to always reach the global minima with the appropriate decreasing cooling temperature schedule. This is made possible because SA authorizes energy increases to escape form local minima. To do so, SA stochastically samples the system probability distribution and randomly generates new configurations. In this paper, the system probability is made of the global energy function [here $U(x,y)$] and a temperature factor $T$. This probability function is similar to Boltzmann's probability function,[21] which can be written as

$$P(x,y) = \frac{1}{\lambda} \exp\left[ -\frac{U(x,y)}{T} \right], \tag{16}$$

where $\lambda$ is a normalization factor. The SA algorithm is presented in the upper section of Table 1.

The main limitation with SA is the number of iterations it requires to reach the frozen ground state. This makes SA unsuitable for many applications for which time is a decisive factor. This explains the effort of certain researchers to find faster optimization procedures. One such optimization procedure is Besag's ICM algorithm.[3] Starting with an initial configuration $x^{[0]}$, ICM iteratively minimizes $U(x,y)$ in a deterministic manner by selecting, for every site $s \in S$, the label $e_k \in \Gamma$ that minimizes the local energy function at that point. Since ICM is not stochastic, it cannot exit from local minima and, thus, requires $x$ to be initialized near the global minima. In practice, however, this limitation is rarely a problem since ICM generates fairly good results, always at a fraction of the time needed by SA (Fig. 3 illustrates the difference between ICM and SA). The ICM algorithm is presented in the lower section of Table 1.

## 6 Parameter Estimation

The two parameter estimation algorithms we have implemented for this paper are *K*-means[5] and ICE.[6] *K*-means is an iterative clustering method that assumes input data $\{y_s\}$ is distributed within $K$ spherical clusters of equal volume (see Table 2). At each iteration, every site $s$ is assigned to the nearest cluster after which the center of mass of every cluster is recomputed. The resulting *K*-means clusters minimize the sum-of-square error function: $\Sigma_{k=1}^{N} \Sigma_{x_s=e_k} \| y_s - \mu_k \|^2$.[3] The variance-covariance of each cluster is estimated once the algorithm has converged.

The *K*-means algorithm has two well-known limitations. First, its assumption that all clusters are spherical with equal volume is simplistic an often unsuited to some observation fields. Second, because *K*-means is a deterministic algorithm, it is sensitive to noise and is likely to converge toward local minima. Consequently, some authors suggest refining $\Phi$ with a more realistic model, less sensitive to noise, and local minima such as the stochastic (and Markovian) ICE estimation algorithm. Further details on this algorithm are presented in Ref. 6, while Table 3 presents a version of ICE adapted to this paper.

## 7 Graphics Hardware Architecture

As mentioned previously, graphics hardware is highly optimized to solve traditional computer graphics problems. Today, graphics hardware is generally embedded on a graphics card, which can receive/send data from/to the CPU and the main memory via the system bus, be it PCI, AGP, or PCIe (see Fig. 4). To our knowledge, most graphics hardware are designed to fit the graphics processing pipeline shown in Fig. 5.[22,23] This pipeline is made of various stages which sequentially transforms images and geometric input data into an output image stored in a section of

**Table 4** Algorithm of a C/C++ application compiling, linking and loading a shader. Notice that lines 2 to 5 are done by functions provided by the API driver.

| | |
|---|---|
| 1 | Buf ← Copy the shader source code in a 1-D buffer |
| 2 | Give Buf to the driver |
| 3 | Compile the shader code |
| 4 | Link the compiled shader code |
| 5 | Send the linked code to the graphics hardware |

**Table 5** High-level representation of an ICM hardware segmentation program.

| | |
|---|---|
| 1 | Copy the input images $y$ into texture memory |
| 2 | Compile, link, and load ICM shader on the GPU |
| 3 | Specify shader parameters ($\beta_{MD}$, $\beta_{ME}$, or $\beta_s$, 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 line 4 and 5 until convergence |
| 7 | Copy the framebuffer into a C/C++ array if needed |
| 1 | $\hat{x}_s \leftarrow \arg\min_{x_s} U(x_s, y_s)$ |
| 2 | $\text{framebuffer}_s \leftarrow \hat{x}_s$ |

The upper section (lines 1 to 7) is a C/C++ CPU program used to load the shader, render the scene and manage textures. The second program (lines 1 and 2) is the ICM fragment shader launched on every fragment (pixel) when the scene is rendered (line 4). Notice that images $x$ and $y$ are contained in texture memory.

graphics memory called the framebuffer. Part of the framebuffer (the front buffer) is meant to be visible on the display device.

Until recently, graphics pipelines have presented a flexible but static interface to application programmers. Although many parameters on each processing stage could be tweaked to adjust the processing, the fundamental graphics operations could not be changed. To answer this limitation, hardware manufacturers have made the vertex and fragment processing stages programmable. These two stages can now be programmed using C-like languages to process vertex and fragments with user-defined operations. Note that a fragment is a per-pixel data structure created at the rasterization stage and containing data such as color, texture coordinates, depth, and so on. Each fragment is used to update a unique location in the framebuffer. For example, a scene made of a 5-pixel-long horizontal line will generate five fragments, whereas a $100\times100$ plan perpendicular to an orthographic camera will generate 10,000 fragments.

Because of the very nature of graphics applications, the GPU has been designed as a streaming processor, that is, a processor with inherent parallel processing abilities. With such processor, the vertices and the fragments are processed in parallel, thus providing all graphics applications a significant acceleration factor.

### 7.1 *Fragment Programs*

With a GPU, general-purpose applications going beyond traditional computer graphics can now be implemented on graphics hardware[10] to take advantage of its parallel abilities. This is especially true for various image processing applications. These applications are generally executed over the fragment processor, mostly because it is the only part of the graphics pipeline that has access to both input memory (texture memory) and output memory (the framebuffer). It has also traditionally been the most powerful stage of the GPU.

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. As shown in Fig. 5(b), such a program typically has access to three kinds of input values.[22,24] First, a fragment shader has a read-only access to the texture memory. In this contribution, the texture memory contains the observation field $y$ and the label field $x^{[t-1]}$, estimated during the previous optimization iteration. Second, fragment shaders have access to built-in variables containing general graphics informations such as the model view matrix, the light sources, the fog, and the material to name a few. The only built-in variable used by our shaders is the one containing the fragment coordinates $(i,j)$. Third, fragment shaders have access to user-defined variables containing all kinds of application-specific data such as weighting factors, class parameters, and window size. These data are typically stored in arrays, vectors, and integer or floating point variables.

