J Supercomput (2014) 68:339–364
DOI 10.1007/s11227-013-1042-4

Medical image segmentation with deformable models on graphics processing units

Rigo Alvarado · Juan J. Tapia · Julio C. Rolón

Published online: 17 December 2013
© Springer Science+Business Media New York 2013

Abstract In this work, the parallel implementation of a segmentation algorithm based on the gradient vector flow (GVF) deformable model in a graphics processing unit (GPU) is presented. The proposed implementation focuses on the parallelization of the computation of the GVF field. In order to make a performance comparison of the proposed GPU algorithm, an OpenMP-based implementation is presented too. We also present an analysis of the textures and global memory performance in the computing of the GVF field. To improve the efficiency and the performance of the active contour segmentation, a novel snaxel reallocation method is proposed. The main advantage of the reallocation process is the small linear system needed to perform the segmentation and its low computational load. To assure the convergence of the active contour deformation, we propose a stopping criterion based on the root mean square error for the iterative solution of the evolution equations.

Keywords GPU · CUDA · Deformable models · Segmentation · Snaxel reallocation · Medical image · OpenMP

1 Introduction

The segmentation of images is one of the most important stages in the digital image analysis. Segmentation is a process in which the image is divided into regions that

R. Alvarado · J. J. Tapia (✉) · J. C. Rolón
Instituto Politécnico Nacional, CITEDI Research Center, Avenida del Parque 1310,
Mesa de Otay, 22510 Tijuana, BC, México
e-mail: jtapiaa@ipn.mx; jjtapia@citedi.mx; juan.tapia@gmail.com

R. Alvarado
e-mail: ralvarado@citedi.mx

J. C. Rolón
e-mail: jcrolon@ipn.mx

in general have irregular shape and that are separated by the contours that bound those regions. Conceptually, the objective of the segmentation process is to separate the different shapes, structures or objects that are located inside the image [1]. Some applications of segmentation are face recognition, traffic control systems, fingerprint recognition and medical image analysis.

The applications of medical image segmentation include the detection of tumors, the measurement of tumor volume and its response to therapy over time and surgery simulations, to name a few. In some cases, segmentation dictates the outcome of the entire analysis, since measurements and other processing steps are based on the segmented regions. Most of the segmentation methods are based on the intensity of image pixels, though neural networks and model-based algorithms are used as segmentation tools too [1,2].

Deformable models, also known as active contours, are the most commonly model-based technique [3–5] used in medical image processing. Their spread use stems from their ability to incorporate *a priori* information about the regions of interest within the image. Furthermore, deformable models support interaction mechanisms that enable medical personnel to modify their behavior when necessary [6]. These characteristics allow the use of deformable models for different image processing tasks such as segmentation, tracking and matching [7,8]. Deformable models can be applied to different medical images sources, e.g. magnetic resonance (MRI), computed tomography (CT) or radiography.

Although the computational load of their algorithm is commonly considered as a drawback to the use of deformable models in some applications, with the advent and widespread use of high-performance computing (HPC) platforms like graphics processing units (GPUs), parallel real-time applications for deformable models are nowadays attainable.

There are many works that involve the implementation of deformable models on GPU cards. In [9], one of the first OpenGL attempts to parallelize on a GPU the gradient vector flow (GVF) field computing is presented. Zheng and Zhang [10] made a GVF deformable model implementation on a GPU that uses texture arrays for the computation of the GVF field and for the iterative solution of the evolution equations. Perrot et al. [11] presented a large image segmentation algorithm based on a statistical deformable model implemented on a GPU. In Li et al. [12], implemented a variation of the Geodesic deformable model on a GPU. Smistad et al. [13] made a performance analysis for the GPU memory spaces applied to the GVF field computation. The segmentation on large images using a tile approach with the GVF active contour is presented in [14]. Other GPU-based image processing works are Češnovar et al. [15] where the authors used semantic classification on large datasets of aerial-images and Valero et al. [16] where the authors presented a Markov Random Fields (MRF) classification of MRI images. Despite its wide use, some research groups and vendors have made attempts to define APIs and languages that simplify the GPU programming in an effort to make it accessible to a large audience, as can be seen in [17]. Besides these GPU-based implementations, some other HPC architectures had been used in the implementation of deformable models, e.g. Lenkiewicz et al. [18] presented a deformable model segmentation algorithm designed for computer clusters or multi-core architectures.

In general, an author presents tables with execution times as proof of the success in the GPU implementations. Nevertheless, although the timing values are the most important issue to consider in parallel implementation, a set of other computational metrics will provide a better evaluation of a given implementation. In [19] the authors defined four important criteria for the evaluation of GPU implementations: performance, programming comfort, accessibility, and cost-effectiveness. Pallipuram et al. [20] made a comparative study of GPU architectures and programming models to determine which platform is the most suitable for a given application. In order to improve the efficiency of GPU applications that involve a large quantity of data transfers between the CPU and GPU, in [21] a convolution for audio applications with the overlapping of data transfer and computational work is presented.

However, some of the cited papers only focus on the GVF field computing and not in a segmentation process. Other works use image processing techniques outside the deformable models theory and other researches only focus on the GPU implementation evaluation. Although a segmentation technique comparison is beyond the scope of this paper, in this investigation we focus on the efficient computation of the GVF field and also in the segmentation results applying deformable models. To address the performance analysis we make a comparison between the parallelization of the GVF field computing with OpenMP and CUDA; to improve the efficiency of the segmentation process of the GVF active contour we proposed a snaxel reallocation approach based on a dynamic mesh adaptation.

Therefore, the main contributions of this work are (1) the objective analysis of the GVF field computing on CPU and GPU and (2) the improvement of the segmentation process with the proposed dynamic snaxel reallocation. Besides the OpenMP and CUDA comparison, we also present an analysis of the GVF field computing using textures and global memory. It is important to clarify that though the GVF snake could be formulated as an interactive segmentation tool, we did not consider it that way because we focus on the acceleration of the algorithm, so we designed it as an autonomous segmentation tool.

In Sect. 2 an introduction to the theory of deformable models is presented. The original active contour of Kass et al. [24] as well as the GVF active contour [22] is described. The proposed snaxel reallocation approach is described in this section too. Section 3 presents a brief introduction to the CUDA programming model and the GPU general architecture. Section 4 details the parallel implementations that we proposed and the implementation of our snaxel reallocation method. Section 5 presents the experimental results of the proposed reallocation technique, the medical images segmentation and the metrics that we defined to evaluate our implementations; finally, these results are discussed in Sect. 6.

2 Deformable models

Deformable models are differential equations that determine the shape and movement of curves (in the case of 2-dimensional signals) or surfaces (in the case of 3-dimensional signals) built of an abstract elastic material. The physical interpretation of a deformable model is that of an elastic body that responds to the forces applied on it [6,23].

In a segmentation process, the interpretation of a deformable model is that of an elastic curve that is introduced into an image plane. The curve deforms from its initial configuration due to external and internal forces applied to it, until its shape resembles the boundary of a region of interest within the image. In [24], due to its behavior, the most common deformable model is called snake.

2.1 Traditional active contour

The traditional active contour or snake is defined in [6] as a contour that is embedded in the image plane $I_M(x, y) \in \mathbf{R}^2$. The position of the active contour is $\mathbf{v}(s) = (x(s), y(s))^{\mathrm{T}}$, where x and y are coordinate functions and $s \in [0, 1]$ is the parametric domain. Snakes can be formulated as open or closed contours. The shape of the active contour within the image $I_M(x, y)$ is determined by the energy functional [6,24]

$$\varepsilon(\mathbf{v}) = \mathcal{S}(\mathbf{v}) + \mathcal{P}(\mathbf{v}), \tag{1}$$

where $\mathcal{S}(\mathbf{v})$ is the internal deformation energy and $\mathcal{P}(\mathbf{v}(s))$ represents the external deformation energy. The internal deformation energy of the active contour is defined in [6] as

$$\mathcal{S}(\mathbf{v}) = \int_0^1 (\alpha(s)|\mathbf{v}_{\mathrm{s}}|^2 + \beta(s)|\mathbf{v}_{\mathrm{ss}}|^2)\,\mathrm{d}s. \tag{2}$$

The term $\mathbf{v}_{\mathrm{s}}$ denotes the first derivative of $\mathbf{v}$ with respect to s and represents the elasticity of the active contour. The term $\mathbf{v}_{\mathrm{ss}}$ represents its rigidity and denotes the second derivative of $\mathbf{v}$ with respect to s. Two non-negative functions define the behavior of the physical energies that are simulated on the active contour: $\alpha(s)$ controls the elastic tension of the contour and $\beta(s)$ controls its rigidity [6]. The tension energy models the behavior of a rubber band and the rigidity energy models the behavior of a flexible bar.

Minimizing $\alpha(s)|\mathbf{v}_{\mathrm{s}}|^2$ causes the contraction of the active contour down to a point; the minimization of $\beta(s)|\mathbf{v}_{\mathrm{ss}}|^2$ causes the active contour to take a circular shape in the case of a closed contour or a straight line in the case of an open contour [25]. This implies that, given an initial size of the active contour and assuming that the external force is null, the snake will never grow; it will only tend to form a circle, in the case of a closed contour, while shrink towards its center.

