Markovian Energy-Based Computer Vision Algorithms on Graphics Hardware

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

Université de Montréal, DIRO,
P.O. Box 6128, Studio Centre-Ville, Montréal, Québec, H3C 3J7
{jodoinp, mignotte, stamourj}@iro.umontreal.ca

Abstract. This paper shows how Markovian segmentation algorithms used to solve well known computer vision problems such as *motion estimation*, *motion detection* and *stereovision* can be significantly accelerated when implemented on programmable graphics hardware. More precisely, this contribution exposes how the parallel abilities of a standard Graphics Processing Unit (usually devoted to image synthesis) can be used to infer the labels of a label field. The computer vision problems addressed in this paper are solved in the maximum a posteriori (MAP) sense with an optimization algorithm such as ICM or simulated annealing. To do so, the *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. Results show that impressive acceleration factors can be reached, especially when the size of the scene, the number of labels or the number of iterations is large. Hardware results have been obtained with programs running on a mid-end affordable graphics card.

1 Introduction

Graphics hardware nowadays available are often equipped with a so called *Graphics Processing Unit* (GPU). This unit can execute general purpose programs independently of the CPU and the central memory. As the name implies, this architecture was optimized to solve typical graphics problems in 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. These primitives are processed either by the *vertex processor* or the *fragment processor*. What makes these processors so efficient is their fundamental ability to process vertices and fragments (see pixels) in parallel, involving interesting acceleration factors.

However, 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* (GPGPU). Some authors have shown that applications such as fast Fourier transforms [1], linear algebra [2], motion estimation and spatial segmentation could run on graphics hardware [3,4]. Even if these applications have little in common

F. Roli and S. Vitulano (Eds.): ICIAP 2005, LNCS 3617, pp. 592–603, 2005.

with traditional computer graphics, they all share a common denominator: they are problems solved by parallizable algorithms.

This paper presents how Markovian segmentation algorithms used to solve computer vision problems such as motion detection [5], motion estimation [6] and stereovision [7], can be significantly accelerated when implemented on a typical GPU. These computer vision problems are herein expressed by Markovian energy-based models through Gibbs distribution. This framework stipulates that a solution (also called *label field* or *segmentation map*) is *optimal* when it minimizes a global energy function made of a *likelihood* term and a *prior* term[8].

For all Markovian energy-based model, the label field has to be inferred by an optimization algorithm such as simulated annealing (SA) [9], ICM [10] or HFC [11]. Although ICM and HFC are much faster than SA, the processing time of these deterministic optimization algorithms can be very much prohibitive. In this contribution, we expose how optimizers such as SA or ICM –used to solve energy-based computer vision problems– can be significantly accelerated when implemented on programmable graphics hardware. Even if GPUs are cutting edge technologies made for graphics rendering, implementing a MAP segmentation algorithm on a fragment processor isn't much more difficult than writing it in a typical C-like procedural language.

The remainder of the paper is organized as follows. In Section 2, a review of the Markovian theory is proposed before Section 3 presents the three computer vision problems we are interested into. Section 4 presents the optimization algorithms ICM and SA after which Section 5 gives a look to the graphics hardware architecture. Finally, Section 6 shows some experimental results before Section 7 concludes.

2 Markovian Segmentation

The computer vision problems this contribution tackles can be seen as segmentation problems. As a matter of fact, these vision problems aim at subdividing observed input images into uniform regions by grouping pixels having high-level features in common such as motion or depth. Starting with some observed data Y (which is typically one or more input images), the goal of any segmentation process is to infer a label field X containing the class labels (i.e. labels indicating whether a pixel belongs or not to a moving area or a certain depth). In computer vision, 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 low-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 $\Gamma = \{e_1, e_2, ..., e_N\}$ where N is the total number of classes. In the case of motion detection for example, N can be set to 2 and $\Gamma = \{\text{StaticClass,MobileClass}\}$. Similarly, the observed value y_s takes a value in $\Lambda = \{\epsilon_1, \epsilon_2, ..., \epsilon_\zeta\}$ where ζ is set to 2^8 for gray-scale images and 2^{24} for color

images. In short, a segmentation model is made of 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 $\hat{x}$ which maximizes the a posteriori probability $P(X = x|Y = y)$ (that we represent by $P(x|y)$ for simplicity), also called the *maximum a posteriori* (MAP) estimate [8] : $\hat{x}_{\text{MAP}} = \arg\max_x P(x|y)$. 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 $P(y)$ isn't related to x. Assuming that X and Y are Markov Random Fields (MRF) and according to the Hammersley-Clifford theorem [8], 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

