Centro Singular de Investigación en Tecnoloxías da Información

GPU-accelerated Level-Set Segmentation

Julián Lamas-Rodríguez
Centro Singular de Investigación en Tecnoloxías da Información
julian.lamas@usc.es
http://citius.usc.es

Dora B. Heras
Centro Singular de Investigación en Tecnoloxías da Información
dora.blanco@usc.es
http://citius.usc.es

Francisco Argüello
Departamento de Electrónica e Computación, Universidade de Santiago de Compostela
francisco.arguello@usc.es

Dagmar Kainmueller, Stefan Zachow
Department of Visualization and Data Analysis, Konrad-Zuse-Zentrum für Informationstechnik Berlin
kainmueller@zib.de, zachow@zib.de

Montserrat Bóo
Departamento de Electrónica e Computación, Universidade de Santiago de Compostela

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, and FEDER funds under contract TIN 2010-17541, and by the Xunta de Galicia under contracts 08TIC001206PR and 2010/28. Julián Lamas-Rodríguez acknowledges financial support from the Ministry of Science and Innovation, Government of Spain, under a MICINN-FPI grant.

Abstract

The level-set method, a technique for the computation of evolving interfaces, is a solution commonly used to segment images and volumes in medical applications. GPUs have become a commodity hardware with hundreds of cores that can execute thousands of threads in parallel, and they are nowadays ideal platforms to execute computational intensive tasks, such as the 3D level-set–based segmentation, in real time.

In this paper we propose two GPU implementations of the level-set–based segmentation method called Fast Two-Cycle. Our proposals perform computations in independent domains called tiles and modify the structure of the original algorithm to better exploit the features of the GPU. The implementations were tested with real images of brain vessels and a synthetic MRI image of the brain. Results show that they execute faster than a CPU-sequential implementation of the same method, without any significant loss of the segmentation quality and without requiring distributed parallel computer infrastructures.

1 Introduction

The level-set method [31] is a numerical technique for analyzing and computing interface motion. Propagating interfaces occur in a wide variety of settings, and level-set methods have a high number of applications, including physics, chemistry, fluid mechanics, combustion, materials sciences, fabrication of microelectronic components, computer vision, and image processing [36].

A key task in computer vision, medical visualization and medical image processing is the identification of regions of interest using, for example, a segmentation process. Surgical planning, navigation, simulation, diagnosis, and therapy evaluation all benefit from the segmentation of anatomical structures, based on the properties of the images, such as the observed intensities, as well as anatomical knowledge on the subjects [21]. The use of level-set methods for image and volume segmentation has been demonstrated as an effective technique [43]. The level-set segmentation process depends upon extrinsic factors (e.g., the intensities or the texture of the image) and intrinsic factors (e.g., the curvature of the segmented region) [34].

The graphics processing unit (GPU) has evolved from a graphics-specific accelerator, with a fixed-function graphics pipeline, into a programmable vector processor with computing power exceeding that of a multicore CPU [23]. Nowadays, GPUs are high-performance many-core processors capable of very high computation and data throughput. GPUs have become general-purpose parallel processors with support for accessible application programming interfaces (APIs) and industry-standard languages, which has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU [32], developing the field of GPGPU (General Purpose Computing on the GPU) [5]. Recent years have seen the offspring of GPU-specific frameworks such as CUDA from NVIDIA [7], BrookGPU from the Stanford University Graphics Lab [2], and OpenCL from the Khronos group [6]. These solutions have leveraged the implementation of high performance computing tasks on the GPU. CUDA is both a software development kit (SDK) and an API that enables the C and C++ programming languages to be used to code algorithms for execution on NVIDIA GPUs. BrookGPU is a compiler and runtime

implementation of the Brook stream program language, a variant of C, for modern graphics hardware. OpenCL is an open standard that provides cross-platform GPGPU capabilities.

In this paper we propose two alternative GPU implementations of the *Fast Two-Cycle* (FTC) level-set segmentation algorithm [37]. The FTC method uses integer-only operations to update the level-set, making it quite appealing for its adaptation from CPU to GPU, where integer computations can achieve excellent performance. Our objective is to accelerate the segmentation of medical images, attempting to achieve real-time execution with no significant loss of quality. We have tested our implementations using a synthetic MRI image of the brain as well as clinical tomographic data of the brain.

The sequential implementation of the FTC method relies heavily on the use of linked lists to track the active domain (i.e., the positions on the level-set field that might be updated). However, linked lists cannot be efficiently implemented on GPU. Instead, our implementations modify the structure of the original algorithm to exploit the features of the graphical hardware.

First, the computational domain is partitioned into 3D fixed-size tiles that can be stored in the shared memory of the streaming multiprocessors and can be efficiently processed by the thread blocks. This partitioning requires the efficient handling of the transition between neighboring tiles.

Second, the iterative structure of the original method is modified, adapting the number of iterations in the algorithm to the features of the GPU, so the streaming multiprocessors can compute several iterations on the tiles. This allows the exploitation of the shared memory, which is faster than the global memory.

Third, a list of active tiles is maintained to compute only those tiles where a portion of the front is present. Our second implementation restricts this list to those tiles where the front is still unstable.

The rest of the paper is organized as follows. Section 2 briefly introduces previous works on GPU-based level-set solvers. Section 3 describes the CUDA architecture. Section 4 presents the fundamentals of level-set segmentation, and, more specifically, the FTC method. Section 5 examines our two proposals of the CUDA-based FTC method. Section 6 analyzes the experimental results, and provides a comparison to recent works on GPU-based level-set segmentation. Finally, Section 7 concludes discussing the results and main contributions.

2 Related Work

Level-set solver algorithms are usually classified into two different groups, known as *narrow-band* and *sparse-field* methods [22]. In order to avoid a level-set computation within the entire image volume, acceleration is achieved by restricting the computation to relevant regions of interest. The narrow-band approach [11] updates only a small region of the image (typically a band of a few voxels) surrounding the front. On the other hand, the sparse-field approach [44, 33] keeps a list of active data elements in the whole domain, which is updated after each iteration of the algorithm. The FTC level-set segmentation algorithm [37] is a sparse-field method that uses two linked lists with the coordinates of the surface that encloses the level-set volume.

Besides improving the speed of level-set computation, memory usage has been another key issue in the design of level-set methods. Quadtree and octree-based methods, using

adaptive meshes that store the values of the level-set function in their vertices, were developed in [39, 40, 25]. The sparse block grid method, which divides the domain in fixed-sized tiles and stores only those traversed by the front, was introduced in [12]. A similar approach was also used to develop the Sorted Tile List method [42], which further reduces memory requirements.

The first GPU implementation of a level-set method was proposed in [35]. This implementation performed the calculations required to evolve the level-set field by executing blending operations in the back framebuffer of the GPU. Borrowing ideas from the narrow-band and sparse-field algorithms, a memory-adaptive approach dividing the domain in 2D tiles that are loaded into the GPU on demand was introduced in [22]. Based on previous work, a CUDA sparse-field solution, which largely reduced the number of processed elements and the required computational time by keeping a list of active elements in the whole domain, was proposed in [34]. At the same time, a CUDA narrow-band method was developed in [18] keeping a list of active tiles that is traversed and updated in parallel (regretfully, this solution requires the use of GPU atomic operations, which degrades performance). More recently, a GPU implementation of the Sorted Tile List method was described in [17], and an OpenCL-based solution for multi-object level-set segmentation was introduced in [24].

To the best of our knowledge, the only existing GPU implementation of the FTC method is described in [41]. However, it is focused mainly in 2D datasets and analyzes only a single step of one of the cycles of the algorithm. Our solution has been designed for 3D datasets and implements the complete FTC method.

3 GPU architecture

A programmable GPU consists of several many-core processors capable of running hundreds of threads concurrently. Modern GPUs offer a high computational power and, in addition, a very high memory bandwidth. In this section we present a brief review of the Fermi GPU architecture employed in our implementations. An extended review can be found in [26].

The NVIDIA's CUDA architecture is organized into a set of streaming multiprocessors (SMs), each with many streaming processors (SPs), also known as CUDA cores [29]. The number of cores depends on the device model; e.g., the GeForce GTX 580 features 512 processor cores grouped into 16 SMs of 32 SPs each. These cores can manage hundreds of threads in a single program multiple data (SPMD) programming model. Figure 1 shows a schematic diagram of this architecture.

In the CUDA computational model, programmers run thousands of *threads* that are grouped into independent *thread blocks*, which in turn are grouped into a *grid* that conforms a virtual architecture where the CUDA program is executed. Every piece of code that is executed on the GPU is called a *kernel*. The programmer can freely configure the number of threads per block and the number of blocks in the grid that a kernel is going to launch, always within the limits of the architecture. Although this programming model is oriented towards a fine-grained level of parallelism [19], it also can be used to implement solutions with a coarser level. In any case, the workflow must always be mapped to thread blocks that operate independently.