The second term of Eq. (1) is the external deformation energy of the active contour that couples the snake to the image. It is defined as

$$\mathcal{P}(\mathbf{v}) = \int_0^1 P(\mathbf{v}(s))\,\mathrm{d}s, \tag{3}$$

where $P(x, y)$ is a scalar function defined on the image plane [6]. $P(x, y)$ is called external energy because is obtained from sources outside the contour and is commonly calculated as

$$P(x, y) = -|\nabla[G_\sigma(x, y) * I_M(x, y)]|, \tag{4}$$

where $I_M(x, y)$ is the image and $G_\sigma(x, y)$ is the Gaussian filter

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{|x|^2 + |y|^2}{2\sigma^2}\right). \tag{5}$$

The active contour $\mathbf{v}(s)$ that minimizes the energy functional of Eq. (1) satisfies the Euler–Lagrange equation

$$-\alpha\mathbf{v}_{\mathrm{ss}}(s) + \beta\mathbf{v}_{\mathrm{ssss}}(s) + \nabla P(v(s, t)) = 0. \tag{6}$$

Equation (6) expresses the balance of the internal and external forces of the snake in equilibrium state, i.e. when the active contour is steady. Each term of Eq. (6) corresponds to a force produced by the respective energy. The first two terms represent the internal stretching and bending forces respectively, while the third term represent the external force that couples the snake to the image [6]. To completely specify the mathematical model of Eq. (6), it is assumed that the boundary conditions are known (see e.g. the conditions in [3,26,27])

$$\mathbf{v}(0), \mathbf{v}(1), \mathbf{v}'(0), \mathbf{v}'(1). \tag{7}$$

In the discrete domain, the contour $\mathbf{v}$ is represented by a set of points called snaxels. The discretization of Eq. (6) produces the iterative equations that calculate the Cartesian coordinates of the snaxels. These discrete iterative equations are

$$x^{t+1} = \left(\frac{I}{\Delta t} + A\right)^{-1} \left(\kappa f_{x^t} + \frac{x^t}{\Delta t}\right), \tag{8}$$

$$y^{t+1} = \left(\frac{I}{\Delta t} + A\right)^{-1} \left(\kappa f_{y^t} + \frac{y^t}{\Delta t}\right), \tag{9}$$

where I is the identity matrix, A is the sparse differentiation matrix that represents $\mathbf{v}_{\mathrm{ss}}(s)$ and $\mathbf{v}_{\mathrm{ssss}}(s)$, Δt is the time step of the iterations and κ is a constant used to control the external force influence [22]. The addition of the matrix $\frac{I}{\Delta t}$ to the matrix A makes it non-singular so it can be inverted. The matrices I and A are of size N^2 with N the number of points of the initial active contour. The terms $f_x = -\frac{\partial E_I}{\partial x_i}$ and $f_y = -\frac{\partial E_I}{\partial y_i}$ of Eqs. (8) and (9) respectively depend on the calculation of the external energy of Eq. (4) [6].

The use of a gradient-based external energy function like Eq. (4) produces large energy values over and in the close neighbourhood of the contours of the image. This characteristic, together with the behavior of the internal forces, gives the traditional snake two main drawbacks: (1) the need for an initialization process that places the snake close to the region of interest, and in the case of a closed contour outside the region of interest so that eventually the snake will find it and (2) the incapacity of the snake to progress into boundary concavities [5]. The reason behind these limitations is the small capture range produced by Eq. (4). The capture range is defined as the area of influence of the contour in which the snake is able to find a local minimum; its extension is closely related to the external energy.

To address the problems of small capture range and convergence in boundary concavities, the GVF snake proposed in [22] is one of the most used methods [5–7]. This is due to its effective improvement in capture range extension and its ability to converge inside boundary concavities. In fact, some investigations about deformable models are related to modifications and improvements to the original GVF snake proposal [6,28].

2.2 Gradient vector flow active contour

The GVF active contour represents an improvement to the snake defined by Eq. (6). The improvement is a vector field $\mathbf{w}(x, y) = [u(x, y), v(x, y)]$ that represents the external force. Therefore, the equation that models the GVF active contour is [22]

$$-\alpha \mathbf{v}_{\mathrm{ss}}(s) + \beta \mathbf{v}_{\mathrm{ssss}}(s) + \mathbf{w} = 0. \tag{10}$$

Equation (10) is solved numerically in the same way as Eq. (6). In the case of the GVF model, **w** is a vector field, moreover, it is the gradient vector flow field; this is the main difference with respect to the traditional active contour model of Eq. (6). The GVF field solves the problems of small capture range and convergence in boundary concavities associated to the active contour of Kass. It is a dense vector field derived from an image by the minimization of an energy functional. The vector nature of **w** is the improvement of the GVF model over the original active contour model that makes the former more efficient [22].

Unlike the traditional snake, a GVF snake could be initialized inside, across or outside the region of interest. This is because the GVF field produces a wide capture range which is the product of an isotropic diffusion process that does not blur the edges within the image, as a Gaussian filter would do [5,22].

The external force used in the formulation of the traditional snake is an irrotational field and that is why the snake is unable to converge inside boundary concavities. The GVF field incorporates an irrotational component and also a curl component so it outperforms the traditional snake [22].

The GVF field formulation focuses on keeping the gradient properties nearby the edges within the image and on extending the scope of the normal vectors of the edges beyond their nearby regions through a diffusion process. This process produces an external force with a significative magnitude over the whole image plane, not only in the edges neighborhood [3]. Due to the competition among the vectors of edges that involves a diffusion process on an image, some of the vectors of the GVF field, according to the geometry, point inward the concavities. This particular feature solves the problems of the capture range and the lack of convergence inside the concavities of the snake of Kass [22].

The first step to calculate the vector field **w** is to obtain the contour map $f(x, y)$ derived from the image $I_M(x, y)$. The contour map is a potential function defined over the image whose value is larger near the image edges. In this work, the Canny edge detector [29] is used to generate the contour map.

The GVF field is defined as the vector field $\mathbf{w}(x, y) = [u(x, y), v(x, y)]$ that minimizes the energy functional

$$\varepsilon = \int\int \mu(u_x^2 + u_y^2 + v_x^2 + v_y^2) + |\nabla f|^2 |v - \nabla f|^2 \,\mathrm{d}x\mathrm{d}y \tag{11}$$

where $f(x, y)$ is the contour map of the image; ∇ is the gradient operator; μ is a positive parameter of regularization; $u(x, y)$ and $v(x, y)$ are functions that represent the GVF field components; and u_x, u_y, v_x and v_y represent the partial derivatives of $u(x, y)$ and $v(x, y)$ with respect to x and y, respectively [5,22]. The energy functional of Eq. (11) keeps $\mathbf{w} \approx \nabla f$ when ∇f is large; otherwise, it produces a slowly varying field in the homogeneous regions [22].

Applying the calculus of variations to Eq. (11), the equation that solves $u(x, y)$ (Eq. 12) and the equation that solves $v(x, y)$ (Eq. 13) are obtained [1,22]

$$\mu\nabla^2 u - (u - f_x)(f_x^2 + fy^2) = 0, \tag{12}$$

$$\mu\nabla^2 v - (v - f_y)(f_x^2 + f_y^2) = 0, \tag{13}$$

where ∇^2 is the Laplacian operator. The solution of Eqs. (12) and (13) generate the vector field $\mathbf{w}$. To implement the GVF snake model, the vector field $\mathbf{w}$ is computed first. Next, Eqs. (8) and (9) are used to minimize Eq. (10).

From Eqs. (12) and (13), the components $u(x, y)$ and $v(x, y)$ of the GVF field are

$$u_t(x, y, t) = \mu\nabla^2 u(x, y, t) - [u(x, y, t) - f_x(x, y)][f_x(x, y)^2 + f_y(x, y)^2], \tag{14}$$

$$v_t(x, y, t) = \mu\nabla^2 v(x, y, t) - [v(x, y, t) - f_y(x, y)][f_x(x, y)^2 + f_y(x, y)^2], \tag{15}$$

where the Laplacian is approximated in the discrete domain using centered finite-difference equations

$$\nabla^2 u = u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}, \tag{16}$$

$$\nabla^2 v = v_{i-1,j} + v_{i+1,j} + v_{i,j-1} + v_{i,j+1} - 4v_{i,j}. \tag{17}$$

The iterative condition of Eqs. (14) and (15) is denoted by t. The capture range of the GVF field is determined by these iterations, i.e. the produced field is proportional to the number of iterations. This way of controlling the capture range enables the GVF method to eliminate the location dependency for the initial active contour. If a certain shape of interest is not included or is partially included in the initial snake, the GVF field will allow deformations of the initial active contour to include the shape. If the shape of interest is located completely out of the initial active contour, it is even possible that the complete snake moves towards the shape to capture it, provided there are not additional shapes in the neighbourhood.

Because the mathematical formulation of the GVF snake is similar to the formulation of the traditional snake, the GVF snake inherits some limitations, e.g. the sensitivity of the internal energy parameters [6,30]. However, the main drawback of GVF snake is the large number of arithmetic operations involved, which directly affects its computation time [28].