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

where $\lambda_{x|y}$ is a normalizing constant and $U(x, y)$ is an *energy function*. Combining Eq.(1) and (2), the optimization problem at hand can be formulated as an *energy minimization problem* i.e.:$\hat{x}_{\text{MAP}} = \arg\min_x \big(W(x, y) + V(x)\big)$, where $W(x, y)$ and $V(x)$ are respectively the likelihood and prior energy functions. If we assume that the noise in y isn't correlated, the global energy function $U(x, y)$ can be represented by a sum of local energy functions such as

$$\hat{x}_{\text{MAP}} = \arg\min_x \sum_{s \in S} \left(W_s(x_s, y_s) + V_{\eta_s}(x_s)\right). \tag{3}$$

Here, η_s is the neighborhood around site s and $V_{\eta_s}(x_s) = \sum_{c \in C_s} V_c(x_s)$ is a sum of potential functions defined on so-called cliques c. Function $V_c(x_s)$ defines the relationship between two neighbors defined by c, a binary clique linking a site s to a neighbor r. Notice that $\hat{x}_{\text{MAP}}$ is estimated with an optimization procedure such as SA or ICM which are typically slow algorithms. Details of these algorithms are discussed in Section 4.

3 Computer Vision Problems

3.1 Motion Detection

Among the first paper in computer vision in which a MAP framework was used is the one by Bouthemy and Lalande [5]. In their paper, they proposed a simple energy-based model to solve the problem of motion detection. The solution presented in this Section was inspired of their work.

The goal of motion detection is to segment an animated image sequence into *mobile* and *static* regions. For this kind of application, moving pixels are the ones with a non-zero displacement vector, no matter what direction or speed they might have. Bouthemy and Lalande's [5] paper influenced many subsequent

contributions including the one by Dumontier *et al* [12] 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.

As for Bouthemy and Lalande [5]'s method, the solution here proposed is based on the concept of temporal gradient and doesn't 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 df/dt 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 the following equation

$$W(x_s, y_s) = \frac{1}{\sigma^2}(y_s - m_p x_s)^2 \tag{4}$$

where m_p is a constant and σ 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 [5] and [12], the following Potts model was implemented

$$V_c(x_s) = \begin{cases} 1 & \text{if } x_s \neq x_r \\ -1 & \text{otherwise.} \end{cases} \tag{5}$$

The overall energy function to be minimized is thus defined by

$$U(x, y) = \sum_{s \in S} \Big(\underbrace{\frac{1}{\sigma^2}(y_s - m_p x_s)^2}_{W(x_s, y_s)} + \beta_{\text{MD}} \underbrace{\sum_{c \in \eta_s} V_c(x_s)}_{V_{\eta_s}(x_s)} \Big) \tag{6}$$

where η_s is a second order neighborhood (eight neighbors). Notice 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 inter-frame 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 optical motion over each site $s \in S$ of an animated sequence [13,14]. 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 at time $t+1$. Although this hypothesis excludes noise, scene illumination variation, and occlusion (and thus is an extreme simplification of the true physical nature of the scene) it allows simple energy functions to generate fairly good 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

$$\hat{\boldsymbol{v}}_s = \arg\min_{\boldsymbol{v}_s} |f_s(t) - f_{s+\boldsymbol{v}_s}(t+1)|, \quad \forall s \in S. \tag{7}$$

Notice that the absolute difference could be replaced by a cross-correlation distance. Such strategy is called *region-matching*. In the context of Eq.(7), the observation field y is the input image sequence and $y(t)$ is a frame at time t. The label field x is a vector field made of 2D vectors defined as $x_s = \boldsymbol{v}_s = (\zeta_i, \zeta_j)$ where ζ_i, ζ_j are integers taken between $-d_{max}$ and d_{max} as shown in Fig. 1 (b).