A fragment shader can return color, alpha, and depth values. The first two values are stored in the framebuffer where as the last one is copied in the depth buffer. In this contribution, only the values copied in the framebuffer are taken into account.

In short, a fragment shader is a program executed on the fragment processor that processes in parallel every fragment (see pixel) returned by the rasterization stage. This shader has access to user-defined parameters, has a read-only access to texture and a write-only access to the framebuffer, and cannot exchange information with the neighboring fragments. Note that while the GPU can be a very powerful processing tool, sending and receiving data from/to the CPU across the system bus introduces significant latency. As such, data traffic between the CPU application and the GPU should be kept at a strict minimum.

### 7.2 *Loading and Executing a Fragment Shader*

As mentioned before, a fragment processor is made to load and execute in parallel a program called the fragment shader. As shown in Fig. 6, the shader source code is first located in a C or C++ application running on the CPU. Typically, this source code is listed in a 1-D "unsigned char" array. While the C/C++ application is running, the shader source code is compiled and linked by the application program interface's (API)s driver. Once the linking has finished, the linker returns a program object that is loaded on the graphics hardware for execution. The shader is executed when one or more graphics primitives are rendered. This procedure is illustrated in Table 4. Notice that most common APIs such as OpenGL and DirectX provide easy to use high-level driver functions.[22,24]

### 7.3 *Markovian Segmentation on the GPU*

Although fragment shaders (as well as vertex shaders) can be written in a C-like procedural languages,[22,24] they have some specificities as compared to ordinary C/C++ programs. The most important ones are

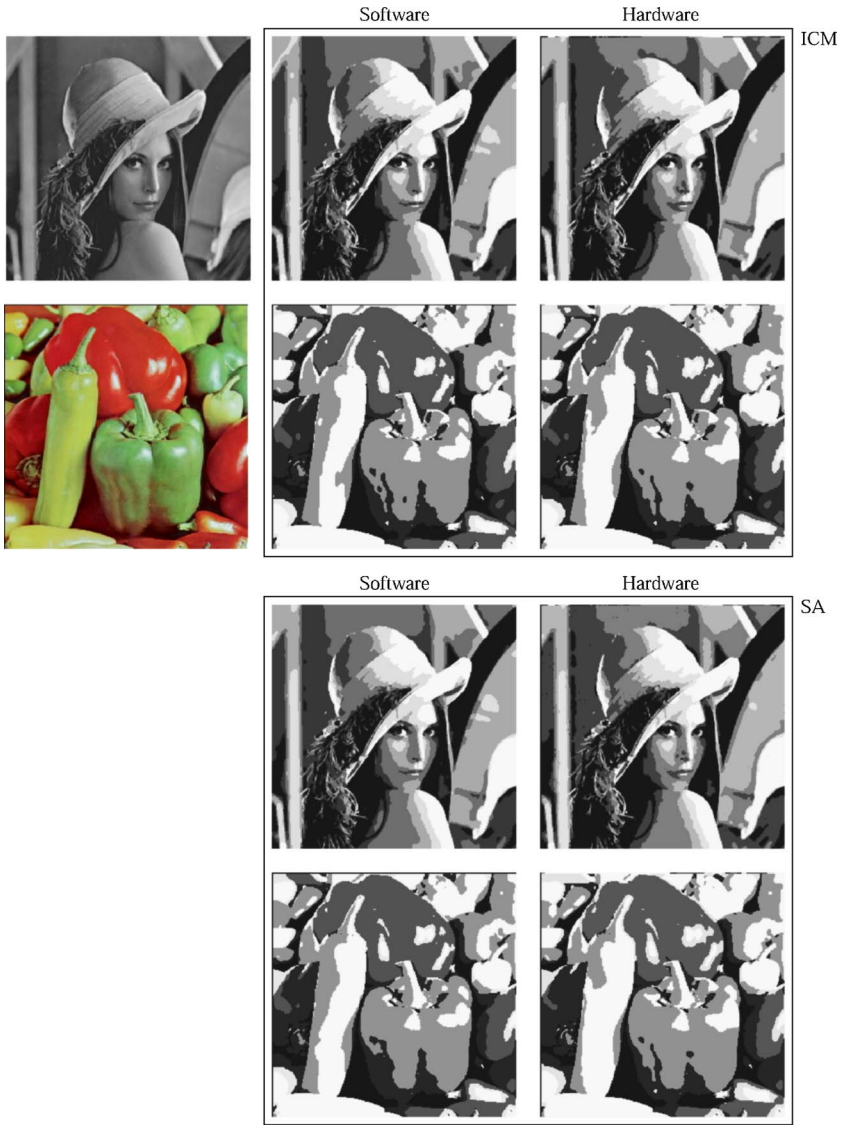


**Fig. 7** Gray-scale and color image segmented with the software (left column) and hardware (right column) version of ICM (top four images) and SA (four lowest images). In every cases, the Gaussian parameters have been estimated with *K*-means (color online only).

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 framebuffer and the depth buffer.
3. The only data a fragment shader can read is contained in the texture memory, in built-in variables and in user-defined variables. As such, it cannot read the content of the framebuffer of the depth buffer.
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. (4) can be tricky. In fact, three main problems must be taken care of. First, when performing a Markovian segmentation, the fragment operations should obviously be performed on every pixel of the input scene. As such, a perfect 1:1 mapping from the input pixels to the output buffer pixels is necessary. This is achieved by rendering a screen-aligned rectangle covering a window with exactly the same size than the input image. To alleviate all distortion due to perspective, we use an orthographic camera. In this way, the rasterization stage generates $\mathcal{N} \times \mathcal{M}$ fragments, one for every pixel of the input images. This is illustrated by the ICM algorithm presented in Table 5. In this example, the fragment shader is executed when the rectangle primitive is rendered (line 4). At that point, the fragment shader minimizes in parallel the energy function $U(x_s, y_s)$ on each site $s \in S$.