As the evolution of Eqs. (14) and (15) depends only on current information, the numerical solution of these equations produce explicit discrete iterative equations. At each iteration, there is no data dependency between the elements of the component $u(x, y)$; this is also true for the $v(x, y)$ elements. For this reason, the GVF field can be computed in parallel on a high-performance computing platform and this way reduce the computation time of GVF snake algorithm.

2.3 Dynamic snaxel reallocation

One problem associated with the snake formulation is the inadequate grouping of snaxels of a closed contour. At certain sections of a boundary, the snaxels tend to group together into clusters whereas in some other regions they tend to separate from each other. This inadequate distribution is illustrated in Fig. 1.

As shown in Fig. 1, the white snaxels are relatively apart from each other whereas the black snaxels are clustered. This effect is not contemplated in the energy functional of the snake although it can derive into unwanted behavior of the active contour, leading to an incorrect final segmentation result. Even when the tension term α of Eq. (6) can be viewed as the internal force that somehow controls the spacing between snaxels, the overall effect of the total forces that affect each snaxel leads to the spacing problems described.

Consider the contour of Fig. 1 and the case where the tension force is set to a relatively higher value than the other forces. Under this condition, large spaces between the snaxels are not expected but on the other hand, the snake will not be able to converge inside the concave region. This is because the internal tensional force between the snaxels is greater than the external force of the boundary of interest and therefore, the snaxels placed over the contour in the proximity of the concave region will not allow the snaxels placed near the concave region enter it. On the contrary, if α is set to a relatively lower value than the other forces, the snake will be able to converge inside concave regions, but the spacing problems shown in Fig. 1 will be present. In order to solve this problem, a mesh adaptation approach for the snaxels is necessary.

In general, a mesh adaptation can be static or dynamic. In the static method, also known as local mesh refinement, some nodes are added wherein the solution has high

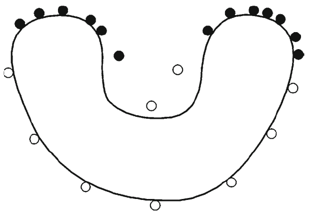


Fig. 1 Typical distribution of snaxels over a border

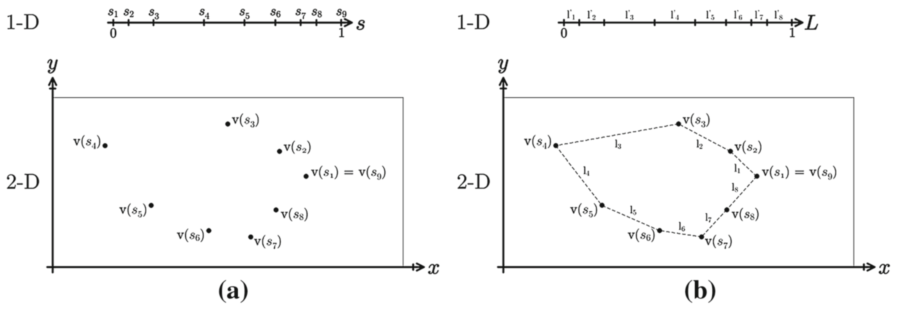

1-D
s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9
0
1
s
2-D
y
x
v(s_1) = v(s_9)
v(s_2)
v(s_3)
v(s_4)
v(s_5)
v(s_6)
v(s_7)
v(s_8)
(a)
l'_1 l'_2 l'_3 l'_4 l'_5 l'_6 l'_7 l'_8
L
l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8
(b)

Fig. 2 **a** Parametric closed contour and **b** normalized arc length of a contour

gradient values and are removed from regions where the solution is almost constant. In the dynamic approach, the number of nodes is fixed and they are only re-allocated based on the characteristics of the solution.

Consider the parametric contour $\mathbf{v}$ of Fig. 2a. As we are considering only closed contours, $\mathbf{v}(s_1) = \mathbf{v}(s_9)$. In Fig. 2a it is clear that the points over the plane are not equally spaced and this condition is reflected over the s domain. It is also clear that each point of $\mathbf{v}$ is defined as $\mathbf{v}(s_i) = \mathbf{v}_i(x_i, y_i)$ and thus, any movement of the s_i points will be reflected in the configuration of the contour $\mathbf{v}$. Considering these conditions, a mesh adaptation technique can be applied over the s domain to equidistribute the s_i points and hence, the points that form the contour $\mathbf{v}$. However, the problem with this approach is the transformation of the s_i values into the corresponding (x_i, y_i) coordinates, i.e. a $\mathbb{R} \rightarrow \mathbb{R}^2$ transformation.

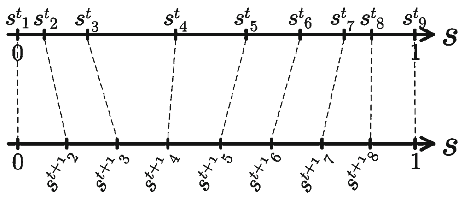
To solve the mesh adaptation issue the normalized arc length value between the points of $\mathbf{v}$ contour is used; the slope value between the ordered pairs (x, y) of consecutive $\mathbf{v}$ points is used to perform the $\mathbb{R} \rightarrow \mathbb{R}^2$ mapping. According to the s domain sequence, the arc length and the slopes values of the $\mathbf{v}$ points are computed in a counterclockwise sense on the plane.

The arc length is calculated as $l_i = \sqrt{\Delta x_i^2 + \Delta y_i^2}$ where $\Delta x_i = x_{i+1} - x_i$ and $\Delta y_i = y_{i+1} - y_i$. The normalized arc length is defined as $l'_i = \frac{l_i}{L_T}$ and is used to determine the position of each s_i point over the s domain. L_T is defined as the total length of the contour. Figure 2b shows the normalized arc length for the contour $\mathbf{v}$ illustrated in Fig. 2a.

To perform the mesh refinement over the s domain a nodal reallocation approach is used. The arc length is utilized as the mesh adaptation function that drives the mesh adaption process, with larger values denoting a region for refinement and smaller values denoting a region for mesh coarsening. The arc length function is passed through the equation

$$l_i = \frac{1}{4}(l_{i-1} + 2l_i + l_{i+1}) \tag{18}$$

to promote smoothness of the mesh adaptation. Once the adaptation function has been smoothed, it is used to define a spring constant $k_{i+\frac{1}{2}} = 0.5(l_i + l_{i+1})$ that connects the



s^t_1 s^t_2 s^t_3 s^t_4 s^t_5 s^t_6 s^t_7 s^t_8 s^t_9
s
0
1
s^{t+1}_2 s^{t+1}_3 s^{t+1}_4 s^{t+1}_5 s^{t+1}_6 s^{t+1}_7 s^{t+1}_8

Fig. 3 s_i^t and s_i^{t+1} after mesh adaptation

s_i points. The relation of the k values that affects each s_i point is $k_{i-\frac{1}{2}}(s_i - s_{i-1}) = k_{i+\frac{1}{2}}(s_{i+1} + s_i)$. Points with higher arc length values will have higher spring constants and thus promote refinement in that region. The new positions are then solved for according to

$$s_i^{t+1} = \frac{k_{i-\frac{1}{2}} s^t_{i-1} + k_{i+\frac{1}{2}} s^t_{i+1}}{k_{i-\frac{1}{2}} + k_{i+\frac{1}{2}}}. \tag{19}$$

It has to be noted that the proposed nodal reallocation allows using the same differentiation matrix of the snake throughout the process of deformation because the number of snaxels remains constant. Also, this dynamic mesh adaptation approach makes relatively unnecessary the use of a large number of snaxels due to the better distribution of them over the boundary of a region of interest. Therefore, it is not necessary to use a large number of snaxels and as a consequence, the linear systems to be solved are relatively small and do not represent a high computational load.

Once the mesh adaptation is completed, the $\mathbb{R} \to \mathbb{R}^2$ mapping is computed. The slope values between each point of the contour $\mathbf{v}$ and the recently adapted mesh are used to perform this transformation. Refer to Fig. 3 where the mesh adaptation process is illustrated. The superindices t and $t+1$ are used to differentiate the old and new positions of the s_i points. Note that the s_i^t points correspond to the contour $\mathbf{v}$ of Fig. 2a.

The first step is to determine the interval where each s_i^{t+1} point is located, i.e. to find out between which s_i^t points is placed. After that, the corresponding (x, y) ordered pairs of these adjacent s_i^t points are determined. The abscissa and ordinate values can be represented as functions of s_i with their respective slope m_i. These slopes are used to compute the (x, y) values corresponding to the s_i^{t+1} points. Refer to Fig. 4 where the approximate value of the ordered pair (x, y) for each of the s_i^t points of Fig. 2a is presented as function of s. The slopes formed by the (x, y) values and the s_i^{t+1} points are shown too.