Eq.(7) has one major limitation which comes from the fact that real-world sequences contain textureless areas and/or areas with occlusions. Typically, over these areas more than one vector x_s have a minimum energy, although only one is valid. This is the well known *aperture problem* [15]. In order to guaranty the uniqueness of a consistent solution, several approaches have been proposed [13]. Among these approaches, many add 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.(5). However, since the number of labels can be large (here $(2d_{max}+1)^2$), we empirically observed that a smoother function was better suited. The following linear function was implemented : $V_c(\boldsymbol{x}_s) = \beta_{\text{ME}}\left(|\boldsymbol{x}_s[0] - \boldsymbol{x}_r[0]| + |\boldsymbol{x}_s[1] - \boldsymbol{x}_r[1]|\right)$, where c is a binary clique linking site s to site r. Notice that other smoothing functions are available [15]. The global energy function $U(x,y)$ to be minimized is obtained by combining Eq.(7) to $V_c(\boldsymbol{x}_s)$ as follows

$$U(x,y) = \sum_{s\in S} \Big(\underbrace{|y_s(t) - y_{s+x_s}(t+1)|}_{W_s(x_s,y_s)} + \beta_{\text{ME}} \underbrace{\sum_{\substack{c\in C_s \\ s,r\in c}} \left(|\boldsymbol{x}_s[0] - \boldsymbol{x}_r[0]| + |\boldsymbol{x}_s[1] - \boldsymbol{x}_r[1]|\right)}_{V_{\eta_s(x_s)}} \Big).$$

Let us mentioned that Konrad and Dubois [6] did proposed a similar solution - short of a *line process* they used to help preserve edges.

3.3 Stereovision

The goal of stereovision is to estimate the relative depth of 3D 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 [7]). To simplify the problem, stereovision algorithms often make some assumptions on the true nature of the scene. One common assumption (which is similar to motion estimation's lightness consistency assumption) states that every point visible in one image is also visible (with the same color) in the second

image. Thus, the goal of stereovision algorithms based on such assumption 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* which is, in this case, proportional to the inverse depth of the object projected on site s. This gives rise to a *matching cost* function that measures how good a disparity $d_s \in [0, D_{\text{MAX}}]$ is for a given site s in a reference image y_{ref}. This is expressed mathematically by

$$C(s, d, y) = |y_{\text{ref}}(i, j) - y_{\text{mat}}(i + d_s, j)| \tag{8}$$

where y_{mat} is the second image familiarly called the *matching image*. Notice that the absolute value could be replaced by a quadratic or a robust function [7]. In the context of the MAP, $C(.)$ is the likelihood energy function and the disparity map d is the label field to be estimated. Thus, to ensure uniformity with Section 2's notation, the cost function of Eq.(8) will be defined 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 [7] 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. Spatial smoothness can also be ensured by adding a prior energy term $V_{\eta_s}(x)$ of the form $V_{\eta_s}(x) = \sum_{s \in S} \sum_{c \in \eta_s} |x_s - x_t|$. Notice that the absolute value could be replaced by another cost function if needed. The global energy function $U(x, y_{\text{ref}}, y_{\text{mat}})$ can thus be written as

$$U(x, y_{\text{ref}}, y_{\text{mat}}) = \sum_{s \in S} \Big(\underbrace{(w * C)(s, x, y)}_{W_s(x_s, y_s)} + \beta_{\text{S}} \underbrace{\sum_{\substack{c \in \eta_s \\ s, r \in c}} |x_s - x_t|)}_{V_{\eta_s}(x_s)} \Big) \tag{9}$$

where β_{S} is a constant. Notice that some authors minimize only the likelihood energy function $W(x, y)$ assuming the low-pass filter is enough to ensure spatial smoothness. This strategy, called *Winner-Take-All* (WTA), is greedy and converges after only one iteration. Another way to save on processing time is to pre-compute the likelihood function in a 3D table. Such table is called the *Disparity Space Integration* (DSI) and contains $\mathcal{N} \times \mathcal{M} \times D_{\text{MAX}}$ cost values.

