Efficient 2D and 3D Watershed on Graphics Processing Unit: Block-Asynchronous Approaches Based on Cellular Automata

Centro de Investigación en Tecnoloxías da Información (CiTIUS)
University of Santiago de Compostela, Santiago de Compostela, 15842 Spain

Pablo Quesada-Barriuso, Dora B. Heras, Francisco Argüello

Abstract

The watershed transform is a method for non-supervised image segmentation. In this paper we show that a watershed algorithm based on a cellular automaton is a good choice for the recent GPU architectures, especially when the synchronization rules are relaxed. In particular, we propose a block-asynchronous computation strategy that maps the cellular automaton on the thread blocks of the GPU. This method reduces the number of points of global synchronization allowing efficient exploitation of the memory hierarchy of the GPU. We also avoid the artifacts produced in the watershed lines by the block-asynchronous updating scheme by correcting the data propagation speed among the blocks. The proposals are compared to an OpenMP multithreaded code. The high speedups indicate the potential of this kind of algorithm for new architectures based on hundreds of cores. The method is tuned to be applied to 3D volumes obtaining similar results.

Keywords: Watershed transform, Cellular Automata, Asynchronous Algorithm, CUDA, Image Segmentation

1. Introduction

The watershed transform is a non-supervised region-based segmentation tool for digital images. The idea behind this method comes from geography. A grey scale image can be represented as a topographic relief, where the height of each pixel is directly related to its grey level. The dividing lines of the catchment basins for precipitation falling over the region are called watershed lines [1]. Various definitions, algorithms and proposals can be found in the literature but, in practice, they can be classified into two groups: those based on the specification of a recursive algorithm by Vincent and Soille [1], and those based on the distance functions defined by Meyer [2].

One of the main advantages of the watershed transform is that all regions of the image are well defined at the end of the segmentation process, even if the contrast of the image is poor. For this reason it has been widely used in image processing, e.g., for medicine and biology [3]. The results of applying directly the watershed algorithm are over-segmented owing to the large number of regions detected [4]. This problem is overcome by preprocessing the image with the objective of reducing the number of regions; e.g., by applying a filter to improve the image contrast, or processing a marker-controlled watershed transform that preselects the regions of interest [4]. The computational cost of these tasks joins to high computational cost of the watershed segmentation itself.

Different sequential algorithms have been designed to compute the watershed transform using sequential structures as queues [1] or graphs [5] to simulate the flooding process. Due to the recursive nature of the watershed transformation, its parallelization is not a trivial task. Two early algorithms for computing the watershed transformations on parallel computers were developed by Moga et al. [6]. Both algorithms start by detecting the regional minima, and then the image is lower-complete transformed and represented as a graph or forest. Two methods of propagating the labels of minima are proposed and compared. These algorithms are implemented using the message passing paradigm and executed on multiprocessor systems. Moreover, the wavefront technique was introduced in this work. This is used to propagate the labels from the border to the inner of a flat zone, so the propagation reaches the middle of the flat zone at the same time. In the field of parallel processing, most efforts have focused on multiprocessors [6] and to a lesser extent on specific architectures, such as field-programmable gate arrays (FPGAs). A compendium of algorithms and parallelization strategies for watershed is presented in [7].

With high computational power, Graphic Processor Units (GPUs) have evolved into low-cost, multithreaded, multicore processors with enormous computational power, which are now

Email addresses: pablo.quesada@usc.es (Pablo Quesada-Barriuso), dora.blanco@usc.es (Dora B. Heras), francisco.arguello@usc.es (Francisco Argüello)

common in PC hardware. The Computed Unified Device Architecture (CUDA) developed by NVIDIA, based on a data parallel programming model, provides support for general-purpose computing on graphics hardware. In recent years, a number of parallel implementations of watershed algorithms in GPUs have been published [8–11]. In particular, a watershed algorithm implemented on GPU using CUDA was presented by Körbes et al. [10]. The algorithm is a four-step procedure. The efficiency of this algorithm is a result of the labelling method used, greatly reducing the number of iterations required for the task completion. The algorithm requires several steps of synchronization and non-local data movement.