The abscissa and ordinate values corresponding to each s_i^{t+1} point are found by

$$x - x_i = m_{\text{abs}}(s^{t+1} - s_i^t), \quad y - y_i = m_{\text{ord}}(s^{t+1} - s_i^t) \tag{20}$$

where

$$m_{\text{abs}} = \frac{x(s^t_{i+1}) - x(s^t_i)}{s^t_{i+1} - s^t_i}, \quad m_{\text{ord}} = \frac{y(s^t_{i+1}) - y(s^t_i)}{s^t_{i+1} - s^t_i} \tag{21}$$

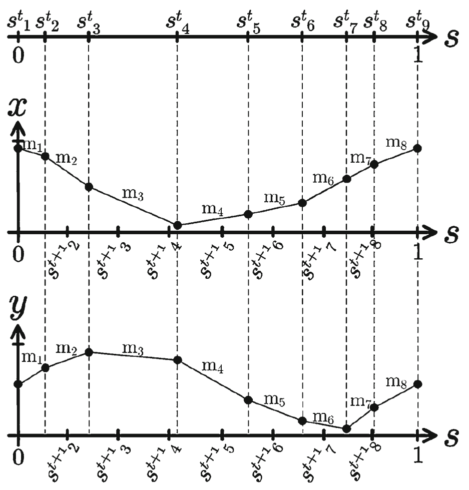


Fig. 4 Abscissa and ordinate values as functions of s

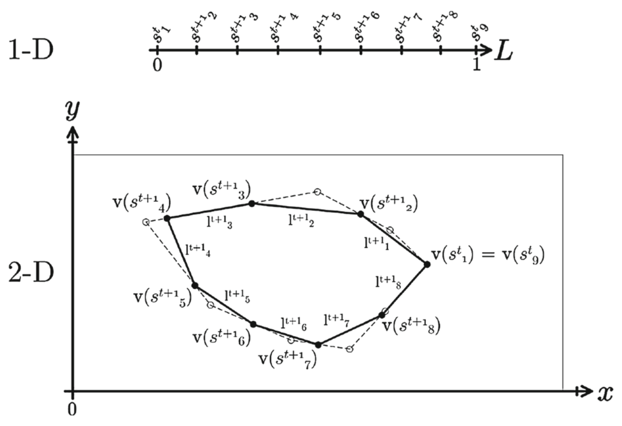


Fig. 5 Result for snaxel reallocation

and the sub-indexes i and $i + 1$ correspond to the left and right adjacent s_i^t points respectively.

The result of the $\mathbb{R} \to \mathbb{R}^2$ mapping described is shown in Fig. 5. It can be seen that the new positions of the snaxels are better distributed over the image than the originals (Fig. 2a) and therefore, represent a better approximation of the border of the region of interest within the image.

3 Graphic processing units

High-performance computing refers to the use of parallel processing platforms in the solution of computational intensive problems efficiently, reliably and quickly. Parallel platforms include clusters, multicore architecture and hybrid systems [31].

Recently, semiconductor industry has settled on two main design lines for microprocessors: the multicore processors and the manycore processors. By their definition, a multicore processor can have up to tens of cores, whereas a manycore processor has hundreds and even thousands of cores [32]. A CPU is an example of a multicore processor; GPUs are an example of a manycore processor.

The ratio between GPUs and multicore processors for peak floating-point operations (FLOPS) is about 10–1 [33]. The reason for this gap in their performance lies in their architecture: a CPU is a processor designed for sequential code execution whereas the GPU design is conceived to increase the execution throughput of parallel applications. The higher memory bandwidth of the GPU is also an important feature that contributes to the performance gap [32].

3.1 CUDA

CUDA is a parallel computing architecture for general purpose as well as a parallel programming model that offers high-level access to the GPU. CUDA allows the use of a GPU to solve computationally intensive problems in a more efficient way than with a CPU [34].

The CUDA programming model allows the transparent scalability of applications through GPUs with different number of cores. The base to get this transparent scalability are three key abstractions: a hierarchy of threads groups, shared memory and barrier synchronization. These abstractions allow the programmer to partition the problem into thread blocks that can be solved independently. The thread blocks can be considered as sub-problems and are executed in the available GPU cores, in any order and in parallel or sequential manner. This independence of the thread block execution is the characteristic that allows the scalability of the CUDA applications [34,35].

The CUDA programming model enables the use of a GPU as a co-processor of the CPU. In this context, the GPU is called device and the CPU is called host, as it is shown in Fig. 6. A CUDA program is composed of sequential code sections for the

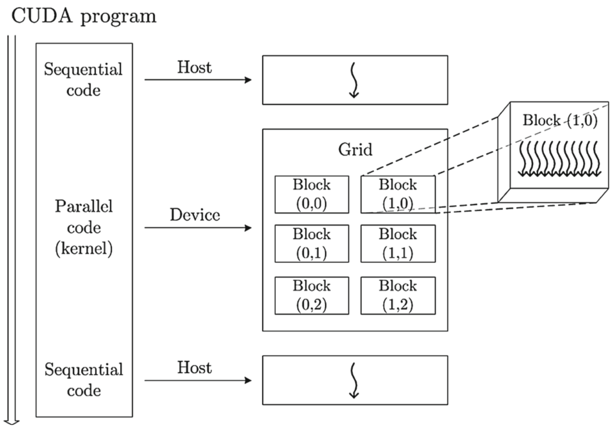


Fig. 6 CUDA heterogenous programming model

host and parallel code sections for the device. In the parallel code sections, thousands of threads are executed concurrently to reduce the computation time [32].

In a CUDA program the main thread is executed in the host, as it is shown in Fig. 6. When a kernel is called, the execution is moved from the host to the device where a massive number of threads are executed concurrently to perform the parallel operations. A kernel is a subroutine that is executed K times by K threads within the device [32]. The threads generated by a kernel are grouped in thread blocks, and the total of thread blocks within a kernel is called a grid. The kernel calls are asynchronous, which implies that after the invocation of a kernel, the host can execute the rest of the sequential code or just wait for the termination of the kernel execution [32].

3.2 Optimization strategies

The optimization of a CUDA program is based on three main aspects: maximize parallel execution, optimize memory usage to achieve maximum memory bandwidth, and optimize instruction usage to achieve maximum instruction throughput [34,36].

The most important rule to optimize the memory space usage of the GPU is to minimize the data transfer operations between the CPU and the GPU, because these memory operations have a lower memory bandwidth than the internal transfers within the GPU. It is also important to minimize the kernel access to the global memory and maximize the use of the shared memory. For a detailed description of many other optimization strategies see [36].

4 Parallel implementation of the GVF snake

The proposed scheme for the parallel implementation of the GVF snake is shown in Fig. 7. We implemented this scheme twice with the parallel frameworks OpenMP and CUDA to do a performance comparison between them. Our approach focuses on the

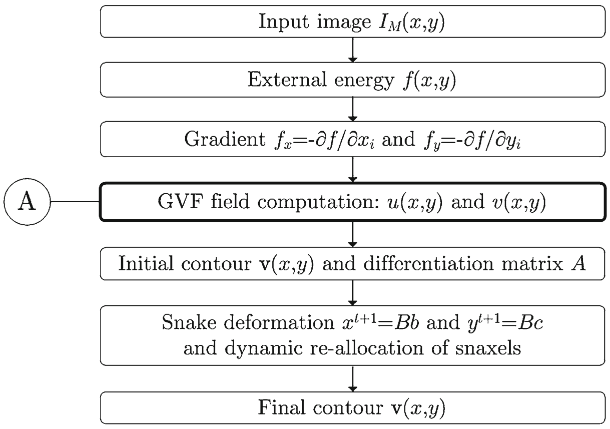

Input image $I_M(x,y)$
External energy $f(x,y)$
Gradient f_x=-$\partial f/\partial x_i$ and f_y=-$\partial f/\partial y_i$
A
GVF field computation: $u(x,y)$ and $v(x,y)$
Initial contour $\mathrm{v}(x,y)$ and differentiation matrix A
Snake deformation x^{t+1}=Bb and y^{t+1}=Bc and dynamic re-allocation of snaxels
Final contour $\mathrm{v}(x,y)$

Fig. 7 General scheme of GVF parallel computation

Algorithm 1 Pseudo-code for OpenMP implementation of GVF computation

```
#pragma omp parallel {
for 0 < $ITER$ do
  #pragma omp master {
    $u^{t+1} = \mu \Delta t \nabla^2 u^t - [u^t - fx^2][fx^2 + fy^2]$; No-flux boundary conditions
    $v^{t+1} = \mu \Delta t \nabla^2 v^t - [v^t - fy^2][fx^2 + fy^2]$;
  }
  #pragma omp for schedule(Dynamic, 16)
  for $m = 1 < N - 1$ do
    for $n = 1 < N - 1$ do
      $u^{t+1} = \mu \Delta t \nabla^2 u^t - [u^t - fx^2][fx^2 + fy^2]$;
      $v^{t+1} = \mu \Delta t \nabla^2 v^t - [v^t - fy^2][fx^2 + fy^2]$;
    end for
  end for
  #pragma omp master {
    $memcpy(u^t, u^{t+1})$;
    $memcpy(v^t, v^{t+1})$;
  }
end for
}
```

parallelization of the GVF field computation, which is denoted as the A-box of Fig. 7. Due to the dynamic reallocation feature proposed, the solution of Eqs. (8) and (9) is performed sequentially as only relatively small linear systems are used to compute the snake deformation. The other less demanding operations, e.g. image reading and writing, are executed sequentially too. The GVF field is computed via the Eqs. (14) and (15). The OpenMP and CUDA codes for this parallel section are detailed below.