When the kernel is executed, thread blocks are assigned to multiprocessors and executed in parallel. A multiprocessor can run more than one thread block at a time depending on

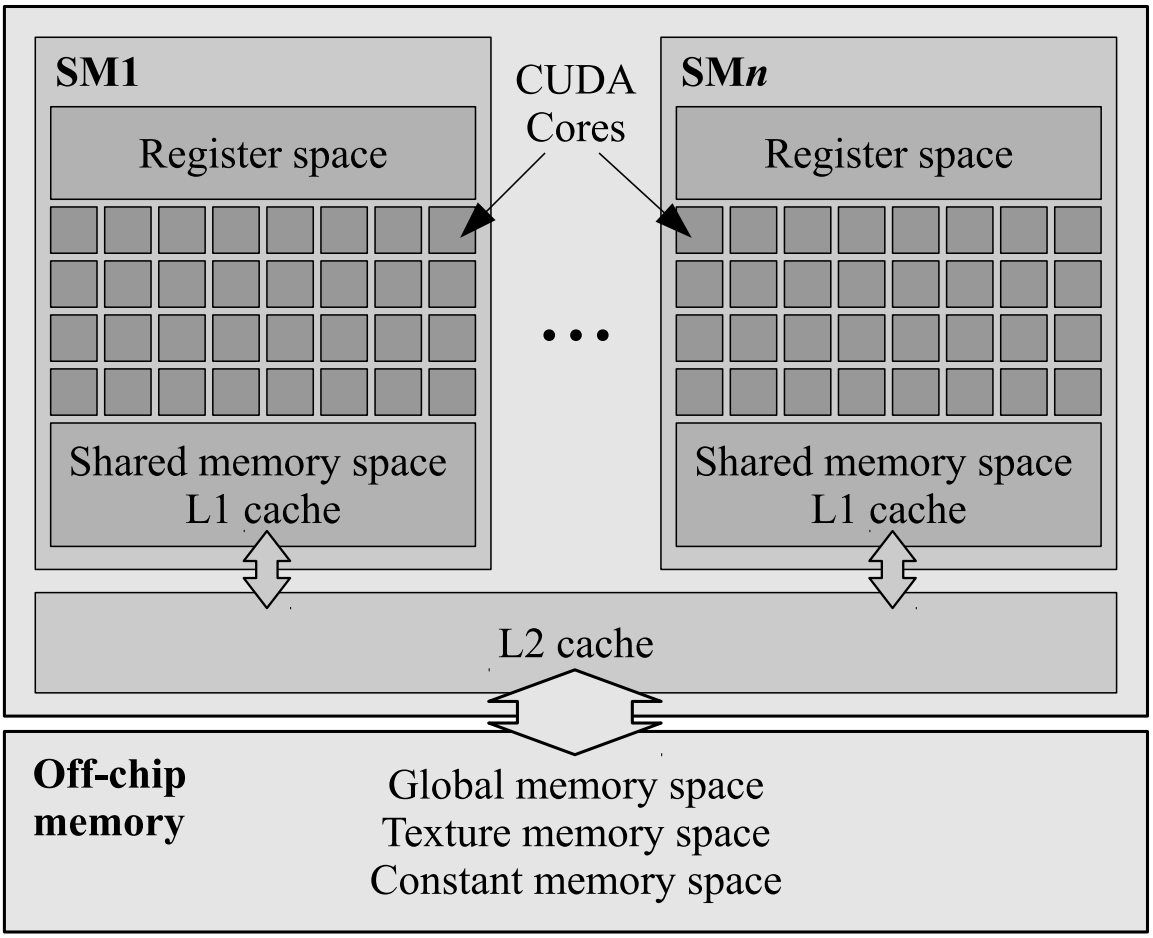


Figure 1: Simplified streaming-multiprocessor architecture for an NVIDIA Fermi GPU.

the resources required by the thread blocks. If more thread blocks are launched than there are available multiprocessors, some thread blocks might have to wait for a multiprocessor.

The memory hierarchy is organized into a *global memory*, a read-only *constant memory* and a *texture memory*. These memory spaces are in an off-chip location and are available for all the threads. Global memory is shared between all SMs, and data must be loaded from the CPU memory into the global memory space before it can be accessed by any thread. There is an on-chip *shared memory* space available per block which enables an extremely rapid read/write access to the data (roughly 100x faster than global memory). Each thread block usually stores the portion of the data that is going to process from global memory into this memory and copies the results back to global memory. The contents in the shared memory space are automatically invalidated after the kernel execution. Prioritizing the use of the shared memory space over the global memory space is recommended, but the decision is up to the programmer [28].

All global memory accesses are cached. The on-chip per-SM memory can be configured to be partially used as an *L1 cache*, aside from the space already devoted to shared memory. There is also an *L2 cache* common for all SMs.

Communication between threads can be achieved and through the shared memory space and using *synchronization barriers*. However, these barriers only affect all the threads in a thread block. As it was noted before, each thread block is independent from all others, which precludes the sharing of data between threads from different blocks.

4 Level-set segmentation

In this section we briefly introduce the mathematical fundamentals of the level-set methods and discuss the characteristics of the local level-set methods. We also describe the FTC method [37], intended for fast segmentation of similar quality to other approaches based on the solution of partial differential equations.

4.1 Level-set fundamentals

The level-set method is a numerical technique for tracking interfaces and shapes. Let $\phi(\mathbf{x},t) : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{x} \in \mathbb{R}^n$, be an n-dimensional Lipschitz continuous function. A closed $(n-1)$-dimensional hypersurface Γ is implicitly defined as the k level-set of ϕ:

$$\Gamma(t) = \{\mathbf{x} \in \mathbb{R}^n \,|\, \phi(\mathbf{x},t) = k\}. \tag{1}$$

The hypersurface, also called front, curve or interface, encloses a (sometimes multiply connected) open region denoted as Ω. Level-set methods compute the motion of Γ, and consequently, the deformation of Ω. The level-set function should verify the following properties [30]:

$$\begin{aligned} \phi(\mathbf{x},t) &< 0 \quad \text{if } \mathbf{x} \in \Omega \\ \phi(\mathbf{x},t) &> 0 \quad \text{if } \mathbf{x} \notin \Omega \\ \phi(\mathbf{x},t) &= 0 \quad \text{if } \mathbf{x} \in \partial\Omega = \Gamma(t). \end{aligned}$$

A typical level-set function is the signed distance function to Γ. The propagation of the front is linked to the evolution of function ϕ through a time-dependent initial value problem [36]. Let Γ with $k = 0$ be the zero level-set of ϕ; in order to derive an equation of the motion of this level-set function and match the zero level-set with the evolving front, the level-set value of a particle in the front with path $\mathbf{x}(t)$ must be zero:

$$\phi(\mathbf{x}(t),t) = 0. \tag{2}$$

Thus, the movement is directed by the following first-order partial differential equation:

$$\frac{d\phi(\mathbf{x}(t),t)}{dt} + \nabla\phi(\mathbf{x}(t),t) \cdot \frac{dx(t)}{dt} = 0. \tag{3}$$

Let $\mathbf{F}$ be the speed in the normal direction, i.e., $\mathbf{F} = \frac{dx(t)}{dt} \cdot \mathbf{N}$, where $\mathbf{N} = \frac{\nabla\phi}{|\nabla\phi|}$. The evolution equation can be rewritten as:

$$\frac{d\phi(\mathbf{x}(t),t)}{dt} + \mathbf{F}|\nabla\phi| = 0, \tag{4}$$

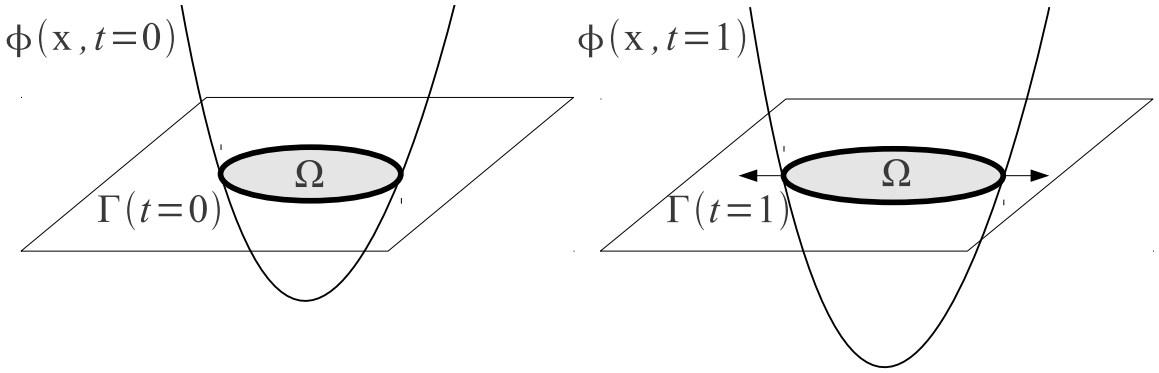


Figure 2: State of the level-set function ϕ at two different moments in time.

which is the level-set equation originally given in [31]. The level-set update equation used in level-set methods is defined as:

$$\phi(\mathbf{x},t+\Delta t) = \phi(\mathbf{x},t) + \Delta t F |\nabla \phi(\mathbf{x},t)|. \tag{5}$$

Figure 2 shows how the front evolves between two different moments in time. In this figure, the front Γ is a 2D hypersurface delimited by the zero level set of ϕ in a 3D space. The front is represented by a thick line, and the region Ω enclosed by the front appears in light gray. On the left, the front is in its initial state ($t = 0$). When $t = 1$, the level-set function has changed, hence implicitly delimiting a different front. In this particular case, the front has grown outwards.

4.2 Local level-set methods

Tracking the evolution of ϕ over the entire domain as given in Equation (5) is not computationally efficient. As we have mentioned in Section 2, several methods, known as local level-set methods, have been developed to restrict computations to relevant regions in the domain. Local level-set methods work under the assumption that only the points close to the front (i.e., where $\phi \simeq 0$) are of interest, as is the case of level-set–based segmentation.

Two of the most common strategies that can be found in the literature are the narrow-band and the sparse-field methods [11, 44]. These methods limit the evolution of ϕ to a tube surrounding the front. Thus, let ϕ be a distance function to Γ, the update equation in (5) can be rewritten as:

$$\phi(\mathbf{x},t+\Delta t) = \\ = \begin{cases} \phi(\mathbf{x},t) & \text{if } |\phi(\mathbf{x},t)| > \gamma \\ \phi(\mathbf{x},t) + \Delta t F |\nabla \phi(\mathbf{x},t)| & \text{if } |\phi(\mathbf{x}),t| \leq \gamma \end{cases}, \tag{6}$$

where γ is the radius of the tube, delimiting the size of the active domain.