Cellular automata (CA) constitute a computing model that has been extensively used for artificial life, pattern recognition [12] or image processing [13]. The popularity of CA is mainly due to the simplicity of modelling complex problems with the help of local information only. CA are dynamical systems that consist of a n-dimensional array of cells [14], each one of which can be in one of a finite number of possible states. They are synchronously updated in discrete time steps according to a local, identical, interaction rule. This requires a strict order for the component updates, where a cell cannot be updated until all other cells have been updated. The concept of parallelism is therefore implicit in CA and matches the computing model of the GPUs, multicore and many-core systems, as it is based on the individual evolution of different cells based on local information. This is the reason why different parallel algorithms based on CA for GPU and multicore systems have been developed for general purpose computing, visualization and image processing [15].

Watershed implementations based on CA were proposed using shaders in the graphic pipeline of the GPU [8]. This implementation is synchronous as it updates the entire image at each step of the evolution of the automaton. The authors developed an algorithm using image integration via the Ford-Bellman shortest paths algorithm. From each minimum of the image, a wavefront is started, labelled by the index of the minimum it started from, and the distance is initialized with the value of the minimum. The transition rule must be synchronously applied to all the cells. This algorithm is simple and fast, but it requires a previous detection of minima through the use of any other method. Galilée et al. [16] introduced a parallel algorithm-architecture based on asynchronous CA to compute the watershed transform, updating each pixel as soon the information from the neighbors becomes available.

In general, the asynchronous computing model is suitable for parallel computing where a problem is split into independent subproblems, each one solved by a different processor, minimizing the number of interprocessor communications that imply synchronization points. This data-parallel model is used by CUDA. In the CUDA parallel model, a multithreaded program is partitioned into blocks of threads that run independently from each other [17]. The order in which blocks are computed is not preset so the communications among them must be performed by means of synchronization points, that are costly. So, we have found that the asynchronous algorithm described in [16] is particularly suitable for the CUDA computing model as different regions of the image can be simultaneously and independently updated during certain number of steps, thus reducing the number of synchronization points. Taking into account that multi-CPU computers also benefit from a high computations to communications ratio, research on algorithms that can be partitioned into blocks and asynchronously computed is relevant.

In this paper we present a block-asynchronous computation method for the watershed algorithm based on CA (CA-watershed) defined by [16]. We first study two proposals based on 2D CA, thus applied to 2D images, and finally they are extended to the case of 3D volumes. The proposals are called block-asynchronous and artifact-free block-asynchronous. An early version for 2D images was published in [18] and applied to hyperspectral images in [19].

We compare our GPU watershed proposals to an efficient multithreaded OpenMP implementation of the watershed on the CPU. When the block-asynchronous CA-watershed algorithm is computed a problem arises as the quality of the segmentation is slightly affected. In particular, the algorithm presents the problem of data propagation at the block boundaries which causes undesirable artifacts. The artifact-free block-asynchronous algorithm is based on the application of a technique known as wavefront [6] increasing the quality of the watershed lines obtained.

This paper is organized as follows: Section 2.1 introduces the watershed transform and, Section 2.2, the main concepts associated to CA. The watershed algorithm based on CA is described in Section 2.3. In section 3 we present the GPU architecture and the CUDA programming model. The different GPU algorithms are described in Section 4 and the results obtained are discussed in Section 5. Finally, Section 6 presents the conclusions.

2. Watershed based on cellular automata

In this section we introduce the watershed transform and the CA principles, and describe a watershed algorithm based on CA.

First, we introduce a few concepts and notations regarding topography in order to continue with the watershed transform. A grey scale image may be considered as a graph $G = (V, A)$ with a finite set of V vertexes (pixels) and a set of arcs $A \subseteq V \times V$ defining the connectivity. Two pixels u and v are connected if $(u, v) \in A$. The pixels connected to u, called neighbors, are denoted by $\mathcal{N}(u)$.