The second problem comes from the fourth limitation. Since GPUs provide no general-purpose memory, one might wonder how the prior energy function $V_{\eta_s}$ can be implemented on a fragment program since it depends on the neighboring labels $x_t$ contained in the (write-only) framebuffer. As shown in Table 5, after rendering the scene,

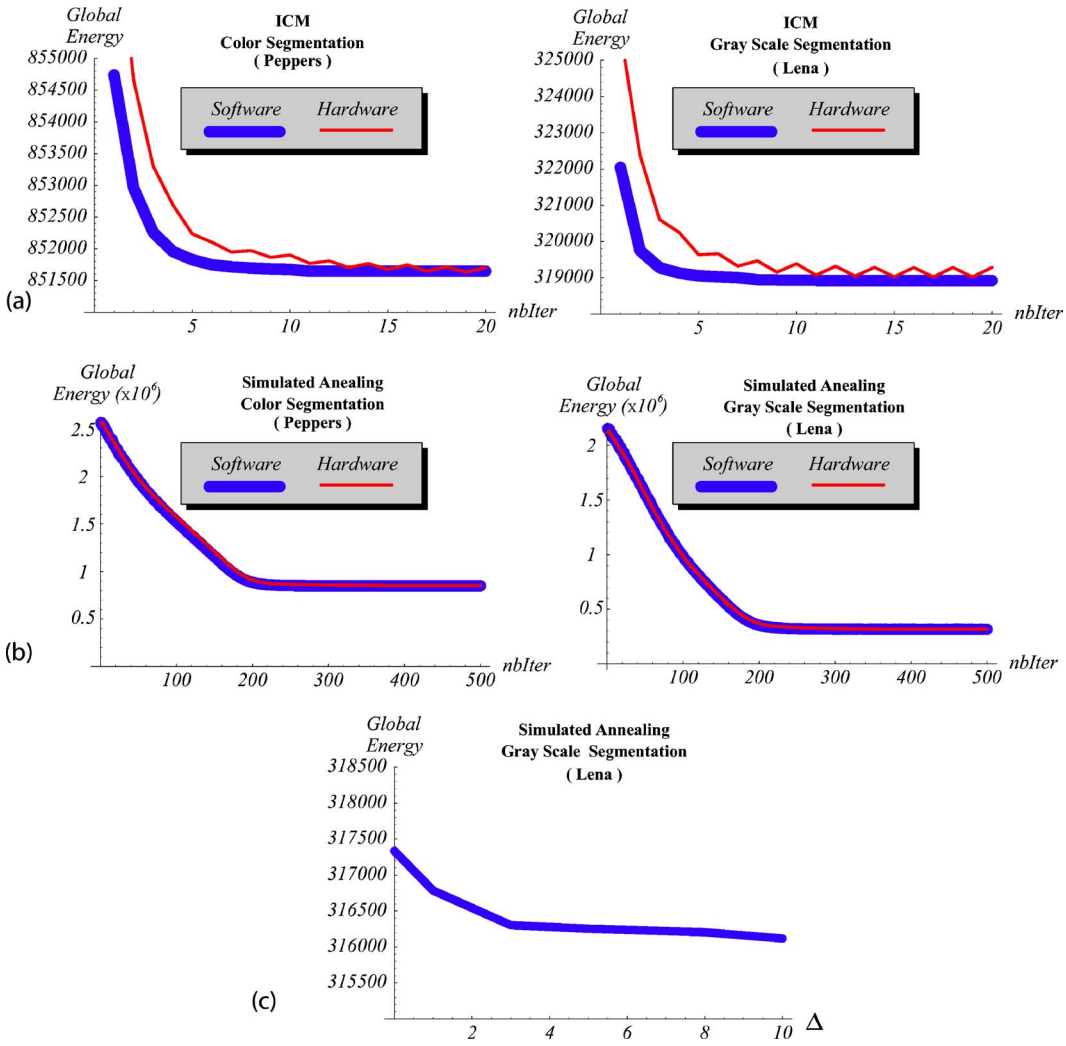


**Fig. 8** Plot of Eq. (15) when segmenting a color or a gray-scale image (see Fig. 7) with (a) ICM or with (b) SA and (c) the influence of variable Δ on the global energy after 500 SA iterations.

the CPU program copies the framebuffer into texture memory (line 5). In this way, the texture memory (which can be read by fragment shaders) contains not only the input images, but also the label field $x^{[t-1]}$ computed during the previous iteration. Thus, $V_{\eta_s}$ is computed with labels iteratively updated and not sequentially updated as it is generally the case. Such strategy was already proposed by Besag[3] and successfully tested by other authors.[14] This iterative updating scheme corresponds to the Jacobi-type version of the initial Gauss-Seidel ICM procedure. However, as mentioned by Besag,[3] such parallel updating scheme can induce small oscillations. In fact, it turns out that for some examples, energy oscillations can occur after 10 or 20 iterations [see Fig. 8(a)]. These oscillations are caused by some labels (rarely more than a few dozen, sparsely distributed in the label field) that endlessly switched between two classes. Although the Gauss-Seidel and the Jacobi versions of ICM do not strictly converge toward the same minima, the results they return are similar both visually and energy-wise (see Figs. 7 and 8). Notice that the iterative nature of ICM is reproduced with multiple renderings of the rectangle (lines 4 through 6 of Table 5), the fragment shader containing no iteration loop.

The last problem with shaders comes from their inability to generate random numbers such as needed by the stochastic SA and ICE algorithms. We thus implemented a workaround that goes as follows. First, in the C/C++ application, an image with random values was created. This random image was then copied in texture memory where the shader can access it. In this way, a stochastic shader processing a site $(i,j)$ has access to an array of random numbers. This procedure is illustrated by the algorithms of Tables 6–8. Please remark that the shaders are fed with a vector $\delta=(\delta_I, \delta_J)$ whose value is between $-\Delta$ and $\Delta$. This is done to ensure that the same random value is not reused at each iteration.