4.1 OpenMP computation of the GVF field

For the case of the OpenMP implementation, the A-box of Fig. 7 represents a parallel *for*. Algorithm 1 contains the pseudo-code used.

We tested many configurations for the OpenMP implementation following the optimizations recommended in [37] and the structure presented in Algorithm 1 produced the best results in terms of overhead, CPU usage, thread concurrency and execution time. Table 3 comprises the timing value results for each of the images used.

The omp parallel directive is placed outside the main loop to reduce the overhead produced by the creation of parallel regions. The computation of the boundary values of the u and v components and also the update process of these components is performed sequentially by the master thread.

The omp for directive is used to make the computing of the u and v components in parallel. Each thread computes chunks of 16 loop iterations and the scheduling of these chunks is dynamic, i.e. the thread executes the chunk of iterations then requests another chunk until there are no more chunks to work on [37].

An OpenMP implementation represents a coarse-grain parallel approach, in which each thread computes the GVF field of hundreds of pixels sequentially, i.e. only few pixels are computed concurrently.

Algorithm 2 Pseudo-code for CUDA implementation of GVF computation using textures

```
for $0 < ITER$ do
  GVF<<< $blocksize, gridsize$ >>> $(u, v, \mu, \Delta t, N)$;
  $cudaMemcpyToArray(texu, u, cudaMemcpyDeviceToDevice)$;
  $cudaMemcpyToArray(texv, v, cudaMemcpyDeviceToDevice)$;
end for
normalization<<< $blocksize, gridsize$ >>> $(u, v, N)$;
__global__ GVF $(u, v, \mu, \Delta t, N)$ {
  $id = blockIdx.x * blockDim.x + threadIdx.x + blockIdx.y * blockDim.y * N + threadIdx.y * N$;
  $x = threadIdx.x + blockIdx.x * blockDim.x$;
  $y = threadIdx.y + blockIdx.y * blockDim.y$;
  $u[id] = texu_{x,y} + \mu * \Delta t * (texu_{x-1,y} + texu_{x+1,y} - 4 * texu_{x,y} + texu_{x,y-1} + texu_{x,y+1}) - ((texu_{x,y} - texfx_{x,y}) * texfxfy_{x,y})$;
  $v[id] = texv_{x,y} + mu * \Delta t * (texv_{x-1,y} + texv_{x+1,y} - 4 * texv_{x,y} + texv_{x,y-1} + texv_{x,y+1}) - ((texv_{x,y} - texfy_{x,y}) * texfxfy_{x,y})$;
}
__global__ normalization $(u, v, N)$ {
  $id = blockIdx.x * blockDim.x + threadIdx.x + blockIdx.y * blockDim.y * M + threadIdx.y * N$;
  $nz = 0.0000000001f$;
  $temp1 = u[id]$;
  $temp2 = v[id]$;
  $temp3 = sqrt(temp1 * temp1 + temp2 * temp2)$;
  $u[id] = temp1/(temp3 + nz)$;
  $v[id] = temp2/(temp3 + nz)$;
}
```

4.2 CUDA computation of the GVF field

For the CUDA implementation, the A-box in Fig. 7 represents a loop that contains kernel calls. The loop is controlled by the host whereas the kernels called within it are executed on the device. Algorithm 2 contains the pseudo-code used.

The computing of the GVF field with a GPU involves the solution of Eqs. (8) and (9) on each pixel of the image matrix $I_M(x, y)$ using a scheme of one pixel per thread. This approach of data mapping facilitates the pixel distribution into thread blocks, which is the base of the CUDA programming model. This way, unlike the sequential implementation of the GVF snake in which the execution thread must calculate the GVF field of one pixel at a time or the OpenMP implementation described in Sect. 4.1, the GPU implementation allows the parallel computation of the GVF field on thousands of pixels concurrently. Figure 8 shows the model of one pixel per thread for an image of 10×10 pixels. It is shown how the image is divided into blocks and the way each block is assigned a set of threads.

Because the equations for the components u and v of the GVF field involve access to data in a regular pattern (a central pixel and 4 of its neighbors), the necessary data are copied to the GPU memory as textures. Textures facilitate access to two-dimensional data, automatically handle the non-valid spatial locations and in some cases improve the computing time as a result of their cache. The kernels that are called to compute the GVF field (A-box of Fig. 7) are detailed in Algorithm 2. After the memory transfer from the host to device, the host starts a loop and the kernel **GVF**$<<< \cdots >>>$ is called once per iteration. This kernel computes the Eqs. (14) and (15). After completing

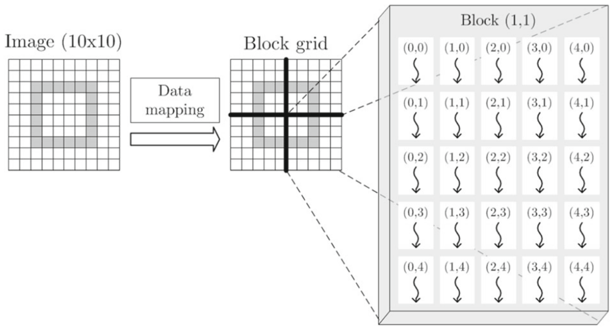


Fig. 8 One pixel per thread scheme

the specified number of iterations, the kernel **normalization**$<<< \cdots >>>$ is called to normalize the u and v components to the range $[-1, 1]$.

The Algorithm 3 presents the pseudo-code for the global memory version. To properly compute the Laplacian of the boundaries of the image, we used temporal arrays u_{temp} and v_{temp} with an increased size by two pixels on each axis, i.e. if the original image is 256×256 pixels in size, we set the temporal images as 258×258 pixels in size. The extra columns and rows are filled with zeros. To identify each thread correctly a padded thread ID is used. By utilizing this approach, the use of *if* statements is not required to avoid non-valid spatial conditions and this prevents thread divergence in the kernels. In Algorithm 3, the index *id* points to the original image locations. The index *ida* points to the temporal arrays of increased size.

4.3 Computation of the snake deformation and dynamic snaxel reallocation

After computing the GVF field, the initial contour for **v** and the differentiation matrix A are created.

The snake deformation is driven by the iterative solution of Eqs. (8) and (9) which can be represented as the linear systems $x^{t+1} = Bb$ and $y^{t+1} = Bc$ respectively where

$$B = \left(\frac{I}{\Delta t} + A\right)^{-1}, \quad b = \left(\kappa f_{x^t} + \frac{x^t}{\Delta t}\right), \quad c = \left(\kappa f_{y^t} + \frac{y^t}{\Delta t}\right), \tag{22}$$

I is the identity matrix and Δt is the time step.

The root mean square error (RMSE) is proposed as the stopping criterion for the deformation of the snake. The equation used is

$$\text{RMSE} = \frac{\sqrt{\sum_{i=1}^{k}(\mathbf{v}_i^{t+1} - \mathbf{v}_i^t)^2}}{k} \tag{23}$$

where k is the number of snaxels.

Algorithm 3 Pseudo-code for CUDA implementation of GVF computation using global memory

```
for $0 < ITER$ do
  boundaries$<<< blocksize, gridsize >>> (u, v, u_{temp}, v_{temp}, N)$;
  GVF$<<< blocksize, gridsize >>> (u, v, u_{temp}, v_{temp}, fx, fy, fxfy, \mu, \Delta t, N)$;
end for
normalization$<<< blocksize, gridsize >>> (u, v, N)$;
__global__ boundaries $(u, v, u_{temp}, v_{temp}, N)$ {
  $id = blockIdx.x * blockDim.x + threadIdx.x + blockIdx.y * blockDim.y * N + threadIdx.y * N$;
  $newsize = N + 2$;
  $ida = newsize * blockDim.y * blockIdx.y + newsize * threadIdx.y + blockDim.x * blockIdx.x + (threadIdx.x + 1) + newsize$;
  $u_{temp}[ida] = u[id]$;
  $v_{temp}[ida] = v[id]$;
}
__global__ GVF $(u, v, u_{temp}, v_{temp}, fx, fy, fxfy, \mu, \Delta t, N)$ {
  $id = blockIdx.x * blockDim.x + threadIdx.x + blockIdx.y * blockDim.y * N + threadIdx.y * N$;
  $newsize = N + 2$;
  $ida = newsize * blockDim.y * blockIdx.y + newsize * threadIdx.y + blockDim.x * blockIdx.x + (threadIdx.x + 1) + newsize$;
  $u[id] = u_{temp}[id] + \mu * \Delta t * (u_{temp}[ida - 1] + u_{temp}[ida + 1] - 4 * u_{temp}[id] + u_{temp}[ida - newsize] + u_{temp}[ida + newsize]) - ((u[id] - fx[id]) * fxfy[id])$;
  $v[id] = v_{temp}[id] + \mu * \Delta t * (v_{temp}[ida - 1] + v_{temp}[ida + 1] - 4 * v_{temp}[id] + v_{temp}[ida - newsize] + v_{temp}[ida + newsize]) - ((v[id] - fy[id]) * fxfy[id])$;
}
__global__ normalization $(u, v, N)$ {
  $id = blockIdx.x * blockDim.x + threadIdx.x + blockIdx.y * blockDim.y * M + threadIdx.y * N$;
  $nz = 0.0000000001f$;
  $temp1 = u[id]$;
  $temp2 = v[id]$;
  $temp3 = sqrt(temp1 * temp1 + temp2 * temp2)$;
  $u[id] = temp1/(temp3 + nz)$;
  $v[id] = temp2/(temp3 + nz)$;
}
```

To ensure that the snake has found the minimum energy region, three results of consecutive values of the RMSE are stored. The snake deformation loop is stopped when

$$\mathrm{RMSE}_{\mathrm{thresh}} > \mathrm{RMSE}_1 > \mathrm{RMSE}_2 > \mathrm{RMSE}_3, \tag{24}$$

where $\mathrm{RMSE}_{\mathrm{thresh}}$ is a predetermined value. The RMSE is computed before the snaxels reallocation process.