The most widely used connectivity is four, considering the orthogonal neighbors, left, right, up and down, known as the Von Neumann neighborhood. Another variation is the Moore neighborhood, where the eight neighbors surrounding a pixel are connected.

The *slope* between two neighbors is defined by:

$$\forall u \in V, \forall v \in \mathcal{N}(u), \quad slope(u, v) = h(u) - h(v),$$

where $h(u)$ is the grey value (altitude) of the pixel u. The *lower slope* is defined as the maximal slope connecting u to any of its neighbors of a lower altitude:

$$LS(u) = max(\, h(u) - h(v)\,) \mid v \in \Gamma(u),$$

with $\Gamma(u)$ the set of neighbors v with $h(v) < h(u)$. If $\Gamma(u)$ has more than one element, $\mathcal{N}^F(u)$ represents an arbitrary element of that set.

A *plateau* is a connected subgraph $P = (V_P, A_P) \subseteq G$, where $\forall u \in V_P, h(u) = c$, and c is the altitude of the plateau. Thus, a plateau is a region of constant grey value within the image. The set $\mathcal{N}^=(u)$ denotes the pixels $v \in \mathcal{N}(u)$ with $h(u) = h(v)$.

Finally, the *lower border* of a plateau P is defined as:

$$\partial_P^- = \{\, u \in V_P \mid \exists v \in \mathcal{N}(u), h(v) < h(u) \,\}.$$

If $\partial_P^- = \emptyset$, the plateau is called a *minimum plateau*. All the pixels within a minimum plateau are also minimum. In contrast, if $\partial_P^- \neq \emptyset$, the plateau is called a *non-minimum plateau*.

2.1. Watershed transform

The watershed transform is a region-based technique for image segmentation which is particularly interesting when the images have low contrast. Although various implementations can be found in the literature, in this paper we follow the Hill-Climbing algorithm based on the topographical distance by Meyer [2].

The Hill-Climbing algorithm starts by detecting and labelling all minima in the image with unique labels. The process continues propagating the labels upwards, *climbing up the hill*, following the path defined by the lower slope of each pixel. The result of the segmentation is a set of regions, each one represented by a catchment basin, with their own label. At the end all the pixels belong to a region and the watershed lines could be defined as the limits between these regions. The precision of the segmentation depends on the connectivity. The same algorithm would permit a more precise placement of the watershed lines or the catchment basins if a higher connectivity were used [2].

Problems arise for digital images with plateaus, as it is not possible to know a priori whether a plateau is minimum or non-minimum; thus, an additional processing is required. The most common solution is to preprocess the image by calculating its lower complete image [4], where each pixel has at least one neighbor with a lower value, except the pixels that are minima. Another alternative is to calculate the distances from an inner pixel to the lower border of the plateau during the watershed processing, which is the strategy selected for the present work, and which requires propagating information outside the closest neighborhood of each pixel.

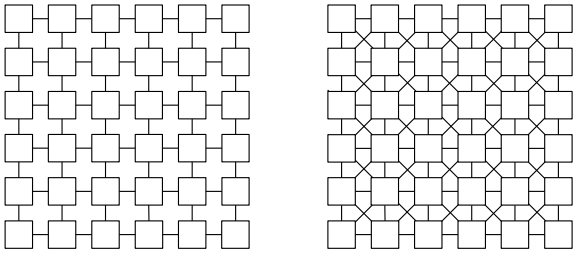

Figure 1: A 2D grid with four (left) or eight (right) neighbors.

2.2. Cellular automata

CA are computing models composed of a set of cells arranged in a regular grid of one, two or three dimensions originally proposed by John Von Neumann as formal models of self-reproducing organisms [20]. In the case of two dimensions, each cell is connected to its four or eight adjacent neighbors, depending on the connectivity, as shown in Fig. 1. CA are characterized by a set of states and a set of transition rules which determine the evolution of each cell among different states [20]. Each cell changes its state depending on the current state and the state of its neighbors. The updates of the cells are usually performed synchronously and in discrete time steps.