Although this workaround is not as efficient as a good random number generator, the results obtained with our hardware programs are very close to the ones obtain with the software programs. To illustrate the effectiveness of our

**Table 6** High-level representation of an SA hardware segmentation program.

| | |
|---|---|
| 1 | Rnd $\leftarrow$ Create and $\mathcal{N}\times\mathcal{M}$ image random values |
| 2 | Copy the images $y$ and Rnd into texture memory |
| 3 | Compile, link and load the SA shader on the GPU |
| 4 | $T \leftarrow T_{\max}$ |
| 5 | $\delta_I$, $\delta_J \leftarrow$ random integers between $-\Delta$ and $\Delta$ |
| 6 | Specify shader parameters ($\delta_I$, $\delta_J$, $T$, and $\beta_s$, for example) |
| 7 | Render a rectangle covering a window of size $\mathcal{N}\times\mathcal{M}$ |
| 8 | Copy the framebuffer into texture memory |
| 9 | $T \leftarrow T^*$ cooling rate |
| 10 | Repeat lines 5 to 9 until $T < T_{\min}$ |
| 11 | Copy the framebuffer into a C/C++ array if needed |
| 1 | $P(x_s = e_k \mid y_s) \leftarrow \frac{1}{\lambda}\exp\left[\frac{-1}{T}U(x_s = e_k, y_s)\right],\ \forall e_k \in \Gamma$ |
| 2 | $r \leftarrow \text{Rnd}_{s+\delta}$ |
| 3 | According to $r$ and $P(x_s \mid y_s)$, randomly select $e_k \in \Gamma$ |
| 4 | $\text{Framebuffer}_s \leftarrow \hat{x}_s$ |

The upper section (lines 1 to 11) is a C/C++ CPU program used to load the shader, render the scene and manage textures. The second program (lines 1 to 4) is the fragment shader launched on every fragment (pixel) when the scene is rendered (line 7). Since shaders are not equipped with a random number generator, random value are stored in the texture memory (Rnd). Notice that the result were obtained with $\Delta = 5$.

**Table 7** High-level representation of a *K*-means program.

| | |
|---|---|
| 1 | Copy the input images $y$ 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 ($N$ and $\Phi_\mu$) |
| 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 | $\mu_k \leftarrow \frac{1}{N_k}\Sigma_{E_s = e_k} y_s,\ \forall \mu_k \in \Phi_\mu$ |
| 8 | Repeat lines 4 to 7 until convergence |
| 9 | $\Sigma_k^{nm} \leftarrow \Sigma_{E_s = e_k}(y_s^n - \mu_k^n)(y_s^m - \mu_k^m),\quad \forall \Sigma_k \in \Phi_\Sigma$ |
| 10 | Copy the framebuffer into a C/C++ array if needed |
| 1 | $x_s \leftarrow \arg\min_{e_k} \lVert y_s - \mu_{e_k} \rVert^2$ |
| 2 | $\text{framebuffer}_s \leftarrow x_s$ |

The upper section (lines 1 to 10) is the C/C++ CPU program used to load the shader, render the scene, and compute the Gaussian parameters $\Phi$. The second program (lines 1 and 2) is the fragment shader launched on fragment (pixel) when the scene is rendered (line 5).

workaround, we have segmented a gray-scale and a color image with a good software random number generator and with our hardware method. As can be seen in Fig. 7 and Fig. 8(b), our hardware method generates results very close to the ones obtained with the traditional software method, both visually and energywise. Note that this random value problem will surely be solved when hardware graphics companies will include a pseudorandom number generator in their shading language library. But in the mean time, our workaround can be used will little loss of precision.

### 7.4 *Energy-Based Segmentation on GPU*

With the techniques described in the previous section, performing motion detection, motion estimation, and stereovision on a GPU is fairly straightforward. Since the shading languages (NVIDIA's Cg language in our case) have a syntax similar to C, the software programs could be almost directly reused in the shader. The implementation of the three fragment programs is conceptually very similar, since they all minimize an energy function composed of a likelihood term and a prior term.

There is one exception, however, that occurs when a prefiltering step is required by the stereovision application. To handle this situation, the cost function $C(s,x,y)$ is first precomputed and stored in a 3-D DSI table located in texture memory. Then, the DSI table is filtered by $w$, after which the optimization procedure (be it ICM, SA, or WTA) is launched. In the context of a typical GPU, these operations can be done with one fragment shader made of three functions: one to compute the matching cost $C(s,x,y)$, one to filter $C(s,x,y)$ with $w$, and one to minimize the global energy function $U(x,y)$. The first shader function computes, for each site $s \in S$, the cost value $C(s,x,y)$ associated with each disparity $d_s \in [0, D_{\max}]$. The cost values are then copied in texture memory, which stands for the DSI table. Immediately after, the second shader function filters the DSI table $[(w * C)(s,x,y)]$ and copies the result in texture memory. At this point, the likelihood values $W_s(x_s, y_s)$, $\forall s \in S,\ \forall x_s \in \Gamma$ are stored in texture memory. The third shader function is finally used to minimize the energy function $U(x_s, y_s)$ with ICM, SA, or WTA. The overall stereovision *hardware* algorithm is presented in Table 9.

### 7.5 *Probabilistic Segmentation on GPU*

Optimizing Eq. (15) with SA or ICM is done the same way as for energy-based applications. The hardware algorithms of Table 5 and 7 can be directly reused, with the difference that shader parameters include the Gaussian mixture parameters $\Phi$. The delicate aspect of probabilistic segmentation concerns more the parameter estimation procedures *K*-means and ICE, which are not perfectly suited for theGPU architecture. While the first step of these algo-

**Table 8** High-level representation of an ICE program.

| | |
|---|---|
| 1 | Rnd ← Create an $\mathcal{N}\times\mathcal{M}$ image with random values |
| 2 | Copy the input images $y$ and Rnd into texture memory |
| 3 | Compile, link, and load the ICE shader on the GPU |
| 4 | $\Phi\leftarrow$ Init Gaussian parameters |
| 5 | $\delta_I$, $\delta_J\leftarrow$ random integers between $-\Delta$ and $\Delta$ |
| 6 | Specify shader parameters ($\delta_I$, $\delta_J$, $N$, and $\Phi$) |
| 7 | Render a rectangle covering a window of size $\mathcal{N}\times\mathcal{M}$ |
| 8 | $E\leftarrow$ Copy the framebuffer into a C/C++ array |
| 9 | Copy the framebuffer into texture memory |
| 10 | $\mu_k\leftarrow\frac{1}{N_k}\Sigma_{E_s=e_k}y_s,\ \forall\mu_k\in\Phi_\mu$ |
| 11 | $\Sigma_k^{nm}\leftarrow\Sigma_{E_s=e_k}(y_s^n-\mu_k^n)(y_s^m-\mu_k^m),\quad\forall\Sigma_k\in\Phi_\Sigma$ |
| 12 | Repeat lines 5 to 11 until convergence |
| 13 | Copy the framebuffer into a C/C++ array if needed |
| 1 | $P(e_k\|y_s)=\frac{1}{Z_s}\exp[W(e_k,y_s)+V_{\eta_s}],\quad\forall e_k\in\Gamma$ |
| 2 | $r\leftarrow\mathrm{Rnd}_{s+\delta}$ |
| 3 | $x_s\leftarrow$ according to $r$ and $P(x_s\|y_s)$, randomly select $e_k\in\Gamma$ |

The first section (lines 1 to 13) is a C/C++ CPU program used to load the shader, render the scene and compute the Gaussian parameters $\Phi$. The second program (lines 1 to 3) is the fragment shader launched on every fragment (pixel) when the scene is rendered (line 7).