It is not necessary to recompute the position of the tube in every iteration of the evolution process. In fact, narrow-band techniques only update the active domain when the front reaches the edge of the tube. While the active domain remains unchanged, only the positions of the level set ϕ enclosed by the tube are computed. On the other hand,

sparse-field techniques, such as the FTC method, update the active domain more often, every few iterations, and the tube surrounding the front is narrower, containing a small set of positions [44, 33]. In either case, both strategies reduce the computational complexity of updating the level-set field from $O(n^3)$ to approximately $O(n^2)$ in a 3D domain with cross-sectional resolution n.

Quadtree and octree methods are other level-set solutions well known in the literature [39, 40]. A quadtree mesh in $\mathbb{R}^2$ (or an octree mesh in the case of $\mathbb{R}^3$) is a hierarchical structure composed of cells organized in L levels. The root cell, which covers the whole domain, is at level 0, and each cell at level l may contain 2^2 smaller subcells (or 2^3 subcells in 3D domains) at level $l+1$. The construction of the mesh is a refinement process which guarantees that the smallest cells are close to Γ.

In quadtree and octree level-set methods, the values of ϕ are computed and stored only in the vertices of the cells of the mesh, which is reconstructed in every iteration of the algorithm. The use of an adaptive mesh makes it difficult to use high-order finite difference schemes [42], so these methods rely on semi-Lagrangian schemes, such as the CIR scheme [13, 38], to resolve the level-set equation given in (4). The computational complexity of each step of these level-set methods is $O(m \log m)$, where m is the number of elements in Γ.

Sparse block grid methods [12] use small fixed-size tiles that represent the narrow band surrounding Γ. Unlike other narrow-band methods, these methods do not need to store the full grid of data containing all the values of ϕ; instead, sparse block grid methods reduce significantly the memory requirements by keeping only the tiles that are traversed by Γ, i.e., the active tiles. Particularly, the Sorted Tile List method [42, 17] uses a list of active tiles lexicographically sorted by coordinates. As the front evolves, tiles are added and removed in the active domain. Although the sorted list must be reconstructed at each iteration of the algorithm, keeping the lexicographical ordering ensures a linear computation complexity $O(p)$ for each step, where p is the number of active tiles.

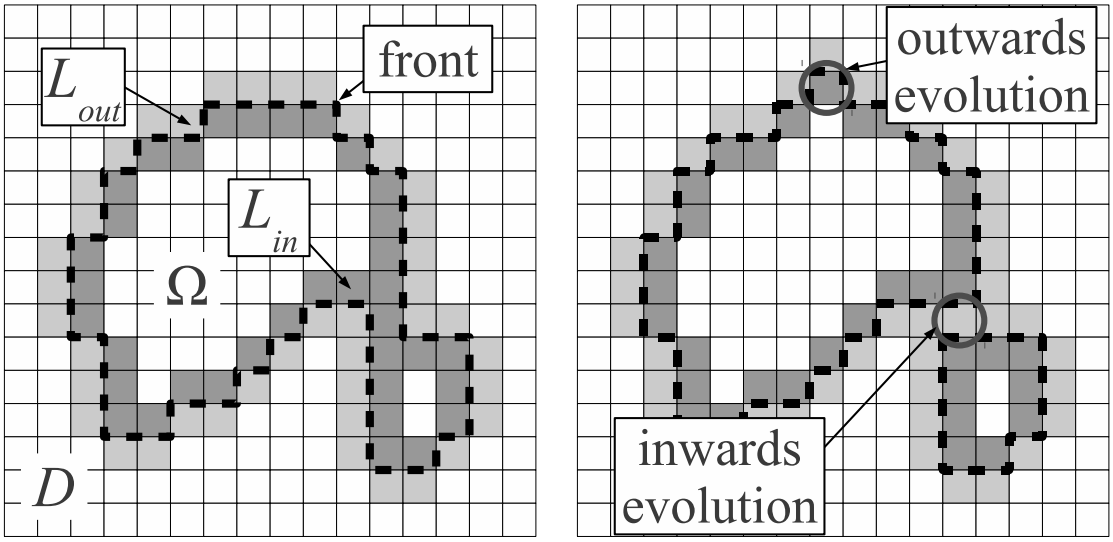
One common issue with local level-set methods is the reinitialization problem: every time the active domain is updated, the values of ϕ that were outside in previous steps must be recalculated to ensure that ϕ is a distance function in the new active domain. There are several approaches to solve this problem, which range from following the gradient of ϕ to using iterative approximations that converge into the distance function to the zero level-set.

4.3 The FTC method

The FTC method described in [37] is an approximation of a sparse-field level-set method. It computes the motion of an implicitly represented front on a discrete and uniformly sampled grid D, as shown in Figure 3(a). Given an object region $\Omega \subseteq D$, the front (the dashed line) is located between the sets of points L_{in} and L_{out} (in dark and light gray, respectively). These two sets of neighboring points are defined as:

$$\begin{aligned} L_{in} &= \{\mathbf{x} \,|\, \mathbf{x} \in \Omega \text{ and } \exists \mathbf{y} \in \eta(\mathbf{x}) \text{ such that } \mathbf{y} \notin \Omega\} \\ L_{out} &= \{\mathbf{x} \,|\, \mathbf{x} \notin \Omega \text{ and } \exists \mathbf{y} \in \eta(\mathbf{x}) \text{ such that } \mathbf{y} \in \Omega\}, \end{aligned} \tag{7}$$

where $\eta(\mathbf{x}) = \{\mathbf{y} \in D \,|\, \sum_{k=1}^{n} |y_k - x_k| = 1\}$ is a discrete neighborhood of $\mathbf{x}$. The algorithm can be generalized to any choice of neighborhood.



(a) (b)

Figure 3: Representation of the level-set front in a 2D discrete grid (a) and example of evolution of the front (b).

Focused on optimization, the FTC method defines ϕ as an integer function that can take values from the limited set $\{-3,-1,+1,+3\}$. Considering *interior points* as those points inside Ω but not in L_{in}, and, conversely, *exterior points* as those points outside Ω but not in L_{out}, ϕ locally approximates the signed distance function:

$$\phi(\mathbf{x}) = \begin{cases} +3 & \text{if } \mathbf{x} \text{ is a exterior point} \\ +1 & \text{if } \mathbf{x} \in L_{out} \\ -1 & \text{if } \mathbf{x} \in L_{in} \\ -3 & \text{if } \mathbf{x} \text{ is a interior point.} \end{cases} \tag{8}$$

The front evolves outwards or inwards by adding or removing points $\mathbf{x}$ in L_{in} and L_{out}, hence changing the value of $\phi(\mathbf{x})$, and updating the neighboring points accordingly in order to keep a consistent representation of the front. Figure 3(b) shows the result of the front evolving outwards and inwards in two different positions in the discrete grid of Figure 3(a).

Listing 1 shows the workflow of the FTC method in pseudocode. The algorithm is iterative, and for each external iteration two different loops are executed: cycle one and cycle two. N_1 and N_2 are, respectively, the number of iterations assigned to cycle one and cycle two. Cycle one corresponds to a data dependent evolution, i.e., the front evolves depending on the input data from the image or the volume being segmented. Cycle two applies a smoothing via Gaussian filtering that makes the front evolve based on its curvature.

Both cycles execute similar steps, summarized in Equation (5), but with a different speed function F. Cycle one uses F_d, which is a function of the image data and can also depend on the geometric properties of the front. This function is defined in accordance to the segmentation problem. Cycle two uses a smoothness regularization speed F_{int}, which is proportional to the mean curvature. Thus, the FTC approximates the evolution of the

```
1: for i = 1 → N1 do
2:     execute cycle one
3:     stopCond ← check stopping condition
4:     if stopCond then break
5: end for
6: for i = 1 → N2 do
7:     execute cycle two
8: end for
9: if not stopCond then
10:    repeat again from line 1
11: end if
```

Listing 1: The FTC algorithm.

front as if $F = F_d + F_{int}$. Evolution depends on the sign of the speed function: the front "advances" if the sign is positive, and "recedes" if the sign is negative.

The segmentation consists in the iterative execution of cycles one and two. The process finishes when the stopping condition is satisfied:

$$\begin{aligned} &\forall \mathbf{x} \in L_{out}, \quad F_d(\mathbf{x}) \leq 0 \\ &\forall \mathbf{x} \in L_{in}, \quad F_d(\mathbf{x}) \geq 0, \end{aligned} \tag{9}$$

or if a prespecified maximum number of iterations is reached.

The goal of the FTC method is to localize object regions within images and volumes. Only the points closer to the front are of interest, and, in consequence, the active domain is a very narrow band of two-voxel width. The method uses two speed functions, so its update equation can be written as:

$$\phi(\mathbf{x}, t+1) = \begin{cases} g(F_d, \phi(\mathbf{x}, t)) & \text{for cycle one} \\ g(F_{int}, \phi(\mathbf{x}, t)) & \text{for cycle two} \end{cases}, \tag{10}$$

where g transforms the values of $\phi(\mathbf{x}, t)$ into $\phi(\mathbf{x}, t+1)$ to make the front advance or recede according to the value of the speed function at each point $\mathbf{x}$.

Notice that the update equation in (10) does not consider the gradient of ϕ. The front evolves with no need of solving PDEs, and the evolution is strictly controlled by the values of the speed functions. Thus, the values of ϕ are just labels that identify positions within the front, inside and outside the object region Ω (similar to the label fields used in other sparse-field approaches, such as [44]).

While restricted to discrete and uniformly sampled grids, the FTC method keeps features from the level-set methods that are relevant to image and volume segmentation, such as the automatic handling of topological changes. The method converges to the expected object region as long as the initial level-set field ϕ and the speed function verify the conditions established in [37], and no reinitialization is required during its execution.