If the updates of the cells are not required to take place synchronously, but each one can be updated to its next state an unbounded number of times without synchronization, then we have an asynchronous automaton [21]. In this case, the grid can be partitioned into different regions which can be independently updated.

It is possible to ignore the synchronization points associated to the computation of the CA, resulting in a so-called asynchronous CA. In this case, however, the correctness and convergence of the algorithm could be severely affected [22]. In order to efficiently develop asynchronously updating computing schemes, it is important to investigate the non-deterministic and probabilistic behavior associated to such schemes [23].

2.3. Watershed based on a cellular automaton

In this section we present the watershed algorithm introduced in [16] which can be synchronously or asynchronously implemented. The main advantage of this algorithm is that minima detection, labeling, and climbing the steepest paths are performed simultaneously and locally.

This algorithm is based on a three-state cellular automaton implementing the Hill-Climbing algorithm, as shown in Fig. 2. Each cell of the automaton computes a pixel of the image. First, the pixels are sequentially labelled in a row-major order. In the initial state, all pixels compute the sets $\mathcal{N}^F(u)$ and $\mathcal{N}^=(u)$ corresponding to an arbitrary lower slope and the neighbors with the same grey value, respectively.

If a pixel has no lower slope, $\mathcal{N}^F(u) = \emptyset$, it switches to the minimum or plateau (MP) state and its label is modified as fol-

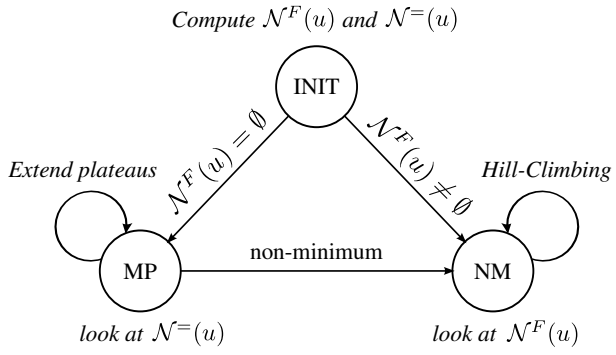

Figure 2: Three-state automaton implementing Hill-Climbing algorithm [16].

lows:

$$l(u) = min\,(\,l(v)\,)\,, \tag{1}$$

where $l(v)$ are the labels of the pixels $v \in \mathcal{N}^{=}(u)$.

Otherwise, the state of the pixel switches to non-minimum (NM) and $\mathcal{N}^{F}(u)$ points to the *climbing direction*. The grey value and the label of the pixel are modified as follows:

$$f(u) = f\Big(\,\mathcal{N}^{F}(u)\,\Big), \tag{2}$$

where $f(u)$ is the pair $h(u), l(u)$, being $h(u)$ the grey value of the pixel u and $l(u)$ its label.

Once the pixel has been initialized, the update stage begins. This is an iterative task that processes the MP and NM states. A pixel in MP state waits for data from any neighbor $v \in \mathcal{N}^{=}(u)$, and, depending on the grey value of the neighbor, two cases are considered. If the value of the neighbor is equal to or greater than its current value, the label of the pixel is updated as:

$$l(u) = min\,(\,l(u), l(v)\,) \quad \text{if } h(v) \geq h(u), \tag{3}$$

otherwise, the pixel belongs to the lower border of the plateau and its state switches to NM as follows:

$$\left.\begin{array}{l} \mathcal{N}^{F}(u) = v, \\ f(u) = f(v), \\ state = \text{NM}, \end{array}\right\} \quad \text{if } h(v) < h(u). \tag{4}$$

On the other hand, a pixel in NM state remains in that state and waits for data from the neighbor $\mathcal{N}^{F}(u)$, updating its data as follows:

$$f(u) = f(\mathcal{N}^{F}(u)). \tag{5}$$

This iterative task ends when no more changes occur. A pseudocode of the algorithm is shown below:

Algorithm 1: CA-watershed

Input: $current\_state(u)$, $f(v)$ from the neighbors
Output: $next\_state(u)$, $f(u)$
Case ($current\_state$ = "INIT")
  Compute $N^{=}(u)$, $N^{F}(u)$
  If ($N^{F}(u) = \emptyset$)
  { $l(u) \leftarrow \min_{v \in N^{=}(u)}(l(v))$; $next\_state(u) \leftarrow$ "MP"; }
  Else
  { $f(u) \leftarrow f(N^{F}(u))$; $next\_state(u) \leftarrow$ "NM"; }
Case ($current\_state$ = "MP")
  For any neighbor $v \in N^{=}(u)$:
  If ($h(v) < h(u)$)
  { $N^{F}(u) = \{v\}$; $f(u) \leftarrow f(v)$;
    $next\_state(u) \leftarrow$ "NM"; }
  Else
  { $l(u) \leftarrow \min(l(u), l(v))$; }
Case ($current\_state$ = "NM")
  For its flooding neighbor $v = N^{F}(u)$:
  $f(u) \leftarrow f(v)$;

This algorithm is non-deterministic and may lead to different segmentation results. A formal proof of correctness and convergence towards a watershed segmentation using a mathematical model of data propagation in a graph is presented in [16].

3. GPU architecture

GPUs provide massively parallel processing capabilities based on a data-parallel architecture. There are Application Programming Interfaces (APIs) for writing programs that are executed in the GPU, such as CUDA for NVIDIA devices, or OpenCL, for heterogeneous platforms.

A CUDA program, which is called a kernel, is executed by thousands of threads grouped into blocks. The blocks are arranged into a grid and scheduled to any of the available GPU cores which enables automatic scalability for future architectures [17].

The CUDA architecture is organized into a set of streaming multiprocessors (SMs), each one with many cores called streaming processors (SPs). These cores can manage hundreds of threads in a Single Instruction, Multiple Data (SIMD) programming model. The number of cores per SM depends on device architecture.

The graphics memory is organized into a *global memory* (usually known as device memory), a read-only *texture memory* and a *constant memory*, with special features, such as caching or prefetching data. These memories are available for all the threads. Each thread has its own *local memory* and *registers*. There is also an on-chip *shared memory* space only available per block. This feature enables extremely rapid read/write access to the data in this memory but with the lifetime of the block. The new Fermi architecture includes a *cache hierarchy* consisting of a L1 and a L2 caches.

There are mechanisms for synchronizing threads within a block but not among different blocks. Owing to this restriction it is not possible to share data among blocks and the communication among them must be through the global memory. Perform computations efficiently when this situation arises is a challenge.

4. Block-asynchronous GPU algorithm

In this section we present the GPU algorithm for the CA-based watershed introduced in Section 2.3. We first explain the method for a 2D automaton and then apply it to 2D images. A synchronous implementation of the watershed algorithm on the GPU is presented in section 4.1. Then we develop a more efficient block-asynchronous GPU algorithm in Section 4.2. As a consequence of the computation by blocks, the block-asynchronous watershed algorithm presents some undesirable artifacts that are corrected in the algorithm proposed in Section 4.3, at the cost of increasing the computational load. Finally, in Section 4.4 the extension of the method on a 3D automaton developed for the application over 3D volumes is presented.

4.1. Synchronous CA-watershed implementation

The GPU synchronous algorithm has two stages: one for initializing and another for updating the automaton. The $\mathcal{N}^{F}(u)$ and the set of neighbors with the same grey value, $\mathcal{N}^{=}(u)$, are computed for each pixel in the first stage. In the second stage, the updates flood each region with a representative label in an

iterative process, with a global synchronization at each step. We have used 4-neighbor connectivity.