**Table 9** Stereovision program using three shader functions to compute and filter the cost function and segment the scene.

| | |
|---|---|
| 1 | Copy input images $y_{\mathrm{mat}}$, $y_{\mathrm{ref}}$ in texture memory |
| 2 | Compile, link, and load the stereovision shader on the GPU |
| | //**** *Compute the DSI table* **** // |
| 3 | Tell the shader to use the DSI function |
| 4 | For $d$=0 to $D_{\max}$ |
| 5 | Specify shader parameters ($d$) |
| 6 | Render a rectangle covering a window of size $\mathcal{N}\times\mathcal{M}$ |
| 7 | Copy the framebuffer into texture memory |
| 8 | $d\leftarrow d$+4 |
| | //**** *Apply filter w* **** // |
| 9 | Tell the shader to use the filter function |
| 10 | For $d$=0 to $D_{\max}$ |
| 11 | Specify shader parameters ($d$ and filter parameters) |
| 12 | Render a rectangle covering a window of size $\mathcal{N}\times\mathcal{M}$ |
| 13 | Copy the framebuffer into texture memory |
| 14 | $d\leftarrow d$+4 |
| | //**** *Launch ICM to minimize* $U(x,y)$ **** // |
| 15 | Tell the shader to use the ICM function |
| 16 | Specify shader parameters ($D_{\max},\beta_s$) |
| 17 | Render a rectangle covering a window of size $\mathcal{N}\times\mathcal{M}$ |
| 18 | Copy the framebuffer into texture memory |
| 19 | Repeat line 17 and 18 until covergence |
| 20 | Copy the framebuffer into a C/C++ array if needed |

From lines 3 to 8, matching cost function $C(s,x,y)$ is computed and stored in texture memory, which stands as a DSI table. Then, from lines 9 to 14, the cost function is filtered ($w*C$) before the label field is estimated with ICM.

rithms (assigning the best cluster for each image pixel for each site $s\in S$, line 2 of Tables 2 and 3) is perfectly implementable in parallel, the second step (Gaussian parameters computation, line 3 of Tables 2 and 3) is not. As such, we have to take an hybrid approach: execute line 2 on the GPU (parallel processing) and line 3 on the CPU (sequential processing).

At first, the input images $y$ are put in texture memory so it is accessible by the fragment shaders. Each site $s\in S$ are then assigned the best class $e_k$ by the fragment shader, before the Gaussian parameters of every class are reestimated. Because this last operation is global and thus cannot be parallelized, the framebuffer containing the class of each pixel is read back to the CPU memory, where the computation takes place. Once the parameters are reestimated, they are passed back to the GPU, after which a new iteration can begin. The implementation of $K$-means and ICE is illustrated in Tables 7 and 8.

## 8 Experimental Results

Results compare software and hardware implementations of the energy-based and probabilistic applications we have discussed so far. The goal being to show how fast a segmentation program implemented on a GPU is compared to its implementation on a CPU. The software programs were implemented in C++ (C and C++ are by far the most utilized languages for hardware programming; empirical tests have shown that the two languages provides similar processing times) and the NVIDIA Cg language was used to implement the fragment programs. Because Cg's syntax is very much similar to C and C++, the C++ code of our

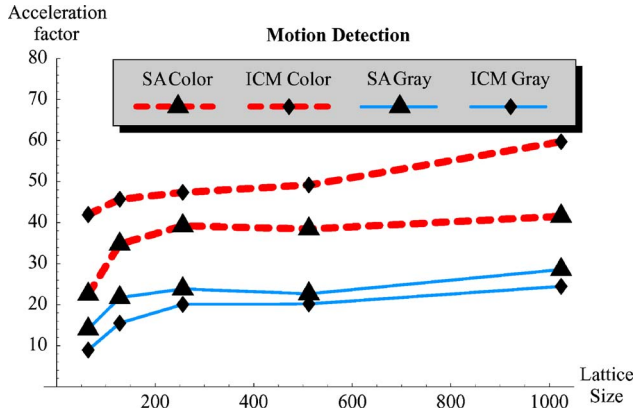


**Fig. 9** Acceleration factor for motion detection programs over square image sequences.

software applications were partly reused to implement the fragment shaders. As such, the differences between the software and hardware implementations were kept at a strict minimum and, thus, could be fairly compared.

Every results were made after varying some variables. In Figs. 9–13, the lattice size vary between $64\times64$ and $1024\times1024$ and the number of classes (or disparities) between 4 and 32, depending on the application. The number of iterations was set to 10 for ICM, $K$-means, and ICE and to 500 for SA. Thus, because the number of iterations is fixed for each algorithm, the processing times are independent of the image content.

Notice that for ICE and SA, $\Delta$ was set to 5. The reason for this choice is mostly technical. As shown in Fig. 8(c), using a value larger than 5 does not minimize the global energy much more. Also, because a value larger than 5 adds a latency to the algorithm, we conclude that 5 is a good compromise between speed and accuracy.

All results are expressed as an acceleration factor between the software and hardware programs. However, the results do not include the time required to load, compile, and link the shaders, which can vary between 0.05 and 5 s. Although this might seem prohibitive, this initialization step is done only once at the beginning of the application. In this way, when segmenting more than one scene (or segmenting a scene with a lattice size larger than $128\times128$), this initialization time soon becomes negligible as compared to the acceleration factor. This is especially true when SA is used as optimization procedure.

All programs were executed on a conventional home PC equipped with a AMD Athlon 2.5 GHz, with 2.0 G of RAM and a NVIDIA 6800 GT graphics card. The NVIDIA fp40 Cg profile was used in addition to the cgc compiler version 1.3.

### 8.1 *Energy-Based Segmentation*

Figures 9–11 contain the acceleration factor for the three energy-based applications. In Figs. 9 and 10 are the results for SA and ICM over gray-scale and color sequences. The way SA and ICM are initialized varies from one application to another. For motion detection, a label field obtained after a trivial thresholding operation is used to initialize the two optimizers. As for motion estimation, the label field is simply initialized to zero, whereas for stereovision, ICM and SA are fed with the disparity map generated by a simple WTA.