4 Optimization Procedures

Since Eq.(3) has no analytical solution, $\hat{x}$ has to be estimated with an *optimization* algorithm. An optimization procedure we have implemented is the simulated annealing (SA) which is a stochastic relaxation algorithm similar to the Gibbs sampler. The concept of SA is based on the manner in which some material recrystallize 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 T [9] from an initial temperature T_{MAX} down to zero. In this

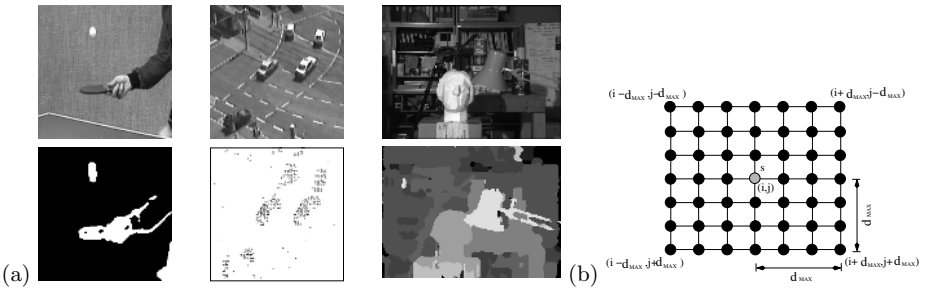

Fig. 1. (a) Motion detection, motion estimation, and stereovision label fields obtained after an ICM optimization. (b) The total number of possible displacement vector for each site $s \in S$ is $(2d_{max}+1)^2$.

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 [9] which can be written as

$$P(x,y) = \frac{1}{\lambda}\exp\{-\frac{U(x,y)}{T}\}. \tag{10}$$

where λ is a normalization factor. The SA algorithm is presented in Table 1.

Table 1. Simulated annealing and ICM algorithms

1 $T \leftarrow T_{\text{MAX}}$
2 For each site $s \in S$
2*a* $P(x_s = e_i \vert y_s) = \frac{1}{\lambda}\exp\left\{\frac{-1}{T}U(e_i, y_s)\right\}, \quad \forall e_i \in \Gamma$
2*b* $x_s \leftarrow$ according to $P(x_s \vert y_s)$, randomly select $e_i \in \Gamma$
3 $T \leftarrow T*$cooling Rate
5 Repeat steps 2-3 until $T \leq T_{\text{MIN}}$

1 Initialize x
2 $x_s = \arg\min_{e_i \in \Gamma} U(e_i, y_s)\ \forall s \in S$
3 Repeat step 2 until x stabilizes

The major limitation with SA is the number of iterations it requires to reach the frozen ground state. This makes SA unsuitable to many applications for which time is an important factor. This explains the effort of certain researchers to find faster optimization procedures. One such optimization procedure is Besag's ICM algorithm [10]. 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 $e_i \in \Gamma$ that minimizes the energy function at that point. Since ICM isn't stochastic, it cannot exit local minima and thus, requires x to be initialized near the global minima. ICM algorithm is presented in Tab 1.

5 Graphics Hardware Architecture

Most graphics hardware are designed to fit the so-called *graphics processing pipeline* [16,17]. 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 in graphics hardware has been that 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. Because the GPU is an inherently parallel processing unit, mapping general computation problems to its unique architecture becomes very interesting [3].

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). 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. This location covers one or many pixels.

5.1 Markovian Segmentation on GPU

As one might expect, fragment programs (also called *fragment shader*) have some specificities as compared to ordinary C/C++ programs. The most important ones are the following:

1. a fragment program is made to process each fragment in parallel;
2. the *framebuffer* and the *depthbuffer* are the only memory a fragment program can write into;
3. the only data a fragment program can read is contained in the texture memory, in built-in variable or in user-defined variable. It cannot read the content of the framebuffer or the depthbuffer;
4. since a fragment program cannot read the framebuffer and since each fragment are processed in parallel, fragment programs cannot exchange information. GPUs do not provide its programs with access to general-purpose memory.