5 Implementation of the FTC method in GPU

In this section we propose two variants of the FTC method implemented in CUDA.

```
 1: <identify initial active tiles (in global mem.)>
 2: for i = 1 → N_0 do
 3:     <execute cycle one (N_1 iters. in shared mem.)>
 4:     <identify active tiles (in global mem.)>
 5:     stopCond ← <check stop. cond. (in gl. mem.)>
 6:     if stopCond then break
 7: end for
 8: <execute cycle two (N_2 iters. in shared mem.)>
 9: <identify active tiles (in global mem.)>
10: if not stopCond then
11:     repeat again from line 1
12: end if
```

Listing 2: Updating the level-set field in the GPU for proposal 1. GPU calls appear enclosed by less-than and greater-than signs. The remaining code is executed by the host.

As described in Section 4.3, the original FTC method uses two linked lists containing the positions where the front is located. When the front evolves, these lists are continuously updated to represent the last state of the front. As linked lists cannot be efficiently implemented on GPU, a different approach was used in our proposals. They are based on the idea of partitioning the domain into fixed-size tiles that can be stored in shared memory and processed by the thread blocks. Our proposals use a broader representation of the active domain, which is composed by the tiles where the front is located. The evolution of the front is computed in parallel for each of the active tiles without requiring frequent updates of the active domain. Our proposed changes of the FTC method require an alteration of the original algorithm's structure, as explained below.

The first proposal (proposal 1) modifies the number of iterations of cycles one and two to map the algorithm to the GPU, avoiding computations that do not imply a significant evolution of the front.

The second proposal (proposal 2) differs in that it performs a more restrictive selection of active tiles, hence reducing the size of the active domain. This improves the performance with no significant loss of quality.

Both proposals of the FTC have the same memory requirements: they allocate a buffer in GPU's global memory to store the 3D volume data. Two 3D buffers are also allocated to store the current and previous level-set field, ϕ^{write} and ϕ^{read}. These buffers are of the same size of the volume and contain 8-bit data. Both the level-set field and the volume data are divided into fixed-size tiles.

Additional buffers are required to track the active tiles on the field during the execution of the algorithm: eight 1D buffers to store the coordinates of the active tiles and two 3D buffers, U and V, used as scratch-pads. These buffers contain as many elements as tiles used to divide the volume.

The contents in ϕ^{read} are initialized by the user, who sets the position and size of the initial seed from which the front evolves. This seed is constructed in the host's memory, and copied to GPU altogether with the volume data as part of the initialization step.

5.1 Proposal based on modifying the structure of the FTC method

The workflow of this proposal is shown in Listing 2. The GPU calls invoked from the host code are enclosed by less-than and greater-than signs, along with the type of memory used. The algorithm starts by identifying the initial active tiles based on the contents of ϕ^{read}. At this stage, a tile is considered active if, and only if, at least one element in the tile belongs to the exterior border of the front, denoted as L_{out} in the original FTC:

$$\exists \mathbf{x} \in \text{Tile}(\phi^{read}), \mathbf{x} \in L_{out}. \tag{11}$$

The remaining tiles are considered inactive and need no processing.

The process of identifying active tiles at the beginning of the algorithm (line 1) is as follows. For each initial active tile, a value of one is written into the scratch-pad buffer U. When all values have been written, U is compacted [15]. Using the auxiliary buffer V, a condensed coordinate list of active tiles, denoted by A, is generated. Compaction was partially implemented using the C++ template library for CUDA Thrust [9].

After the initialization, the algorithm follows the general steps as the original FTC algorithm already described in Section 4.3. However, the workflow has been modified to optimize the GPU execution, as shown in Listing 2. First, the host code is no longer required to iterate through each of the individual steps of both cycles, as this is done internally in the device code. Two CUDA kernels, denoted as cycle-one and cycle-two kernels, execute N_1 and N_2 iterations of cycle one and cycle two, respectively. The main differences with the CPU implementation are discussed below.

With respect to the sequential code, a new loop has been added to execute several groups of iterations of cycle-one kernel for each main iteration of the algorithm. A new parameter N_0 determines the number of times this loop is executed. The cycle-one kernel applies N_1 times the iterative process that results in the evolution of the front, which is independently performed within each tile. However, due to the restricted size of the dimensions of these tiles ($8 \times 8 \times 8$ in our case), applying this iterative process more than 8 times does not have any effect, as the front cannot be moved beyond the edges of the tile. The solution consists in using values of N_1 less or equal to 8, and executing the kernel calls (lines 3 and 4 in Listing 2) within this new loop, so the active tiles are recomputed more frequently, allowing the front to evolve a greater distance before being modified by the cycle-two kernel. It should be noted that for cycle two there is no similar parameter as the number of times this cycle is executed is commonly less than 8.

The cycle-one and cycle-two kernels launch as many thread blocks as active tiles. Within the kernel code, each thread identifies the active tile assigned to its thread block and, from its identifier in the thread block, it also infers the coordinates of the element in the volume data and the level-set field that must process. Then, the level-set field data from ϕ^{read} is copied into a 3D buffer in shared memory that accommodates all the elements belonging to a tile including an additional overlap region. This overlap region is needed as the processing of an element requires its neighboring elements. The overlap region is not modified during the kernel execution.

Our implementation of cycle one uses an overlap region with a width of one element for each tile of data. For cycle two, a larger overlap region might be needed depending on the size of the smoothing filter, which is set by the user. Therefore, the process of loading data into the shared memory space must be general enough to be easily extended to any possible

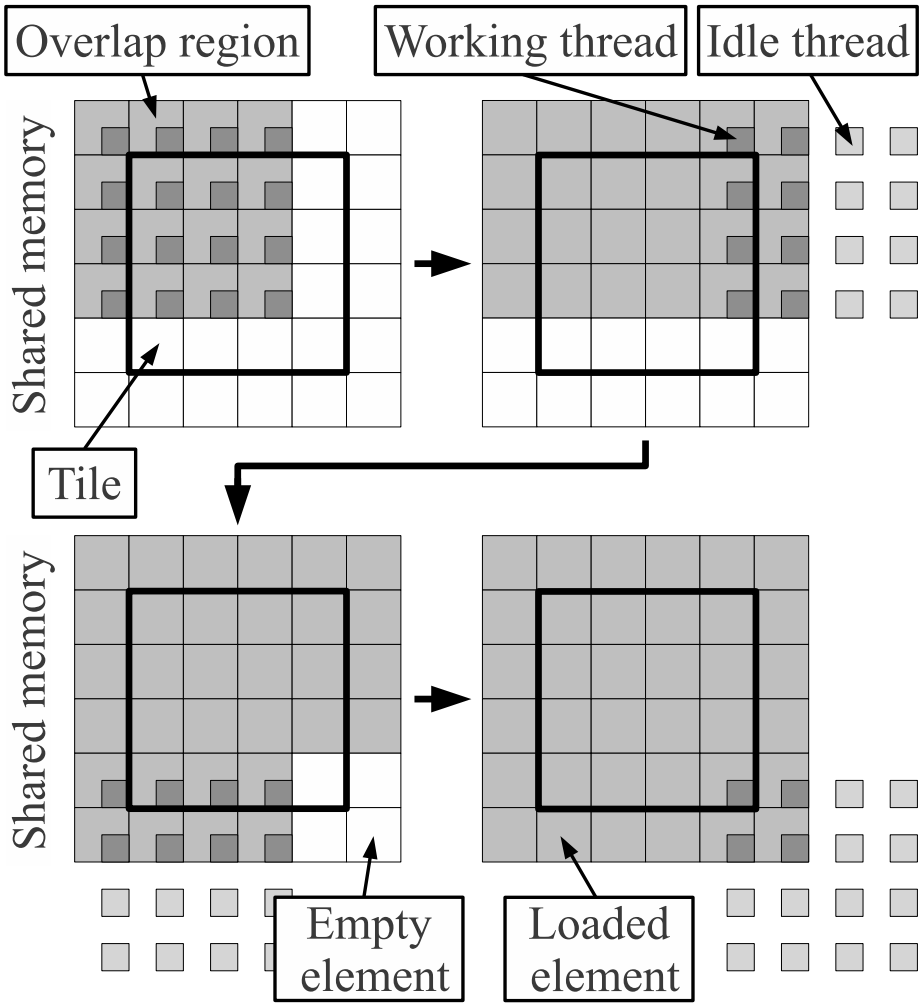


Figure 4: Iterative process of loading a 4×4 tile of data into GPU shared memory.

configuration of tile size, thread-block size and overlap-region width. Figure 4 shows the process of loading 6×6 data elements into the shared memory space using a block of 4×4 threads. The thread block is shifted over the set of data in four steps.

Splitting the workflow into tiles processed in parallel generates inconsistencies that must be fixed. An example is shown in Figure 5. The front evolves to the right (subfigure (a)), staying in the middle of tiles 1 and 2 (subfigure (b)) and leaving the exterior border (L_{out}) outside the tile. Then, tile 2 is activated (subfigure (c)). In the next iteration of the algorithm, L_{out} is added to tile 2, where the inconsistency is fixed. The code to perform this operation is shown in Listing 3. In this code, i_s denotes the coordinates of an element on the edge of a tile, and i_b denotes the coordinates of a neighboring element in an adjacent tile.

Once the inconsistencies on the edges of the tile have been fixed, the evolution of the level-set field is computed in an iterative way. This process is identical in both the cycle-one and cycle-two kernels, with the sole exception of how the speed function is calculated.