With the objective of increasing the data locality, we pack the information required for each pixel into 64 bits. The minimum amount of data required are 4 bytes for $l(u)$, 1 byte for $h(u)$, 1 byte for $\mathcal{N}^{=}(u)$ and 2 bytes for $\mathcal{N}^{F}(u)$. It is not necessary to store the state of each pixel as it can be deduced from $\mathcal{N}^{F}(u)$. If the lower slope is empty, the state is MP; otherwise, the state is NM. Fig. 3 (a) shows an example of data packing for one pixel. The set $\mathcal{N}^{=}(u)$ is compressed in 1 byte using 1 bit for each neighbor (L, R, U, D in Fig. 3 (b)) where "1" means a neighbor with the same grey level and "0" means a neighbor with a different one. The four least significant bits are ignored but they may be used to connect up to 8 neighbors, the maximum possible number for a two dimensional automata. $\mathcal{N}^{F}(u)$ is stored as an offset relative to the position of pixel u in memory, considering that the image is stored in row-major order. Its possible values are ± 1 and $\pm w$, as shown in Fig. 3 (c), being w the width of the image.

The algorithm consists of two kernels which implement the initialization and updating stages, respectively. These kernels are configured to work in 16×16 thread blocks with a thread operating on one pixel. With the first kernel the automaton is initialized according to Eq. (1) and Eq. (2). The grey values are read from texture memory as this read-only memory speeds up the accesses to data when they present high spatial locality. Once all data have been initialized, they are packed into 64 bits before being transferred to global memory. At the end of the initialization stage the state of each pixel, a cell of the automaton, has switched to NM or MP. In order to update all the pixels synchronously, we use two 64-bit buffers, one *input buffer* for reading data and one *output buffer* for writing the results.

The updating stage has been implemented through a loop executed by the CPU, which calls a CUDA kernel at each step, so there is one global synchronization per step. The pseudo-code shown in Algorithm 2 describes this iterative process. In each call to the kernel, data are read from the input buffer in global memory and are unpacked in registers. The pixels are updated once as described by Eq. (3) and Eq. (4) if their state is MP, and by Eq. (5) if their state is NM. Finally, the resulting data are packed and stored in the output buffer. The input and output buffers are swapped before the next iteration. Only one flag needs to be moved to the CPU at each inter-block iteration indicating whether a pixel must be further processed. The update ends when all regions have been flooded.

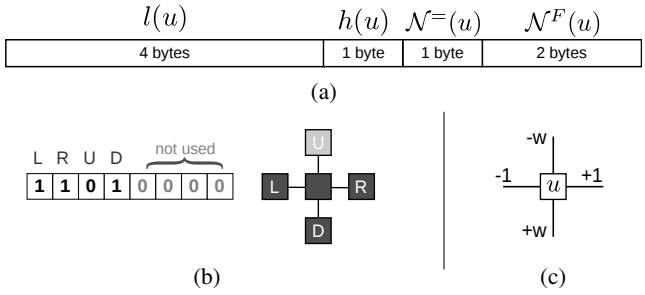

Figure 3: Data structures for one pixel. (a) Data packed into 64 bits, (b) structure of the variable $\mathcal{N}^{=}(u)$ and (c) possible values for the variable $\mathcal{N}^{F}(u)$.

Algorithm 2: CA-watershed synchronous algorithm – Inter-block iterative updating

```
Host code
do (inter-block updating)
   CUDAKernel( i_buffer, o_buffer)
   Global synchronization among blocks
   swapBuffers( i_buffer, o_buffer)
while new updates
```

```
CUDAKernel( i_buffer, o_buffer )
   loadToRegisters( i_buffer)
   Updating according to Eqs. (3) – (5)
   storeResults( o_buffer)
```

4.2. Block-asynchronous CA-watershed algorithm

In this section we explain the block-asynchronous algorithm, which has the advantage of reusing information within a block, unlike the synchronous implementation, efficiently exploiting the shared and cache memories of the GPU. By block-asynchronous, we mean updating a group of pixels to their next state an unbounded number of times without a global synchronization. Thus, the image can be partitioned into different regions which can be updated independently. Thus, each region is updated asynchronously with respect to other regions.

This algorithm has two kernels configured to work in blocks of 16×16 threads operating on 16×16 pixel regions of the image.

The storage requirements are the same as for the synchronous implementation, but in this case, for the variable $\mathcal{N}^{F}(u)$, the value w represents the width of the region of the image processed within the block. The initialization stage is also the same.