In every case, the hardware implementation is faster than its software counterpart by a factor between 10 and 60. Notice that the acceleration factor is more important for color sequences than for gray-scale sequences. This is explained by the fact that the likelihood energy function $W$ of the motion estimation and motion detection programs is more expensive to compute with RGB values than with gray-scale values. Thus, distributing this extra load on a fragment processor results in a more appreciable acceleration factor. In Fig. 10, $d_{\max}$ was set to 4 in the first graphic and the lattice size was set to $256\times256$ in the second graphic.

For stereovision (Fig. 11), we have tested our programs for the three tasks presented in Table 9, namely, the computation of the DSI table, the aggregation filtering, and the optimization procedure (SA, ICM, and WTA). The three optimization procedures were tested on scenes of various size and with different number of disparities. In the leftmost graphics, $D_{\max}$ was set to 16 and the lattice size was set to $256\times256$ in the other graphics. As we can see, the acceleration factor for ICM and SA is more important than the one for WTA. This can be explained by the fact that WTA is a trivial and efficient algorithm (it converges in

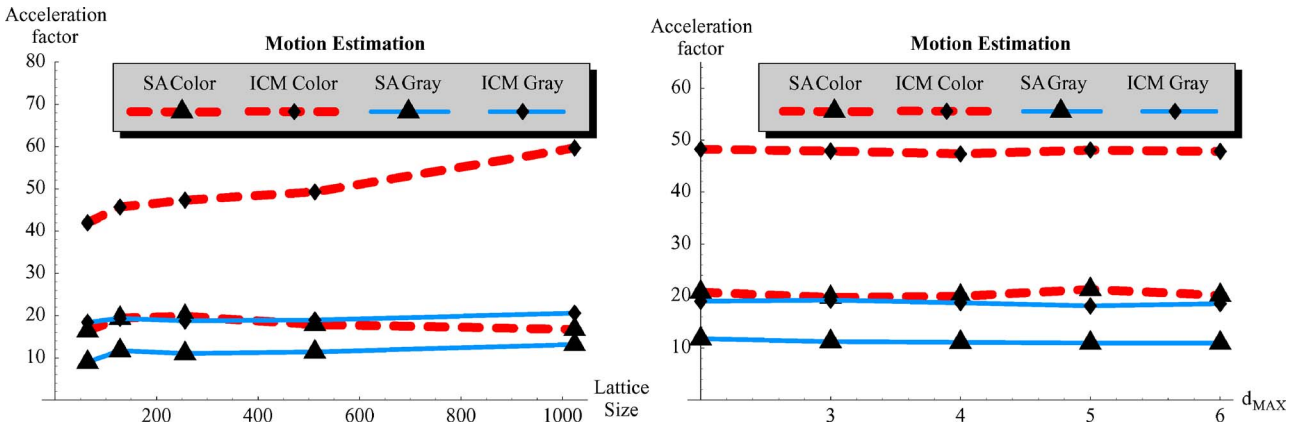


**Fig. 10** Acceleration factor for the motion estimation programs.

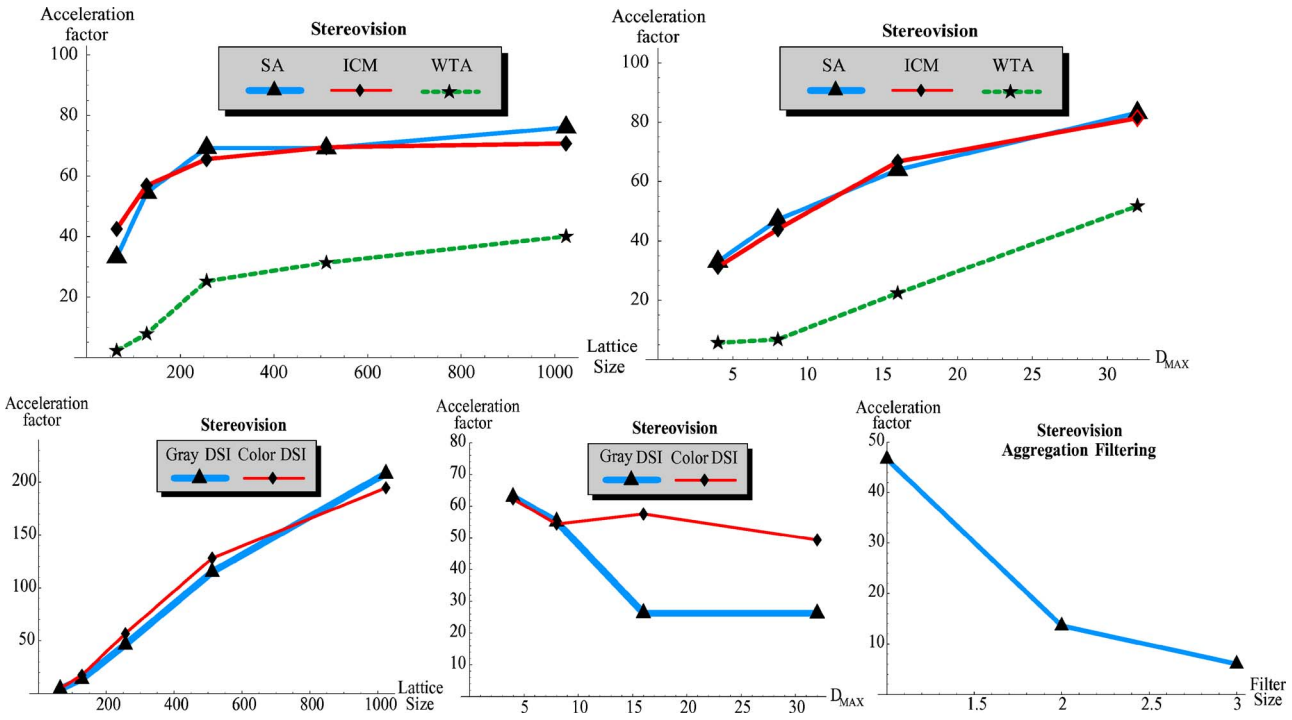


**Fig. 11** Acceleration factor for the stereovision programs.

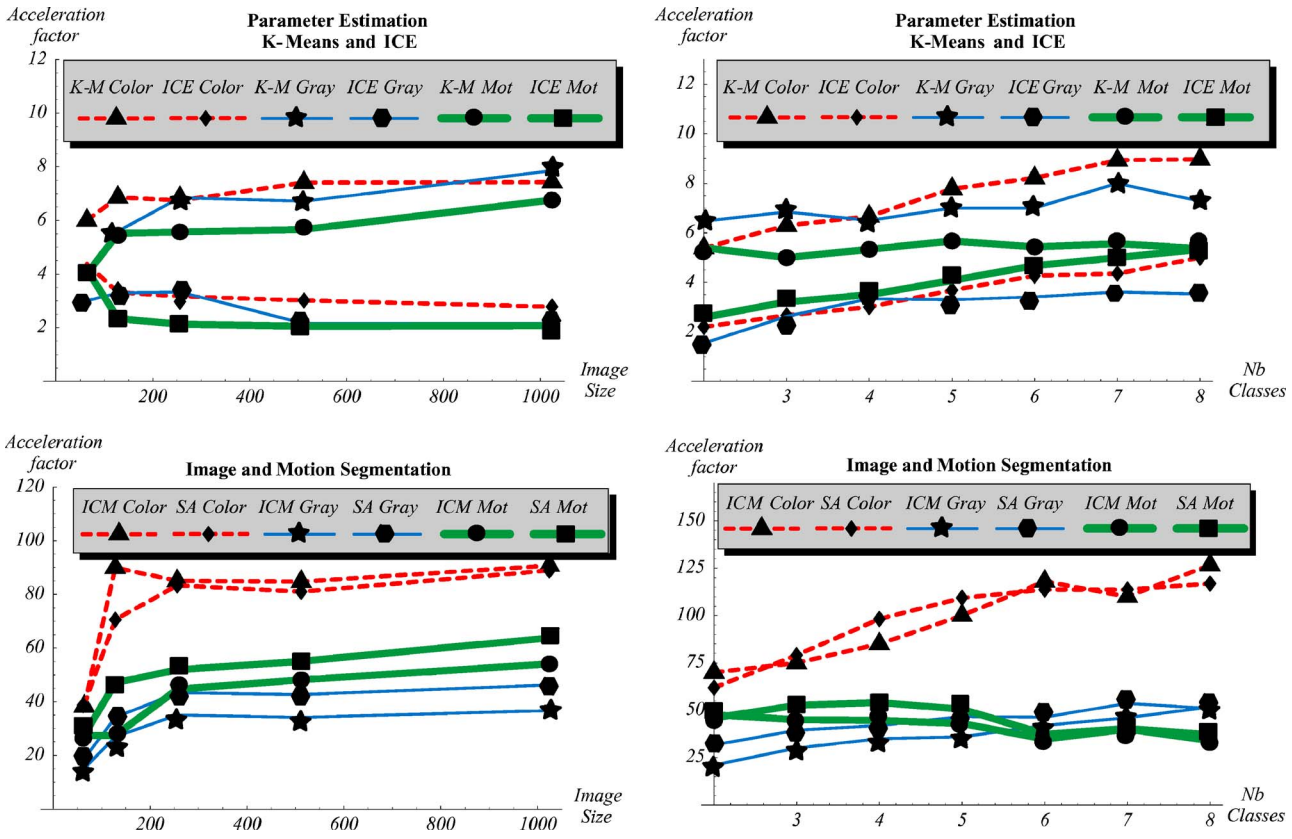


**Fig. 12** Acceleration factor for *K*-means, ICE, SA, and ICM obtained on gray-scale and or color images and on vector fields (motion segmentation).

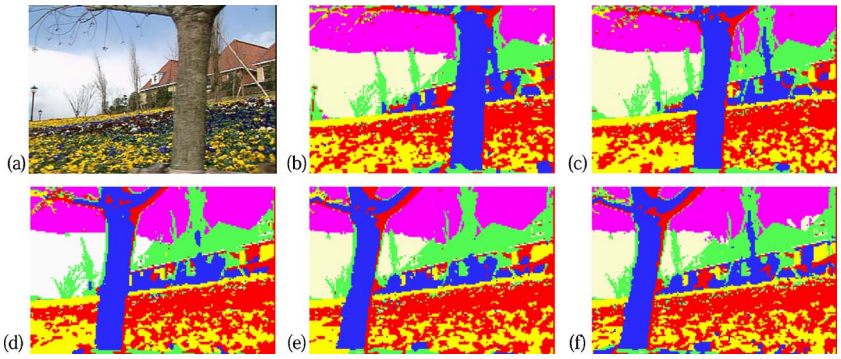