Listing 4 shows the code executed by cycle one and cycle two. This code computes the evolution of the front in N iterations according to the values of the speed function f

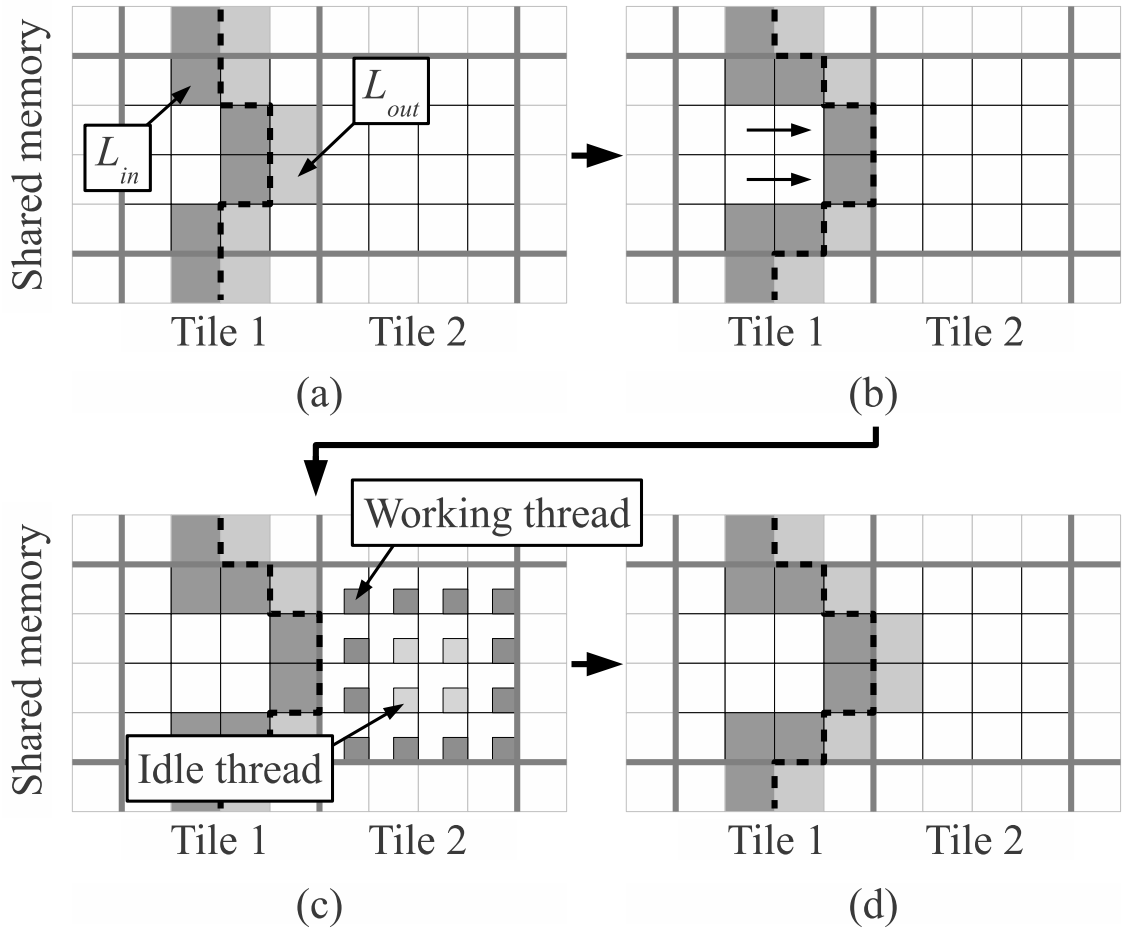


Figure 5: An example of fixing inconsistencies in shared memory.

(where $N = N_1$ and $f = F_d$ for cycle one, and $N = N_2$ and $f = F_{int}$ for cycle two). ϕ^{shared} is modified on shared memory (coordinates i_s) and finally copied into global memory (ϕ^{write}, coordinates i_g).

Once cycle one and cycle two's code has been executed, and before finishing the kernel execution, a thread in each of the thread blocks starts the process of checking if its thread block's tile and the adjacent ones will be active in the next iteration of the algorithm. The current tile is considered active if it contains an element belonging to the exterior border of the front (L_{out}):

$$\exists \mathbf{x} \in \text{Tile}(\phi^{shared}), \mathbf{x} \in L_{out}. \tag{12}$$

Additionally, an adjacent tile is considered active if the closest edge on the current tile contains at least one element from the exterior border.

After the active tile list has been built, the stopping condition is checked. A kernel call evaluates the stopping condition for each element in the level-set field that belongs to an active tile. Checking the stopping condition requires the computation of F_d for all elements that belong to the exterior and the interior border of the front (L_{out} and L_{in}, respectively).

```
1: for all elements ∈ Edge(φ^shared) parallel do
2:     if φ^shared(i_s) = +3 and φ^shared(i_b) = −1 then
3:         φ^shared(i_s) ← +1
4:     end if
5:     if φ^shared(i_s) = −3 and φ^shared(i_b) = +1 then
6:         φ^shared(i_s) ← −1
7:     end if
8: end for
```

Listing 3: Fixing inconsistencies in GPU shared memory.

```
1: for all elements ∈ Tile(φ^shared) parallel do
2:     for i = 1 → N do                          ▷ N can be N_1 or N_2
3:         f ← F(i_g)                            ▷ F can be F_d or F_int
4:         outwardsEvolution(f)
5:         inwardsEvolution(f)
6:     end for
7:     φ^write(i_g) ← φ^shared(i_s)
8: end for
```

Listing 4: Cycle-one and cycle-two evolution process in GPU.

Again, possible inconsistencies in the values of the level-set field derived from the tile-based updating must be fixed.

5.2 Proposal based on reducing the number of active tiles

This proposal modifies the previous one by using a different criterion to select the active tiles in lines 4 and 9 of the pseudocode in Listing 2. The objective is to reduce the number of active tiles by not setting as active those tiles where the front is already stable, hence, not launching thread blocks that would compute tiles where the front would not evolve. Our proposal requires a final iteration processing the entire front.

As the segmentation front grows in size and complexity, the number of active tiles in the level-set field also increases. But only one subset of the front is growing at any given time and some parts of the front stabilize long before the segmentation finishes. In proposal 1, tiles that belonged to the stabilized front were still considered active tiles, although they

```
do several iterations until stop. cond. is satisfied
    execute lines 2–9 of Listing 2
    (use a more restrictive criterion for active tiles)
end do
recompute active tiles using less restrictive criterion
do only one iteration
    execute lines 2–9 of Listing 2
end do
```

Listing 5: Updating the level-set field in the GPU for proposal 2.

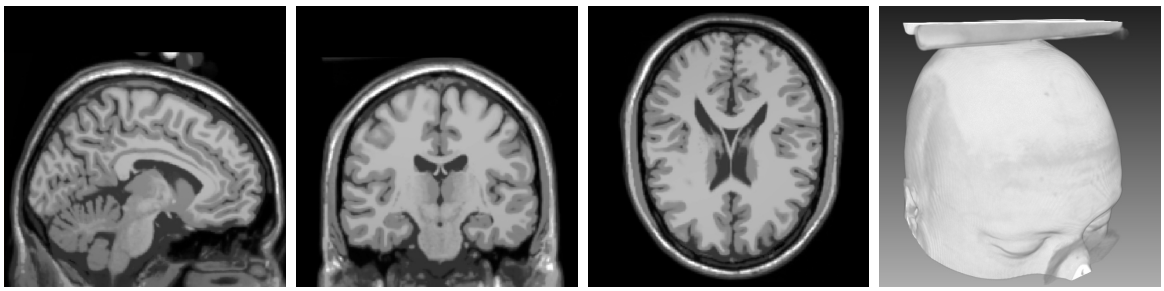


Figure 6: Sagittal, coronal, and transverse slices, and a rendering of the BrainWeb volume.

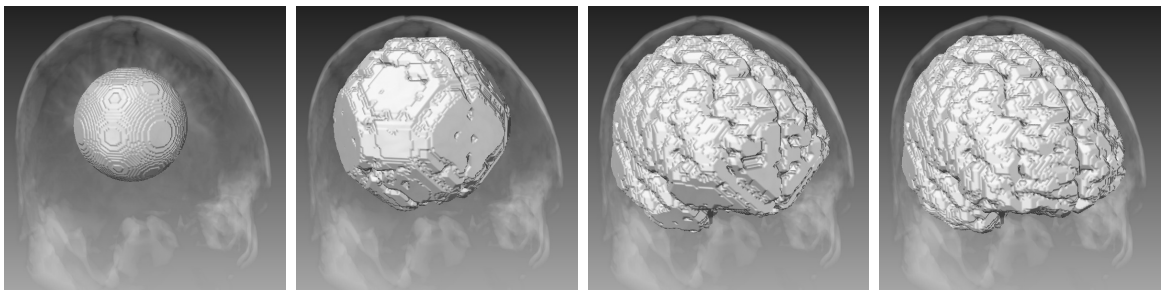


Figure 7: Segmentation of the BrainWeb volume at different stages of execution.

did not require more computation. In proposal 2, a tile is considered active if, and only if:

$$\forall \mathbf{x} \in \text{Tile}(\phi^{shared}), \begin{cases} F_d(\mathbf{x}) > 0 & \text{if } \mathbf{x} \in L_{out} \\ F_d(\mathbf{x}) < 0 & \text{if } \mathbf{x} \in L_{in}. \end{cases} \tag{13}$$

It should be noted that this is the same check as the stopping condition (see Equation (9)).

Listing 5 shows the workflow of this new proposal. The same algorithm as for our first proposal is executed, only changing the criterion to identify active tiles. At the end of the algorithm, an additional iteration is applied, this time identifying the active tiles with the less restrictive previous condition, hence obtaining all the tiles that are traversed by the front. This performs a smoothing operation even on those tiles where the front was already stable and had not evolved in the last iterations. The computational cost of this operation is greater than the sum of the previous iterations. Even so, our results show that the overall performance of this proposal is better than the previous one with no significant loss of quality.