While the stopping criterion is not met, the snaxels reallocation process is performed. After each solution of Eq. (22), (18) and (19) are used to perform the mesh adaptation over the s domain and then, the reallocation of the snaxels is computed via the Eqs. (20) and (21). The new vectors x and y are then used in Eq. (22) to compute the next solution of the snake deformation. Although in all the experiments presented in this work the mesh adaptation process is computed just once after each solution of Eq. (22), it can be implemented iteratively to enhance the snaxel distribution.

5 Results

In this section, we present the results of our investigation. First, we discuss the performance of our snaxel reallocation approach by means of a synthetic test image. Then, we present the results of our GVF snake parallel implementations.

5.1 Snaxel reallocation implementation

Figure 9a shows the binary synthetic image that we used to test our snaxels reallocation method. The image presents a rounded star shape defined by the contrast difference. The star features five concave regions and is symmetric about the y-axis. The star image is 8 bits per pixel and 256×256 pixels in size.

In this part of the investigation we implemented the algorithm described in Fig. 7 sequentially because the computation performance is not the subject under analysis. The initial contour is a circle with center at [128, 135] and radius $r = 118$. This initial snake configuration is the same for the three implementations presented in this section.

Figure 9b shows the result of plain segmentation with the active contour. The snake parameters used are $\alpha = 0.2$, $\beta = 0$, $\kappa = 1$, 3,000 iterations for the GVF computation,

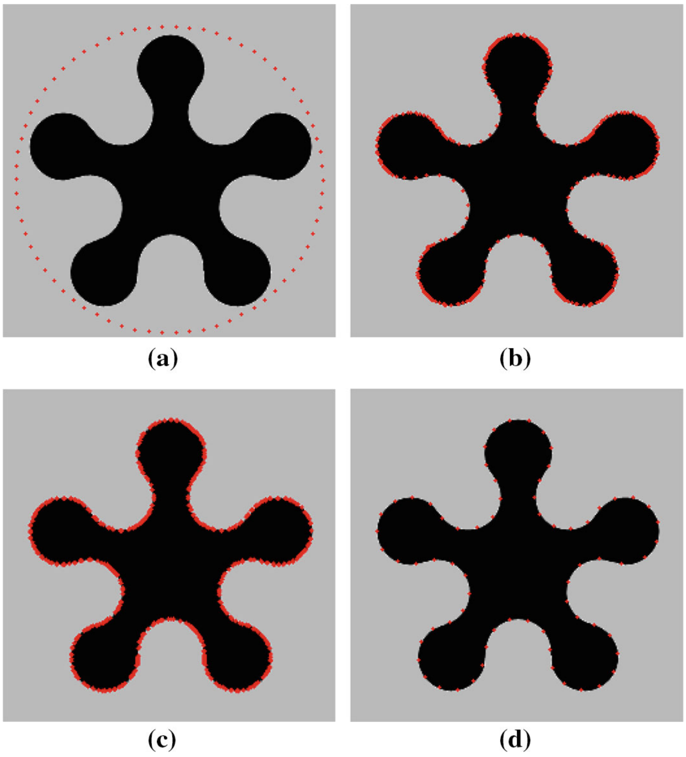


Fig. 9 **a** Initial contour, **b** plain segmentation, **c** segmentation with static refinement and **d** segmentation with the proposed snaxel reallocation method

$\mu = 0.15$ and N(number of snaxels) $= 1{,}300$. The active contour was indeed able to converge inside the concave regions after 2,000 iterations of the deformation Eq. (22) but the snaxels distribution over the edge of interest presents spacing problems as there are regions with clustering and some other regions where the snaxels are separated from each other. This behavior is the reason why a large number of snaxels is needed in order to let the snake converge in the concave regions. The large number of iterations required for the GVF field is another important issue for this particular case.

The final active contour shape for the case of static refinement of snaxels is shown in Fig. 9c. We used $\alpha = 0.01$, $\beta = 0$ and $\kappa = 0.6$, 70 iterations for the GVF computation, $\mu = 0.2$, $N_{\text{ini}} = 50$ and $N_{\text{fin}} = 605$. In fact, with this approach the active contour deformation process starts with a given N value (N_{ini}) but ends with a different number of snaxels (N_{fin}). The deformation process took 2,000 iterations of Eq. (22). The arc length between snaxels is the criterion for the refinement process, i.e. if the snaxels tend to cluster some of them are removed, if the snaxels tend to separate from each other a given number of snaxels is added. It can be seen that the snake was able to converge inside the concave regions but with a high computational cost, because the deformation matrix A needs to be calculated every time the number of snaxels is changed. This is an important drawback of this refinement approach as it directly affects the computational load of the algorithm.

Figure 9d shows the final snake configuration for our proposed snaxels reallocation approach. The snaxels are placed almost symmetrically about the y-axis and the active contour was able to effectively converge inside the concave regions. The settings used are $N = 70$, $\alpha = 0.01$, $\beta = 0$, $\kappa = 0.6$, 70 iterations for the GVF field and $\mu = 0.2$.

Considering the segmentation results, the number of snaxels used and the iterations for the GVF field, we can state that our reallocation approach is more efficient than the other two methods. The advantage is clear not only in the segmentation results but also in a performance computational point of view because with the proposed mesh adaptation method we obtain better results with fewer snaxels or GVF iterations and these conditions directly impact the execution time.

It has to be mentioned that, if needed, our approach can be parallelized as there are no data dependencies in the computing of the mesh adaptation. Since only a relatively small number of snaxels is utilized in the experiments presented in this work, we estimate that the parallel implementation was not necessary.

5.2 GVF snake parallel implementation

We used MRIs, PET/MRIs and CTs medical images only for illustrative purposes to show the performance of active contours. In order to analyze the efficiency of our algorithms, we have used six different sizes of images within the range of 64×64 to $2{,}048 \times 2{,}048$ pixels in size. The implementation time for the sequential version of each experiment is presented for reference. All the timing values presented are result of optimized versions of our algorithms and an average of six runs.

The experiments have been conducted under Ubuntu 13.04 using C language on a computer with an *intel i7-930* processor at 2.8 GHz and 6 GBytes of RAM memory. We disabled the Hyperthreading and Turboboost features to reduce variations between runs of the programs. The CUDA implementation is made with a Kepler *GeForce GTX* 670

GPU that has 2 GBytes of global memory. All the results presented are single-precision floating point. The OpenCV library is used to perform the image reading, writing and Canny edge detection operations. The same initial contour configuration is used for the three implementations of each experiment. Depending on the region of interest, we used ellipses or circumferences as the initial contour. All the medical images used in the experiments are 8 bits per pixel. The contour map for all the experiments is created with the Canny edge detector [29]. The final segmentation is equal for the sequential and parallel implementations, so only one image is shown.

We used the Intel Inspector and Intel Vtune Amplifier software to evaluate our OpenMP algorithms. Intel Inspector software can be used to locate and fix memory-related problems, e.g. data races and deadlocks presented in shared memory platforms; Intel Vtune Amplifier is used to determine the thread concurrency level, CPU usage, FLOPS and many other parameters. Therefore, when used together, the Amplifier and Inspector software provide a complete profile of an OpenMP application. To profile and debug the CUDA algorithms the Nvidia Visual Profiler and nvprof are used.

To compute the GVF field on the GPU we used blocks of 16×16 or 32×32 threads depending on the image size and these block sizes produce respectively 256 and 1,024 CUDA threads. For the OpenMP implementation we used four threads, i.e. one thread per physical core.

Figure 10a, b shows *Image1* which is a binary synthetic image that is 64×64 pixels in size. *Image2* is a 128×128 pixel-size positron emission tomography/magnetic resonance image that is showed in Fig. 10c, d [38]. The image highlights a brain lesion. Moving one step forward, *Image3* is a magnetic resonance image which is 256×256 pixels in size and it is shown in Fig. 10e, f [39]. *Image3* shows a section of the axial plane of the brain. Our fourth experiment is conducted with *Image4* which is a 512×512 pixels in size computed tomography shown in Fig. 11a, b [39]. The image presents a section of the sagittal plane of a knee. Figure 11c, d shows *Image5* which is a computed tomography that is $1{,}024 \times 1{,}024$ pixels in size [40]. This CT shows a section of the axial plane of a colon. Figure 11e, f shows a computed tomography (*Image6*) that we have used to perform our last experiment. The image is $2,048 \times 2,048$ pixels in size and shows a section of the axial plane of the brain [40]. For each experiment, the initial active contour and its final shape are shown. The GVF field parameter μ, the settings for the initial contour and the parameters of the snake for all the experiments presented are summarized in Table 1.