**Fig. 13** Image sequence of size 352×240 segmented in six classes. The Gaussian parameters estimated on the first frame (a) are used to segment the entire sequence (b) to (e). (f) Last frame segmented with newly estimated Gaussian parameters.

only one iteration) with a less impressive amount of computation to distribute on the GPU than for ICM and SA.

As for the task of computing the DSI table, we compared our hardware and software implementations over color and gray-scale input images. Again, since the likelihood cost function $C(s,x,y)$ is more expensive to compute with RGB values than with gray-scale values, and the acceleration factor for the color DSI is more important than the one for the gray-scale DSI.

### 8.2 *Statistical Segmentation*

The statistical applications presented in Secs. 5 and 6 were also implemented in C++ and in Cg. The performances of each implementation were evaluated by varying the number of segmentation classes and the size of the images to be segmented. The acceleration factor between the software and hardware version of the programs is presented in Fig. 12. In both cases, ICM and SA are initialized by the label field returned by ICE.

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). The reason for this is that *K*-means and ICE must exchange information (for the Gaussian parameter estimation) with the CPU, which is a major bottleneck for such hardware programs. Hence, this is why the parameter estimation programs seem less efficient than ICM and SA.

As we can also see, the speedup factor for *K*-means is larger than for ICE. This is explained by the fact that ICE has to estimate (and invert) the variance-covariance matrix at each iteration, which is not required for *K*-means. This extra load on the CPU makes ICE less efficient than *K*-means.