6 Results

Our tests were performed on an NVIDIA GeForce GTX 580 with 512 processor cores grouped into 16 SMs of 32 SPs each, at a clock rate of 1.544 GHz, and with 1.5 GB of global memory. Each SM has 64 kB of RAM with a configurable partitioning of shared memory and L1 cache (16 kB of shared memory and 48 kB of L1 cache, or vice versa). In our tests we have selected a large cache. Additionally, a unified L2 cache of 768 kB is available for all SMs [27, 29]. The CPU sequential version of the original FTC algorithm was evaluated

Cache type	Intel Core i7	NVIDIA GTX 580
L1	64 kB / core	48 kB / SM
L2	256 kB / core	768 kB
L3	8192 kB	n/a

Table 1: CPU and GPU cache memory hierarchy

on an Intel Core i7 with four cores at 2.8 GHz (that can reach 3.7 GHz for intensive computational tasks that require only one thread of execution) and 8 GB of RAM [16]. Each core has separate L1 caches for instructions and data (32 kB for instructions and 32 kB for data), and a unified L2 cache. Table 1 presents a summarized description of the different memory hierarchies. Our code was compiled using the NVIDIA `nvcc` compiler provided by the CUDA 4.0 toolkit under Linux. The sequential code was compiled with `gcc` version 4.4.3.

Given that the main objective of our work is to achieve a close-to-real-time execution of the FTC algorithm by adapting it to the GPU, we have to compare our GPU implementations to the original CPU one as described in [37]. To this end, we used the same speed function as the authors of the FTC algorithm:

$$F_d(\mathbf{x}) = \begin{cases} +1 & \text{if } I(\mathbf{x}) \in [I_1, I_2] \\ -1 & \text{otherwise,} \end{cases} \tag{14}$$

where I is the image, and $[I_1, I_2]$ is the range of intensities in the region to be segmented.

Performance has been evaluated by measuring the execution times and speedups obtained by our proposals, and comparing those measures to the CPU execution of the FTC algorithm. Moreover, the quality of the segmentations has been assessed using the Dice coefficient [14], which is an index commonly applied to compare segmentation results to a known correct segmentation. For two segmented object regions Ω_1 and Ω_2, it is defined as:

$$d(\Omega_1, \Omega_2) = \frac{2 \cdot |\Omega_1 \cap \Omega_2|}{|\Omega_1| + |\Omega_2|}. \tag{15}$$

The Dice coefficient is a special case of the kappa statistic (κ) [45]. Values of $\kappa > 0.6$ indicate substantial agreement [20], although in the literature it is more common to find Dice values of 0.9 as representative of desirable segmentation quality when a ground truth is available.

6.1 Performance and quality measurements

We present two sets of results. First, we validate our proposals by segmenting a synthetic MRI image of the brain for which a ground truth segmentation is available. Second, we measure the performance of our proposals segmenting blood vessels in a brain CT.

In order to assess the quality of our proposals, we have used a volume downloaded from the BrainWeb Simulated Brain Database [1]. This database contains a set of realistic MRI data volumes produced by an MRI simulator and it is broadly used in other published works. Customized simulations can be generated by setting a wide variety of parameters,

Image		N_0	N_1	N_2	N_g	σ	Time	Speedup	Dice coeff.
BrainWeb	CPU impl.	-	30	3	3	2	2.346 s	–	0.95
	GPU prop. 1	5	6	3	3	2	0.957 s	2.5x	0.96
	GPU prop. 2	5	6	3	3	2	0.565 s	4.2x	0.96

Table 2: Segmentation parameters used in our tests and results obtained for the BrainWeb volume of size $181 \times 217 \times 181$ segmenting gray and white matter.

including slice thickness and noise level. In our tests, we have used the anatomical head model without brain lesions, T1 modality, a slice thickness of 1 mm, an infinitely high signal to noise ratio, and 20% level of intravoxel intensity non-uniformity. The final result of the simulation is an MRI image of size $181 \times 217 \times 181$ bytes with uniform coordinates. We have segmented the gray and the white matters of the brain, comparing our results to the ground truth available at the BrainWeb database.

Figure 6 shows selected slices and a volume rendering of the BrainWeb volume with a transfer function sensitive to soft tissue. Figure 7 shows the progressively GPU-segmented brain rendered as an isosurface. The segmentation process was implemented in Amira [10], and all the screen captures were taken within that application. The image data are rendered using a volume rendering technique with a transfer function that results in a high transparency. Initially, the segmentation volume takes the shape of a sphere whose starting position and size has been configured by the user. As the segmentation progresses, this initial seed grows following the shape of the region expected to be segmented. The process ends when the segmentation volume cannot evolve any longer. This segmentation needed roughly 20 MB of the GPU global memory.

Table 2 contains the list of parameters used by the FTC implementations. N_0 indicates the number of times the cycle-one kernel is called for each iteration of the algorithm (this parameter is only applicable to the GPU implementations). N_1 and N_2 are the number of iterations that correspond to cycle one and cycle two, respectively. N_g is the Gaussian-filter size for the smoothing operations. Finally, σ is the variance term of the Gaussian filter.

Table 2 also details the results obtained in terms of time, speedup (calculated with respect to the CPU execution of the original FTC algorithm), and Dice coefficient (measured against the aforementioned ground truth). GPU proposal 1 is 2.5 times faster than the sequential implementation, whereas GPU proposal 2 achieves a speedup by a factor of 4.2. In both cases, using the GPU to parallelize the segmentation results in a noticeable increase of the performance. In addition, the Dice coefficient values obtained establish our proposals as valid segmentation solutions, at least as good as the original FTC method. Besides, it can be seen that reducing the number of active tiles in GPU does not impact the quality of the segmentation.

Additional results were obtained by segmenting a dataset comprising several contrast-enhanced brain vessel CT images that presented some observable cases of aneurysms. All datasets are represented as a uniform (regularly spaced) scalar field. Scalars are encoded as 16-bit integer values. Table 3 shows the size of the images (labelled as A80, A81, A83, and A86) and the space required in the global memory of the GPU to perform the segmentation according to the memory space requirements detailed in Section 5. Figure 8 contains selected slices of the set of images used in our tests. Figure 9 shows the progression of the GPU segmentation of the vessel images.

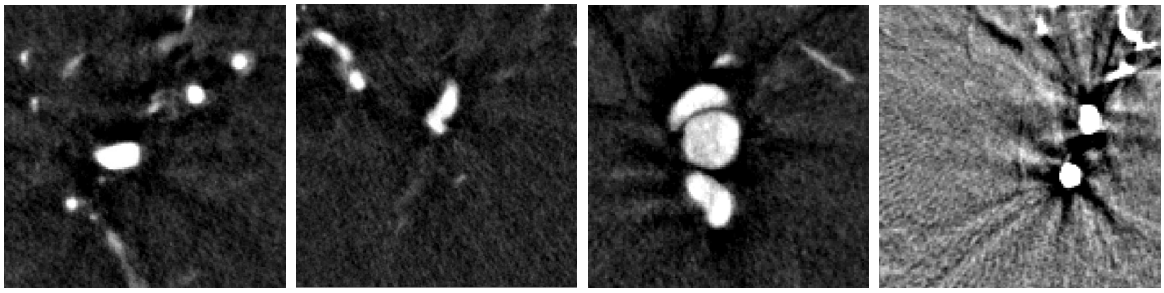


Figure 8: Selected slices from the images used in our tests. From left to right: A80, A81, A83, and A86.

Figure 9: Segmentation of the A80 image at different stages of execution.

Table 4 details the parameters used for the three implementations in each case of study, and the results obtained in time and speedup, validating our proposals for accelerating the segmentation without requiring expensive high performance computers. These results are also shown in Figure 10. Our tests show that GPU proposal 1 of the FTC algorithm is at least as fast as the CPU implementation, even for small segmentation volumes such as the brain vessels. GPU proposal 2 is three to eight times faster, as reducing the size of the active domain results in a better performance.

In order to analyze the influence of the number of active tiles and the number of iterations with respect to the execution time some experiments have been performed over the BrainWeb volume. Figure 11 represents the number of active tiles each time the computational domain is updated in our two GPU proposals for the BrainWeb case of study. The segmentation starts with a computational domain of nearly 500 active tiles, and, in the case of proposal 1, reaches nearly 4000 active tiles at the end of the process, when the segmentation volume is at its maximum size. Some noticeable drops happen each

Image	**Size**	**Glob. mem. consumption**
A80	$160 \times 161 \times 226$	22.3 MB
A81	$181 \times 176 \times 204$	24.9 MB
A83	$146 \times 174 \times 255$	24.8 MB
A86	$182 \times 133 \times 119$	11.0 MB

Table 3: Properties of the images of brain vessels used in our tests.

Image		N_0	N_1	N_2	N_g	σ	Time	Speedup
A80	CPU impl.	-	30	3	3	1	1.400 s	–
	GPU prop. 1	6	6	3	3	1	0.977 s	1.4x
	GPU prop. 2	6	6	3	3	1	0.222 s	6.3x
A81	CPU impl.	-	30	3	3	1	0.999 s	–
	GPU prop. 1	5	6	3	3	1	0.789 s	1.3x
	GPU prop. 2	5	6	3	3	1	0.205 s	4.9x
A83	CPU impl.	-	30	3	3	1	1.895 s	–
	GPU prop. 1	5	6	3	3	1	1.447 s	1.3x
	GPU prop. 2	5	6	3	3	1	0.240 s	7.9x
A86	CPU impl.	-	30	3	3	1	0.314 s	–
	GPU prop. 1	5	6	3	3	1	0.310 s	1.0x
	GPU prop. 2	5	6	3	3	1	0.094 s	3.3x

Table 4: Segmentation parameters and results obtained for the images of brain vessels.

time cycle two is computed, as this cycle has the effect of slightly shrinking the volume. The shape of the curve depends on the volume being segmented, but it tends to stabilize towards the end of the process.