First, we present in Table 2 the comparison of timing values between the texture and global memory implementations of our algorithm. From these results, it is clear that the texture implementation is faster than the one with global memory, but the speedup is not as high as it could be with older GPU architectures. This is due of the L1 and L2 caches that feature the GPU but in addition, because the texture cache in the Kepler architecture is now available for general load operations as a read-only memory without using texture units. These results agree with the conclusions presented in [13]. Based on this timing results, we use the texture implementation in the performance comparison against the openMP version of the GVF computing.

The number of iterations for the GVF field, the best computation times for the sequential, OpenMP and CUDA implementations and the ratio between the OpenMP and CUDA timing values of all the experiments are shown in Table 3.

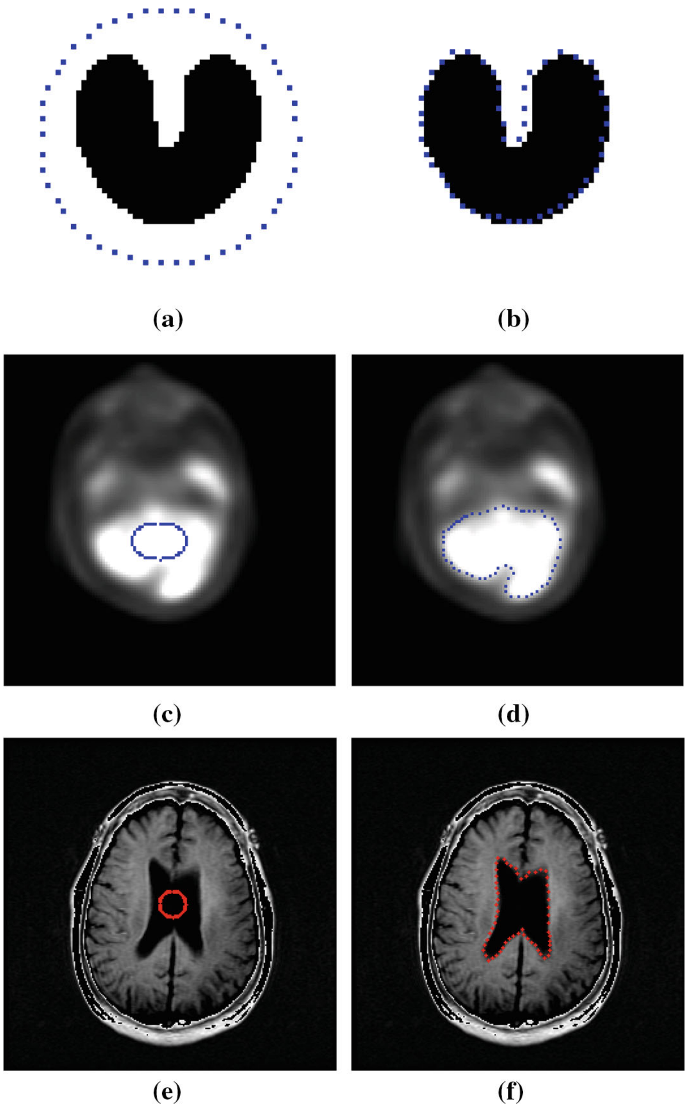


Fig. 10 **a** Initial and **b** final contour for synthetic image *Image1*; **c** initial and **d** final contour for PET/MRI *Image2*; **e** initial and **f** final contour for MRI *Image3*

In order to make a more objective comparison and better evaluation between the implementations, we have taken into account three key aspects: (1) execution time, (2) occupancy and (3) FLOPS. Table 4 presents these results for the OpenMP and CUDA implementations. We used the occupancy as a metric because the GVF computation is memory-bound and for this type of algorithm, we required to increase the occupancy to hide latencies. The FLOPS are used to roughly estimate the computational peak of the algorithms.

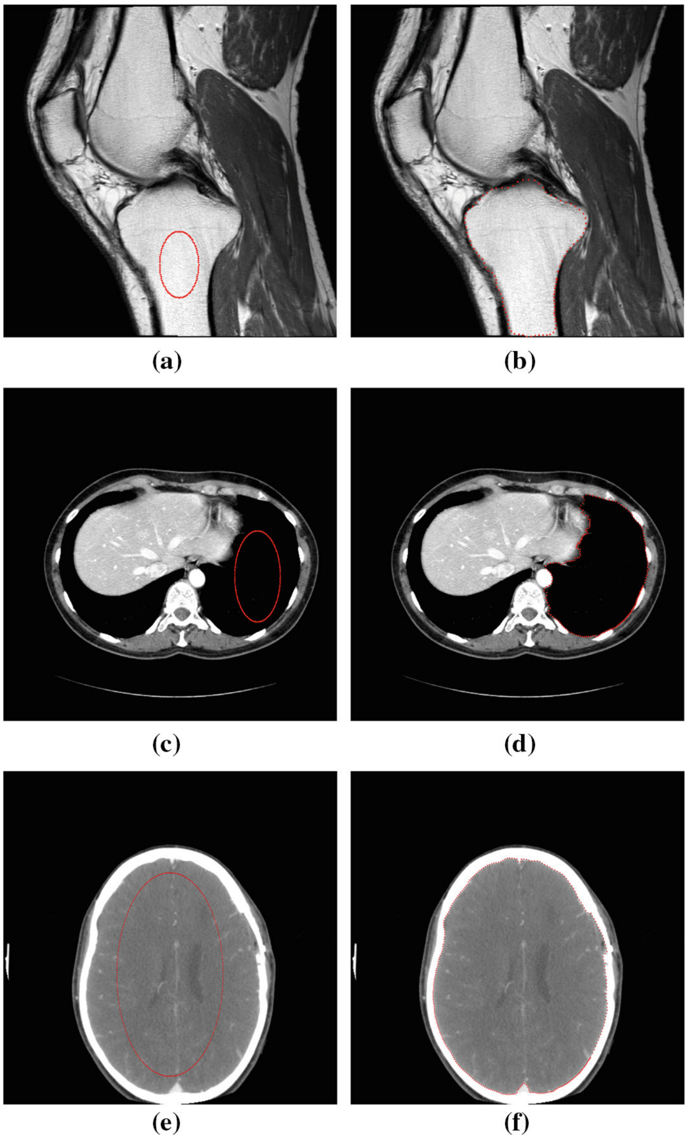


Fig. 11 **a** Initial and **b** final contour for CT *Image4*; **c** initial and **d** final contour for CT *Image5*; **e** initial and **f** final contour for CT *Image6*

For the GPU, occupancy is defined as the ratio between active warps and the maximum number of warps. For the OpenMP implementation, we defined occupancy as the ratio between the thread concurrency (number of active threads) and the number of physical cores, which in our particular case is four.

Besides the metrics presented in Table 4, there are many other parameters that can be considered in the evaluation of an application but not all of them represent a good aspect to take into account because all the problems have specific issues. However,

Table 1 GVF μ value, settings for initial contours and snake parameters

Image	μ	Snaxels	Radius	Center	α	β	κ
Image1	0.2	50	20	[32, 32]	0.01	0	0.6
Image2	0.2	60	7	[60, 72]	0.01	0	0.6
Image3	0.2	60	10	[130, 128]	0.01	0	0.6
Image4	0.2	100	30	[270, 400]	0.21	0	0.6
Image5	0.2	200	70	[780, 580]	0.01	0	0.6
Image6	0.15	300	250	[1,024, 1,250]	0.01	0	0.6

Table 2 Timing values for global memory and texture computing of GVF field

Image	Global memory (s)	Texture (s)
Image1	0.00119	0.00125
Image2	0.00161	0.00166
Image3	0.00435	0.00392
Image4	0.04275	0.04009
Image5	0.22794	0.22602
Image6	5.25097	5.12791

Table 3 Execution time comparison of GVF field computing between the sequential, OpenMP and CUDA implementations

Image	GVF Iter.	CPU (s)	OpenMP (s)	CUDA (s)	Ratio
Image1	50	0.00083	0.00203	0.00119	1.70
Image2	50	0.00368	0.00465	0.00161	2.88
Image3	80	0.0162	0.01155	0.00435	2.65
Image4	300	0.25015	0.12945	0.04275	3.02
Image5	500	2.57215	1.68421	0.22794	7.38
Image6	3,000	59.4689	43.75361	5.25097	8.33

Table 4 Metrics of GVF field computing for the OpenMP and CUDA implementations

Image	Platform	Time (s)	Occupancy (%)	GFLOPS
Image1	OpenMP	0.00203	76.25	0.166
	CUDA	0.00119	47.10	14.960
Image2	OpenMP	0.00465	78.32	0.615
	CUDA	0.00161	79.30	35.350
Image3	OpenMP	0.01155	78.32	2.741
	CUDA	0.00435	82.10	71.900
Image4	OpenMP	0.12945	79.10	7.150
	CUDA	0.04275	85.40	71.180
Image5	OpenMP	1.68421	95.40	7.540
	CUDA	0.22794	85.70	88.650
Image6	OpenMP	43.75361	92.65	8.290
	CUDA	5.25097	86.70	89.740

the execution time is the most representative metric to evaluate a parallel application and, in addition with other selected metrics, gives a complete performance profile of an algorithm.