With such limitations, minimizing a global Markovian energy function such as Eq.(3) can be tricky. In fact, three main problems have to be overcome. Firstly, when performing a Markovian segmentation, fragment operations should be performed on every pixel of the input scene. As such, a $1:1$ 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 as the input image. This generates exactly the right amount of fragments in the graphics pipeline such that a label estimated by a fragment program will be copied into one and only one pixel of the framebuffer. Such implementation

is illustrated in Table 2. Notice that a fragment program is launched over each pixel when the rectangle is rendered (line 4). In this way, the $\mathcal{N} \times \mathcal{M}$ label field x is estimated with the help of one CPU program and $\mathcal{N} \times \mathcal{M}$ fragment programs. In other words, the CPU program renders the scene and manages the texture memory while a fragment program minimizes the energy function $U(x_s, y_s)$ for each pixel.

Table 2. High level representation of an ICM hardware segmentation program. For an SA implementation, a temperature factor as well as a cooling rate shall be added. The first program (line 1 to 7) is the C/C++ CPU program loading the fragment program, rendering the scene and managing textures. The second program (line 1-2) is the fragment program launched on every pixel when the scene is rendered (line 4). Notice that images x and y are contained into texture memory.

1	Copy the input images into texture memory
2	Compile, link and load fragment shader (FS) on the GPU
3	Specify parameters of FS (β_{MD},β_{ME} or β_{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 steps 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 second problem comes from the fourth limitation. Since GPUs provide no general-purpose memory, one might wonder how the prior energy function V_{η_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 2, after rendering the scene, the CPU program copies the framebuffer into texture memory (line 5). In this way, the texture memory contains not only the input images, but also 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 [10] and successfully tested by other authors [12]. Notice that the iterative nature of ICM and SA is reproduced with multiple rendering of the rectangle (lines 4-5-6).

The last problem with shaders comes with their inability to generate random numbers such as needed by SA. As a workaround, we generate a random image that we copy in texture memory where the shader can access it.

5.2 Computer Vision on GPU

With the technique illustrated in table 2, performing motion detection, motion estimation and stereovision on a GPU is fairly straightforward. Since the shading languages available to write fragment programs (NVIDIA's *Cg* language in our case) are similar to C, the software programs can be reused almost directly.

The implementation of the three fragment shaders is conceptually very similar since they all minimize an energy function made of a likelihood term and a prior term. There is one exception though when stereovision requires a pre-filtering step ($(w * C)(s, x, y)$). We deal with this situation by pre-computing $C(s, x_s, y_s)$ in a 3D DSI table located in texture memory. This 3D table is then filtered with w after which the optimization procedure (SA, ICM or WTA) is launched.

6 Experimental Results

Results compare software and hardware implementations of the three 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++ while NVIDIA's Cg language was used to implement the fragment shaders. All programs were executed on a conventional home PC equipped with a AMD Athlon 2.5 Ghz, 2.0 Gig of RAM and a NVIDIA 6800 GT graphics card. NVIDIA fp40 Cg profile was used in addition to the cgc compiler version 1.3.

Every results were made after varying some variables. In Fig. 2, the lattice size vary between 64×64 and 1024×1024, the number of disparities D_{MAX} between 4 and 32, d_{MAX} between 2 and 6, and the aggregation window size