GPU proposal 2 presents a completely different behavior. The number of active tiles increases up to nearly 1500 tiles, and then experiences a decrease until the end of the algorithm, as shown in Figure 11. Some spikes occur regularly, as the number of active tiles increases after each execution of cycle two. This effect is explained below.

The speed function used in cycle two, F_{int}, does not consider the intensities of the volume being segmented, but the shape of the segmentation volume. Nevertheless, as shown in Section 5.2, active tiles are selected by checking whether elements in the tile fulfill the stopping condition, which is inherently related to the way the front grows and stabilizes during cycle one. This means that an element in the front that is already stable (i.e., that fulfills the stopping condition) can evolve following the criterion used in this proposal after the execution of cycle two. This activates its current tile (and possibly the adjacent ones), thus increasing the number of active tiles.

It should be noted that the chart in Figure 11 also shows the very last iteration of the algorithm of the second proposal, where the domain is recomputed so all tiles traversed by the front are considered active. This final stage, whose main purpose is to recompute the whole front as it is done in the original FTC implementation, has the highest number of active tiles and, hence, consumes a great percentage of the total computation time of the algorithm.

Figure 12(a) shows a cumulative graph of the computation time distributed between the cycle one, cycle two, and the evaluation of the stopping condition in the GPU proposal 2 (excluding the final stage) for the segmentation of the BrainWeb image. It can be observed that the computation time consumed by each iteration describes a curve consistent with the number of active tiles in the domain (see GPU prop. 2 in Figure 11). Figure 12(b) shows, for the same case of study, how the total computation time is distributed between the kernels for all iterations, including the final stage. Neither figure shows the time consumed to compute and maintain the active domain as this overhead is negligible. Note how the

required computation time for cycle one is always much higher than for cycle two, as it requires a greater number of iterations (see Table 2).

6.2 Comparison to other works

Comparing our results to other level-set–related works is not a straightforward task. Generally, level-set–based algorithms have a broad variety of applications, and even in the segmentation field, there is a fair amount of highly specific solutions that are not suitable for a direct comparison. Besides, it is not possible in some cases to access the same databases used in other works. Moreover, not all solutions are focused on speeding up the computation time.

As mentioned in Section 2, one of the first implementations of the level-set–segmentation in CUDA is described in [34]. This solution uses a fine-grained computational domain whose tracking accounts for the 77% of the total computation time, and requires 7 seconds to complete the segmentation of the gray and white matter from a 256^3 volume obtained from the BrainWeb database on an NVIDIA GeForce GTX 280 [3].

Our GPU proposals of the FTC method split the volume into $(8 \times 8 \times 8)$-voxel tiles, dramatically reducing the size of the active domain and making the computation time required to track it negligible. In order to compare with the work published in [34], we have also evaluated our implementations on an NVIDIA GTX 295 [4], which has two cores with similar characteristics to the single-core GTX 280 (although the GTX 280 has slightly higher clock and memory frequency). In this comparison, we used just one of the cores of the GTX 295. Our second GPU proposal of the FTC method required 1.1 seconds to segment a similar volume from the BrainWeb database of size $256 \times 256 \times 181$. We can conclude that our implementation can be more than six times faster than the approach described above.

A more recent CUDA-based approach is presented in [17], with applications in surface reconstruction. Results are shown for collapsing the Stanford Dragon model (which can be obtained from the Stanford 3D Scanning Repository [8]) reconstructed on a grid of 512^3 voxels as a benchmark, taking roughly 10 seconds to collapse the model on an NVIDIA GeForce GTX 280. We repeated the same benchmark with our second GPU proposal, requiring 2.9 seconds to complete the task on an NVIDIA GeForce GTX 295. For the reasons expounded above, our solution is approximately more than three times faster.

6.3 Final remarks

In our implementation of the FTC method, the active domain is divided in tiles, and the computations are entirely performed in the GPU. Our approach makes intensive use of the shared memory space, which has a latency close to that of the register space, much lower than any other available CPU or GPU memory. We expect the next GPU generations to increase the size of the shared memory space, so our solution could use bigger tiles, and hence, improve its current performance by reducing the number operations in the global memory space.

An alternative hybrid implementation partitioning the segmentation tasks between the CPU and the GPU could be considered. This approach would necessarily require to share data between the CPU and the GPU memories due to the dependencies between tiles.

However, this communication would be through the PCI bus, which has a small bandwidth compared to the GPU memory bandwidth, and would result in the major bottleneck. Our solution could also be ported to multi-CPU architectures, or implemented in OpenMP to exploit currently widespread multicore CPUs.

The approach taken in our proposals, which roughly follows the steps (1) partition the problem in independent tasks, (2) solve each task in the shared memory space, (3) resolve dependencies by detecting and fixing inconsistencies, could be applied to other problems with a similar pattern of dependencies. The data must be divisible in chunks where a few steps of the solution can be processed independently, and the implementation must be able to detect and solve any possible discrepancies that might appear.

7 Conclusion

In this work we accelerate the segmentation of the brain from medical image data by the implementation on the GPU of the FTC algorithm. With this objective we implemented two proposals that are characterized by avoiding unnecessary computations and increasing the overall performance without impacting the segmentation quality.

Common to both proposals is the fact that the computational domain is partitioned into fixed-size tiles that can be stored in shared memory and efficiently processed in parallel by the CUDA thread blocks on the GPU. This partitioning requires to handle an overlap region per tile and to fix inconsistencies among data computed in neighboring tiles. Also in both proposals the iterative structure of the algorithm is modified to better exploit the GPU features. A list of active tiles is required to avoid computations in tiles where the level set can no longer evolve. To this end, in our first proposal this list contains all the tiles that are traversed by the front at any given moment. Our second proposal uses a more restrictive criterion to reduce the size of the active domain.

Both implementations have been executed on an NVIDIA GeForce GTX 580, and their performance has been compared to the sequential version executed on a CPU. We obtained speedup values between 3x and 8x for the brain vessels of the medical image dataset considered. Our results are competitive when compared to other recent GPU implementations of level-set methods.

References

[1] BrainWeb simulated brain database. http://www.bic.mni.mcgill.ca/brainweb/.

[2] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/.

[3] GeForce GTX 280 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications.

[4] GeForce GTX 295 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295/specifications.

[5] GPGPU. http://gpgpu.org/.

[6] OpenCL: the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[7] Parallel programming and computing platform, CUDA, NVIDIA. http://www.nvidia.com/object/cuda_home_new.html.

[8] The Stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[9] Thrust. http://code.google.com/p/thrust/.

[10] ZibAmira: Amira for research partners. http://amira.zib.de/.

[11] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 118(2):269–277, May 1995.

[12] Robert E. Bridson. *Computational aspects of dynamic surfaces*. PhD thesis, Standford University, Standford, CA, USA, 2003.

[13] Richard Courant, Eugene Isaacson, and Mina Rees. On the solution of nonlinear hyperbolic differential equations by finite differences. *Communications on Pure and Applied Mathematics*, 5(3):243–255, 1952.

[14] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, July 1945.

[15] Mark Harris, Shubhabrata Segupta, and John D. Owens. *GPU Gems 3*, chapter Parallel prefix sum (scan) with CUDA. Addison Wesley, August 2007.

[16] Intel Corporation, Santa Clara, California, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, May 2011.

[17] Andrei C. Jalba, Wladimir J. van der Laan, and Jos B. T. M. Roerdink. Fast sparse level-set on graphics hardware. *IEEE Transactions on Visualization and Computer Graphics*, 99, 2012. PrePrints.

[18] Won-Ki Jeong, Johanna Beyer, Markus Hadwiger, Amelio Vazquez, Hanspeter Pfister, and Ross T. Whitaker. Scalable and interactive segmentation and visualization of neural processes in EM datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1505–1514, November 2009.

[19] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: a Hands-on Approach*. Elsevier, Burlington, Massachussets, USA, 2010.

[20] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, March 1977.

[21] Aaron E. Lefohn, Joshua E. Cates, and Ross T. Whitaker. Interactive, GPU-based level sets for 3D brain tumor segmentation. Technical report, University of Utah, School of Computing, 2003.

[22] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

[23] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March–April 2008.

[24] Blake C. Lucas, Michael Kazhdan, and Russell H. Taylor. Multi-object geodesic active contours (MOGAC): a parallel sparse-field algorithm for image segmentation. Technical report, Johns Hopkins University, Department of Computer Science, 2012.

[25] Chohong Min. Local level set method in high dimension and codimension. *Journal of Computational Physics*, 200(1):368–382, 2004.

[26] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, March–April 2010.

[27] NVIDIA, Santa Clara, California, USA. *NVIDIA GeForce GTX 580 GPU datasheet*, 2010.

[28] NVIDIA, Santa Clara, California, USA. *CUDA C best practices guide (version 4.0)*, 2011.

[29] NVIDIA, Santa Clara, California, USA. *CUDA C programming guide (version 4.0)*, 2011.

[30] Stanley Osher and Nikos Paragios. *Geometric Level Set Methods in Imaging, Vision, and Graphics*. Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 2003.

[31] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, December 1988.

[32] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[33] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang. A PDE-based fast local level set method. *Journal of Computational Physics*, 155(2):410–438, July 1999.

[34] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross Mitchell. A work-efficient GPU algorithm for level set segmentation. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 123–132, Saarbrucken, Germany, 2010. Eurographics Association.

[35] Martin Rumpf and Robert Strzodka. Level set segmentation in graphics hardware. In *Proceedings of IEEE International Conference on Image Processing*, ICIP' 01, pages 1103–1106, Thessaloníki, Greece, 2001. IEEE.

[36] James A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving interfaces in computational fluid mechanics, computer vision, and materials science*. Cambridge University Press, Cambridge, UK, 1996.

[37] Yoggang Shi and William C. Karl. A real-time algorithm for the approximation of level-set-based curve evolution. *IEEE Transactions on image processing*, 17(5):645–656, May 2008.