6 Conclusions

The proposed snaxel reallocation method improves the efficiency of the segmentation process with the GVF snake. From a HPC point of view, it improves the performance of the snake deformation computing by significantly reducing the number of snaxels required to perform the segmentation of a given region of interest. This condition has a direct impact on the computation time because the deformation of the snake can be driven with the use of small linear systems. Furthermore, the proposed reallocation technique in some cases reduces the necessity of a large number of iterations for the GVF field, decreasing the overall computation time of the segmentation.

Our implementation provides better results if it is used on images that have been processed with noise reduction and enhancement techniques, and present high contrast. If it is used in images without these characteristics, there is a probability that the active contour will not converge to the region of interest due to the many local energy minima.

We present an objective comparison between the parallel implementation of our algorithm on a CPU and on a GPU. OpenMP is used to implement the CPU parallel version. Although the elapsed time of an application is not the best way to evaluate a CUDA algorithm, it is indeed the most important aspect. That is why it is the most common metric for assessing a program. However, execution time alone is not the best choice because is not objective, fair nor qualitative. Many factors are involved in a speedup such as the capacity of the processors or available resources, among others. That is why we think that an execution time comparison plus an objective measurement of the GPU and CPU resources utilization is the best way to evaluate a GPU application. For this reason, we used three metrics for the evaluation of our CUDA application: execution time, occupancy and FLOPS.

The results encountered in this work show three important aspects. First, the OpenMP version of the GVF field computing is in general just slightly faster than the sequential version of the algorithm. This somehow unexpected result can be explained by the auto-vectorization feature of the icpc compiler which reduces the computational load of loops using SSE packed (vector) instructions rather than scalar instructions. This process can be seen as a type of loop parallelization. Due the characteristics of the GVF field computing loop, the auto-vectorization feature greatly reduces the sequential computation time. Second, in our analysis of textures and global memory performance for the Kepler architecture, we did find out that the GVF computing using textures is faster than the global memory version. However, the difference is not as significant as it could be in older GPU architectures. Kepler texture objects were not used in this work because they are designed for applications where a large number of textures is required and the binding/unbinding overhead of texture references represents a time execution problem. Third, the difference in the execution time of the OpenMP and CUDA implementations coincides with the theoretical gap between the computational capacity of a CPU and a GPU. This result indirectly indicates that our implementations are correct considering that speedups of orders of magnitude are not consistent with the peak capacity of the processors.

Acknowledgments This work has been partially supported by COFAA-IPN, and by Grant IPN-SIP-20120606.

References

1. Dhawan AP, Huang HK, Kim DS (2008) Principles and advanced methods in medical imaging and image analysis. World Scientific Publishing Co. Pte, Ltd, Singapore
2. González RC, Woods RE (2002) Digital image processing, 2nd edn. Prentice Hall, Englewood Cliffs
3. He L, Peng Z, Everding B, Wang X, Han CY, Weiss KL, Wee WG (2008) A comparative study of deformable contour methods on medical image segmentation. Image Vis Comput 26(2):141–163
4. Li B, Acton ST (2007) Active contour external force using vector field convolution for image segmentation. IEEE Trans Image Process 16(8):2096–2106
5. Wang Y, Liu L, Zhang H, Cao Z, Lu S (2010) Image segmentation using active contours with normally biased GVF external force. IEEE Signal Process Lett 17(10):875–878
6. Terzopoulos D, McInerney T (1996) Deformable models in medical image analysis: a survey. Med Image Anal 1(2):91–108
7. Suri JS, Farag AA (eds) (2007) Deformable models II: theory and biomaterial applications. Springer, Berlin
8. Mahmoud MKA, Al-Jumaily A (2011) Segmentation of skin cancer images based on gradient vector flow (GVF) snake. In: IEEE international conference on mechatronics and automation. Beijing, China, pp 216–220
9. He Z, Kuester F (2006) GPU-based active contour segmentation using gradient vector flow. In: Advances in visual computing second international symposium, ISVC 2006. Lake Tahoe, NV, USA, pp 191–201
10. Zheng Z, Zhang R (2011) A GPU-accelerated GVF snake algorithm. In: Proceedings of the 2011 workshop on digital media and digital content management, DMDCM '11. Hangzhou, China, pp 233–236
11. Perrot G, Domas S, Couturier R, Bertaux N (2011) GPU implementation of a region based algorithm for large images segmentation. In: 11th IEEE international conference on computer and information technology. Belfort, France, pp 291–298
12. Li T, Krupa A, Collewet C (2011) A robust parametric active contour based on Fourier descriptors. In: 18th IEEE international conference on image processing. Brussels, Belguim, pp 1037–1040
13. Smistad E, Elster AC, Lindseth F (2012) Real-time gradient vector flow on GPUs using OpenCL. J Real Time Image Process. doi:10.1007/s11554-012-0257-6
14. Kienel E, Brunnett G (2009) GPU-accelerated contour extraction on large images using snakes. Technical Report. CSR-09-02, Chemnitz University of Technology, Germany
15. Češnovar R, Risojević V, Babić Z, Dobravec T, Bulić P (2013) A GPU implementation of a structural-similarity-based aerial-image classification. J Supercomput 65:978–996
16. Valero P, Sánchez JL, Cazorla D, Arias E (2011) A GPU-based implementation of the MRF algorithm in ITK package. J Supercomput 58:403–410
17. Reyes R, López I, Fumero JJ, de Sande F (2013) A preliminary evaluation of OpenACC implementations. J Supercomput 65:1063–1075
18. Lenkiewicz P, Pereira M, Freire MM, Fernandes J (2009) A new 3D image segmentation method for parallel architectures. In: IEEE International conference on multimedia and expo, 2009. New York, USA, pp 1813–1816
19. Schellmann M, Gorlatch S, Meiländer D, Kösters T, Schäfers K, Wübbeling F, Burger M (2011) Parallel medical image reconstruction: from graphics processing units (GPU) to grids. J Supercomput 57:151–160
20. Pallipuram VK, Bhuiyan M, Smith MC (2012) A comparative study of GPU programming models and architectures using neural networks. J Supercomput 61:673–718
21. Belloch JA, González A, Martínez-Saldívar F, Vidal AM (2011) Real-time massive convolution for audio applications on GPU. J Supercomput 58:449–457

22. Xu C, Prince JL (1998) Snakes, shapes, and gradient vector flow. IEEE Trans Image Process 7(3):359–369
23. Terzopoulos D (1986) On matching deformable models to images. Technical Repprt 60, Schlumberger Palo Alto Research, USA
24. Kass M, Witkin A, Terzopoulos D (1988) Snakes: active contour models. Int J Comput Vis 1(4):321–331
25. Terzopoulos D, Platt J, Barr A, Fleischer K (1987) Elastically deformable models. SIGGRAPH 21(4):205–214
26. Davatzikosa C, Prince JL (1996) Convexity analysis of active contour problems. In: IEEE conference on computer vision and pattern recognition, CVPR '96. San Francisco, USA, pp 674–679
27. Mishra AK, Fieguth PW, Clausi DA (2011) Decoupled active contour (DAC) for boundary detection. IEEE Trans Pattern Anal Mach Intell 33(2):310–324
28. Boukerroui D (2009) Efficient numerical schemes for gradient vector flow. In: 16th IEEE international conference on image processing (ICIP). Cairo, Egypt, pp 4057–4060
29. Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell 8(6):679–698
30. Huang S, Wang B, Huang X (2006) Using GVF snake to segment liver from CT images. In: 3rd IEEE/EMBS International Summer School on Medical Devices and Biosensors, 2006. Cambridge, USA, pp 145–148
31. Wagner S, Steinmetz M, Bode A, Muller M (2010) High performance computing in science and engineering. Springer, Berlin
32. Kirk DB, Wen-mei WH (2010) In Praise of Programming massively parallel processors: a hands-on approach. Elsevier, Amsterdam
33. Cook S (2013) CUDA programming: a developer's guide to parallel computing with GPUs. Elsevier, Amsterdam
34. nVIDIA (2013) NVIDIA CUDA C programming guide
35. Farber R (2011) CUDA application design and development. Elsevier Inc., Amsterdam
36. nVIDIA (2013) CUDA C best practices guide
37. Chapman B, Jost G, van der Pas R (2008) Using OpenMP: portable shared memory parallel programming. The MIT press, Cambridge
38. The cancer imaging archive (2012) http://www.cancerimagingarchive.net/. Accessed 15 Jan 2012
39. Patient contributed image repository (2012) http://www.pcir.org/. Accessed 15 Jan 2012
40. Dicom sample image sets (2013) http://www.osirix-viewer.com/datasets/. Accessed 15 Jun 2013