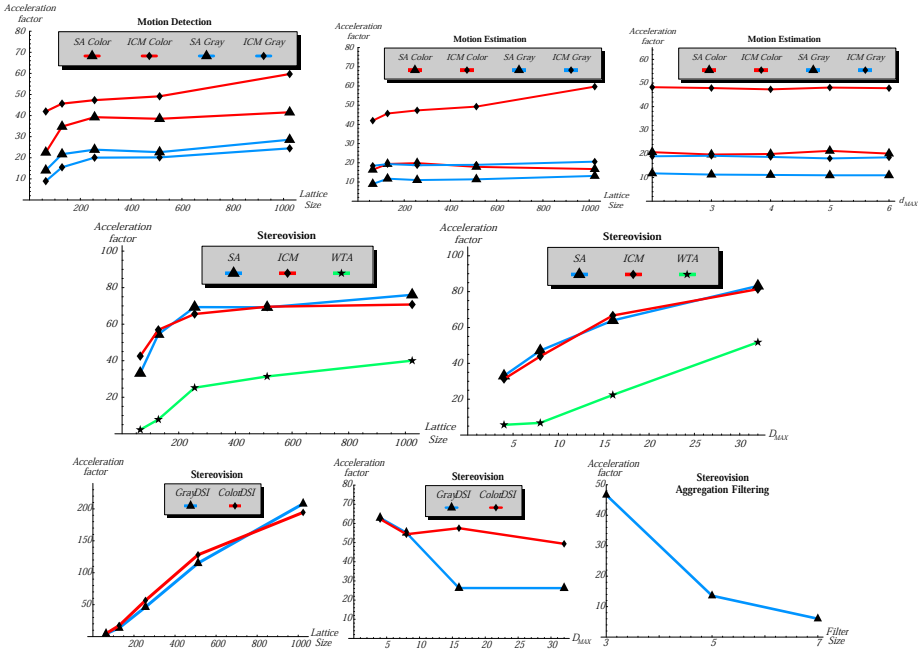

Fig. 2. Acceleration factor for the motion detection, motion estimation and stereovision programs

between 3×3 and 7×7. The number of iterations was set to 10 for ICM and to 500 for SA. Every results are expressed as an acceleration factor between the software and hardware execution time.

As shown in Fig 2, the hardware implementation is faster than its software counterpart by a factor between 10 and 100 approximately. Notice that the acceleration factor is generally more important for color sequences than for grayscale sequences. This is explained by the fact that the likelihood energy function W is more expensive to compute with RGB vectors than for grayscale values. Thus, distributing this extra load on a fragment processor results in a more appreciable acceleration factor.

For stereovision, we have tested the three tasks we have made reference to in Section 5.2, namely the computation of DSI, the aggregation filtering, and the optimization procedure (SA, ICM and WTA). As can be seen, the acceleration factor for ICM and SA is more important than for WTA. This can be explained by the fact that WTA is a very trivial and efficient algorithm (it converges in only one iteration). The computational load to distribute on the GPU is thus less important than for ICM and SA.

7 Conclusion

This paper shows how programmable graphics hardware can be used to performe typical energy-based segmentation applied to computer vision. Results show that the parallel abilities of GPUs significantly accelerate these applications (by a factor of 10 to 100 approximately) without requiring any specific skills in hardware programming. Such hardware implementation is usefull especially when the image size is large, when the number of labels is large or when the number of iteration is large.

References

1. Moreland K. and Angel E. The fft on a gpu. In *in proc. of Workshop on Graphics Hardware*, pages 112–119, 2003.
2. Kruger J. and Westermann R. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.
3. http://www.gpgpu.org/.
4. R. Strzodka and M. Rumpf. Level set segmentation in graphics hardware. In *Proc. of ICIP*, 3, pages 1103–1106, 2001.
5. P. Bouthemy and P. Lalande. Motion detection in an image sequence using gibbs distributions. In *Proc. of ICASSP*, pages 1651–1654, 1989.
6. Konrad J. and Dubois E. Bayesian estimation of motion vector fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(9):910–927, 1992.
7. Scharstein D., Szeliski R., and Zabih R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proc. of the IEEE Workshop on Stereo and Multi-Baseline Vision*, 2001.
8. 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.

9. Kirkpatrick S., Gelatt C., and Vecchi M. Optimization by simulated annealing. *Science*, 220, 4598:671–680, 1983.
10. Besag J. On the statistical analysis of dirty pictures. *J. Roy. Stat. Soc.*, 48(3):259–302, 1986.
11. Chou P. and Brown C. The theory and practice of bayesian image labeling. In *Proc. of ICCV*, pages 185–210, 1990.
12. 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.
13. Nagel H-H. Image sequence evaluation: 30 years and still going strong. In *proc. of ICPR*, pages 1149–1158, 2000.
14. Mitiche A. and Bouthemy P. Computation and analysis of image motion: a synopsis of current problems and methods. *Int. J. Comput. Vision*, 19(1):29–55, 1996.
15. Black M. and Anandan P. The robust estimation of multiple motions: parametric and piecewise-smooth flow fields. *Comput. Vis. Image Underst.*, 63(1):75–104, 1996.
16. Randi J. Rost. *OpenGL Shading Language.* Addison-Wesley, 1st edition, 2004.
17. Tomas Akenine-Möller and Eric Haines. *Real-time Rendering.* AK Peters, 2e edition, 2002.