[38] John Strain. Semi-Lagrangian methods for level set equations. *Journal of Computational Physics*, 151(2):498–533, 1999.

[39] John Strain. Tree methods for moving interfaces. *Journal of Computational Physics*, 151(2):616–648, 1999.

[40] John Strain. A fast modular semi-Lagrangian method for moving interfaces. *Journal of Computational Physics*, 161(2):512–536, 2000.

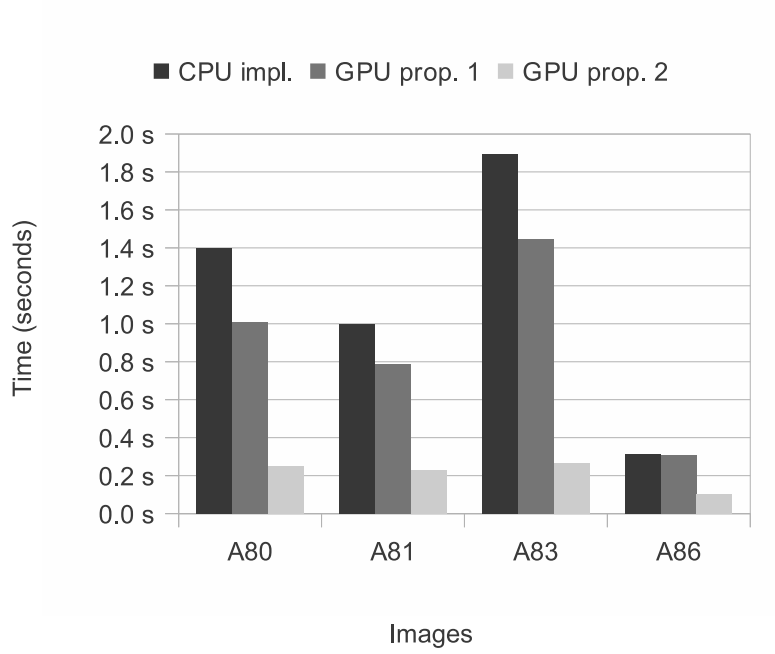
[41] Gábor J. Tornai and György Cserey. 2D and 3D level-set algorithms on GPU. In *Proceedings of the 12th International Workshop on Cellular Nanoscale and their Applications*, CNNA '10, pages 1–5, Berkeley, California, USA, 2010. IEEE.

[42] Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. A memory and computation efficient sparse level-set method. *Journal of Scientific Computing*, 46(2):1–22, February 2011.

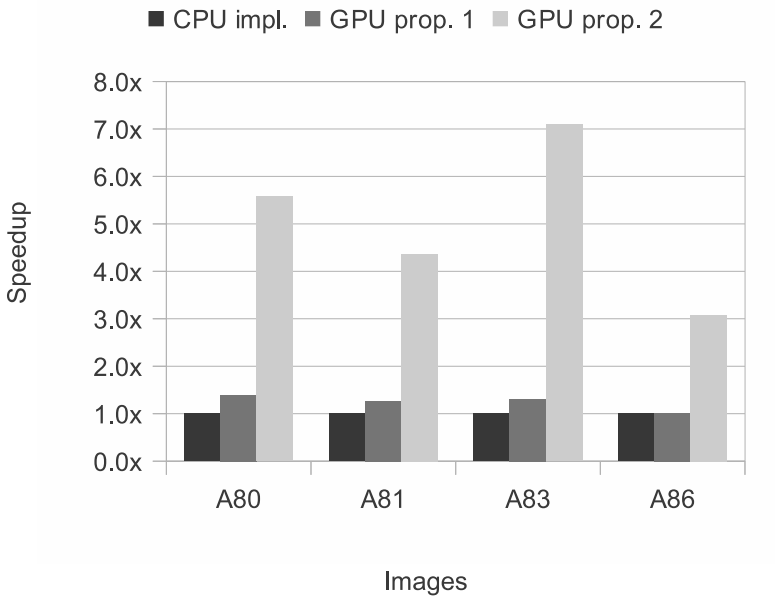
[43] Ross T. Whitaker. Volumetric deformable models: active blobs. In *Visualization in Biomedical Computing 1994*, Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, pages 122–134, Rochester, Minnesota, USA, 1994. SPIE.

[44] Ross T. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, September 1998.

[45] Alex P. Zijdenbos, Benoit M. Dawant, Richard A. Mangolin, and Andrew C. Palmer. Morphometric analysis of white matter lesions in MR images: method and validation. *IEEE Transactions on Medical Imaging*, 13(4):716–724, December 1994.



(a)



(b)

Figure 10: Performance measures in time (a) and speedup (b) for the images of brain vessels.

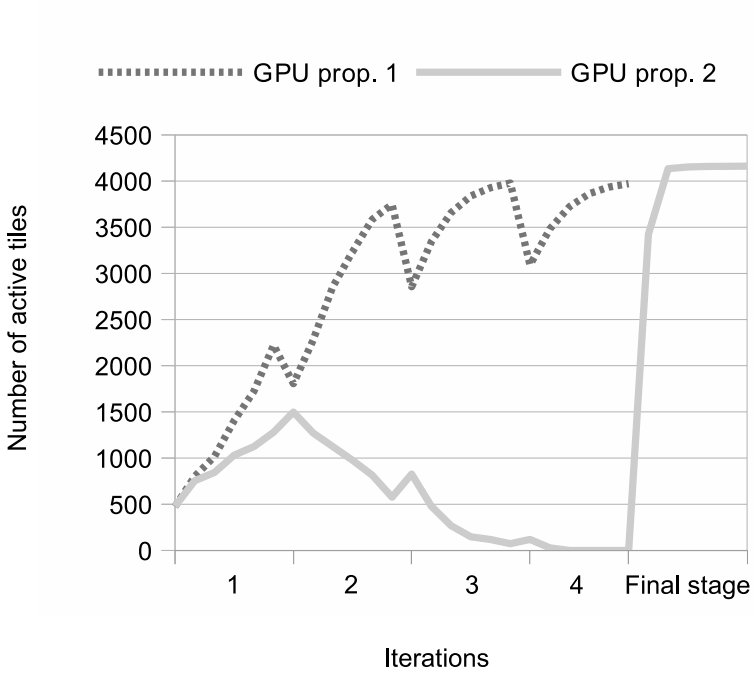
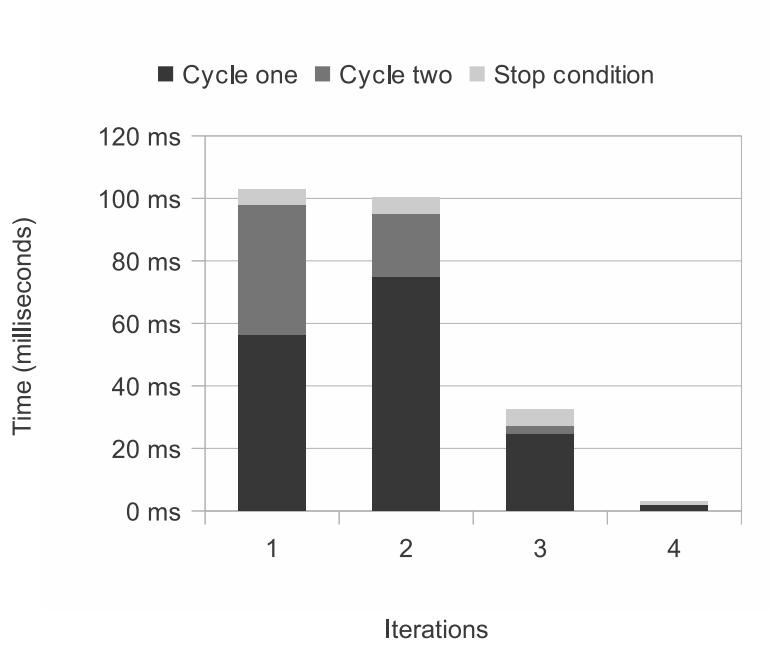
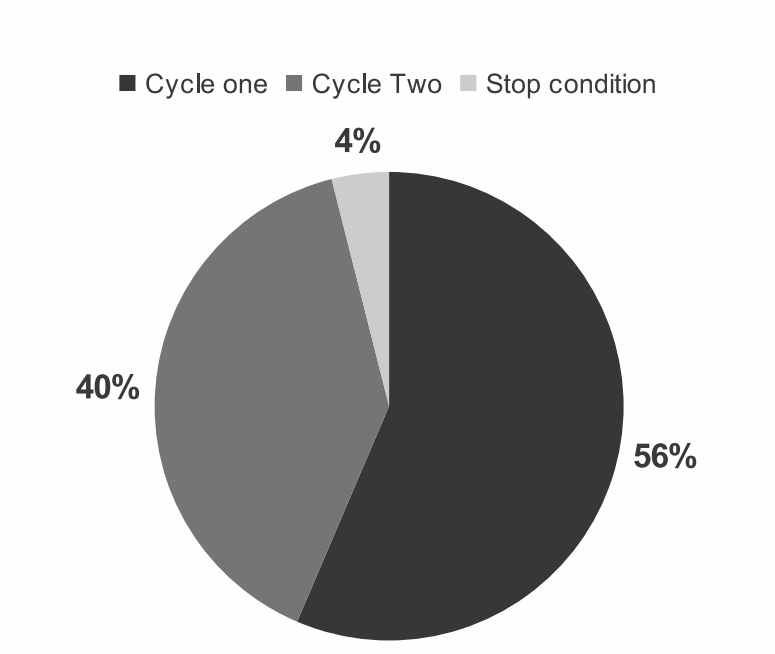


Figure 11: Active tiles during segmentation for the BrainWeb volume in both GPU proposals.



(a)



(b)

Figure 12: Computation time for BrainWeb segmentation distributed between the main steps of the algorithm for the GPU proposal 2.