As is the case for most energy-based applications, the speedup factor for SA and ICM is more important in color images than in gray-scale images. Again, this is explained by the fact that the energy function of Eq. (15) is more expensive to compute for color images than for gray-scale 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 four classes at a rate of 76 frames/s with ICM, 1.4 frames/s with SA, 2.5 frames/s with ICE and 14 frames/s with *K*-means. Although *K*-means and ICE estimate parameters at an interactive rate, they can be seen as slow procedures, at least compared to ICM. Thus, to save on processing time, when segmenting an image sequence with frames having mostly the same color distribution, the Gaussian parameters estimated on the first frame can be reused for the rest of the sequence. As an example, Fig. 13 shows an image sequence of size 352×240 segmented in six classes. At first, the fragment shader is loaded, compiled, and linked (approximately 1.5 s). The Gaussian parameters are then estimated on the first frame with *K*-means and ICE (approximately 2 s). Assuming the color distribution of every frame is similar, the Gaussian parameters are reused to segment the rest of the sequence. Segmenting the 30 frames with our hardware implementation of ICM took approximately 1.5 s for the entire sequence, i.e., an average of 0.05 s/frame. This represents a segmentation rate of 20 frames/s. Notice how little the difference is between the segmentation map of the last frame [Fig. 13(e)] inferred with the Gaussian parameters initially computed and the one obtained with the Gaussian parameters estimated on the last frame [Fig. 13(f)].

## 9 Conclusion

This paper exposed how programmable graphics hardware can be used to performed typical Markovian segmentation applied to energy-based and statistical problems. The results show that the parallel abilities of GPUs significantly accelerate these applications (by a factor of 4 to 200) without requiring any advanced skills in hardware programming. Such a hardware implementation is useful especially when the image size is large, when the number of labels is large, or when the observation field $y$ is composed of color images. Notice that a multiresolution version of every program could be implemented on a GPU, at the expense, however, of a more elaborate setup.

### *Acknowledgments*

The authors would like to thank Jean-François St-Amour for his precious help with hardware programming.

## References


1. L. Lucchese and S. Mitra, "Color image segmentation: A state-of-the-art survey," in *Proc. Indian National Science Academy*, pp. 207–221 (2001).
2. S. Geman and D. Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," *IEEE Trans. Pattern Anal. Mach. Intell.* **6**(6), 721–741 (1984).
3. J. Besag, "On the statistical analysis of dirty pictures," *J. R. Stat. Soc., Ser. C, Appl. Stat.* **48**(3), 259–302 (1986).
4. S. Li, *Markov Random Field Modeling in Computer Vision*, Springer-Verlag, New York (1995).
5. C. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford (1996).
6. W. Pieczynski, "Statistical image segmentation," *Mach. Graphics Vision* **1**(1), 261–268 (1992).
7. P. Bouthemy and P. Lalande, "Motion detection in an image sequence using Gibbs distributions," in *Proc. Int'l. Conf. on Acoustics, Speech, and Signal Proc.*, pp. 1651–1654 (1989).
8. J. Konrad and E. Dubois, "Bayesian estimation of motion vector fields," *IEEE Trans. Pattern Anal. Mach. Intell.* **14**(9), 910–927 (1992).
9. D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *Int. J. Comput. Vis.* **47**(1), 7–42 (2002).
10. "General-Purpose Computation Using Graphics Hardware," http://www.gpgpu.org/, July 2006.
11. K. Moreland and E. Angel, "The FFT on a GPU," in *Proc. Workshop Graphics Hardware*, pp. 112–119 (2003).
12. J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graphics* **22**(3), 908–916 (2003).
13. M. Rumpf and R. Strzodka, "Level set segmentation in graphics hardware," in *Proc. Int'l. Conf. on Image Proc.*, Vol. 3, pp. 1103–1106 (2001).
14. C. Dumontier, F. Luthon, and J-P. Charras, "Real-time dsp implementation for mrf-based video motion detection," *IEEE Trans. Image Process.* **8**(10), 1341–1347 (1999).
15. D. Murray, A. Kashko, and H. Buxton, "A parallel approach to the picture restoration algorithm of geman and geman on a simd machine," *Image Vis. Comput.* **4**(3), 141–152 (1986).
16. H-H. Nagel, "Image sequence evaluation: 30 years and still going strong," in *Proc. Int'l. Conf. on Pattern Recogn.*, pp. 1149–1158 (2000).
17. A. Mitiche and P. Bouthemy, "Computation and analysis of image motion: a synopsis of current problems and methods," *Int. J. Comput. Vis.* **19**(1), 29–55 (1996).
18. J. Barron, D. Fleet, and S. Beauchemin, "Performance of optical flow techniques," *Int. J. Comput. Vis.* **12**(1), 43–77 (1994).
19. M. Black and P. Anandan, "The robust estimation of multiple motions: parametric and piecewise-smooth flow fields," *Comput. Vis. Image Underst.* **63**(1), 75–104 (1996).
20. R. Duda, P. Hart, and D. Stork, *Pattern Classification*, 2nd ed., Wiley-Interscience, Hoboken, New Jersey (2000).
21. S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science* **220**(4598), 671–680 (1983).
22. R. J. Rost, *OpenGL Shading Language*, 1st ed., Addison-Wesley, Boston, Massachusetts (2004).
23. T. Akenine-Möller and E. Haines, *Real-Time Rendering*, 2nd ed., AK Peters, Wellesley, Massachusetts (2002).
24. R. Fernando and M. Kilgard, *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, Boston, Massachusetts (2003).


**Pierre-Marc Jodoin** graduated in 2000 as an engineer in computer science from the Ecole Polytechnique of Montreal, Canada, and received his MS degree in computer science in 2003 from the University of Montreal, Canada, where he is currently a PhD student in the Computer Science Department, under the supervision of Dr. Max Mignotte. He received three national research grants from FQNRT and the National Sciences and Engineering Research Council (NSERC) and his major research interests are realtime imaging, stereovision, and motion segmentation.

**Max Mignotte** received his DEA postgraduate degree in digital signal, image, and speech processing from the INPG University, Grenoble, France, in 1993 and his PhD degree in electronics and computer engineering from the University of Bretagne Occidentale (UBO) and the digital signal laboratory (GTS) of the French Naval Academy, France, in 1998. He was an INRIA postdoctoral fellow at University of Montreal (DIRO), Quebec, Canada, from 1998 to 1999. He is currently an assistant professor (Professeur adjoint) with DIRO in the Computer Vision & Geometric Modeling Lab at the University of Montreal. He is also a member of LIO (Laboratoire de recherche en imagerie et orthopedie, Centre de recherche du CHUM, Hopital Notre-Dame) and a researcher with CHUM. His current research interests include statistical methods and Bayesian inference for image segmentation (with hierarchical Markovian, statistical templates or active contour models), hierarchical models for high-dimensional inverse problems from early vision, parameters estimation, tracking, classification, shape recognition, deconvolution, 3-D reconstruction, and restoration problems.