In this block-asynchronous algorithm the updating stage has been adapted to perform in shared memory as many updates inside a region as possible (called *intra-block* updates) before performing a synchronization among thread blocks which we call *inter-block* updates. Each region is synchronously updated (i.e. all cells within a region are updated at each time step), while the regions themselves are asynchronously updated (an update of the entire grid is performed at some selected steps). Hence, this is a hybrid iterative process that includes asynchronous intra-block updates and synchronous inter-block updates.

This model is shown in Fig. 4. During the intra-block updating the values used from outside the block are kept constant (equal to their values at the beginning of the stage). In the inter-block updating process after a global synchronization, data are read at the block boundaries, which allows the propagation of data across the entire grid.

The pseudo-code shown in Algorithm 3 describes the iterative process of the updating stage. The inter-block updating loop is executed in the CPU and calls the updating kernel that is executed in the GPU. In the kernel, for each block, once data

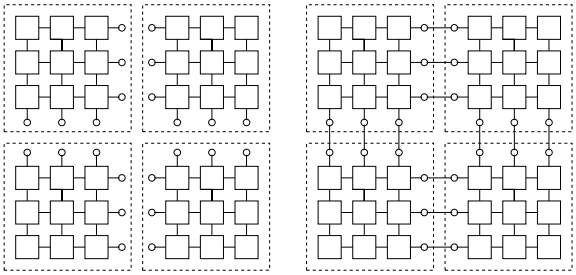

Figure 4: Cellular automata with 4-connectivity and 4 blocks: intra-block updating (left) and inter-block updating (right).

are loaded in shared memory, the pixels are modified according to Eqs. (3) – (5) in an iterative intra-block process within each region of the image. Data are updated in registers and stored back to shared memory. Threads within a block are synchronized locally at each step of the intra-block process, so data updated within a block can be reused from the shared memory, which is much faster than the global memory space [17].

Algorithm 3: CA-watershed block-asynchronous algorithm – Inter- and intra-block iterative updating

Host code
do (inter-block updating)
  CUDAKernel(*i_buffer*, *o_buffer*)
  Global synchronization among blocks
  swapBuffers(*i_buffer*, *o_buffer*)
while new updates

CUDAKernel(*i_buffer*, *o_buffer*)
loadToSharedMemory(*i_buffer*)
do (intra-block updating)
  Updating according to Eqs. (3) – (5)
  Local synchronization among threads
while new updates
storeResults(*o_buffer*)

The intra-block updating ends when no new modifications are made with the available data within the region. Then the data in shared memory are packed and stored in global memory in the output buffer and the input and output buffers are swapped. The updating stage ends when all regions have been flooded.

In order to update the pixels at the edge of the block in this block-asynchronous implementation, the shared memory allocated for each region must be extended with a border of size one. Thus, the border of one region overlaps the adjacent regions. Fig. 5 shows an image divided into regions of 4×4 pixels (left) and the shared memory allocated (right). Threads on the edge of the block have to do extra work loading the data of the border.

4.3. Artifact-free block-asynchronous watershed algorithm

In the previous section we developed a block-asynchronous algorithm for computing the watershed transform based on a

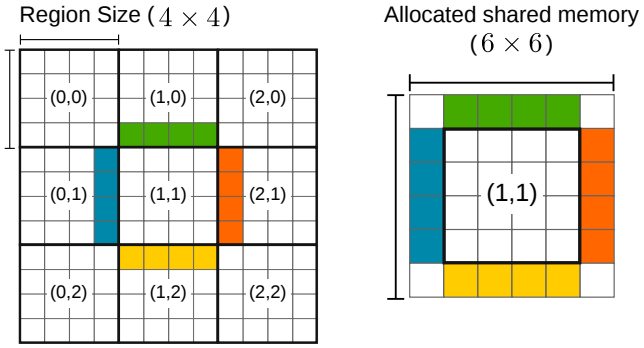

Figure 5: An image divided in regions of 4×4 pixels (left) and the extended shared memory allocated for one region of the image (right).

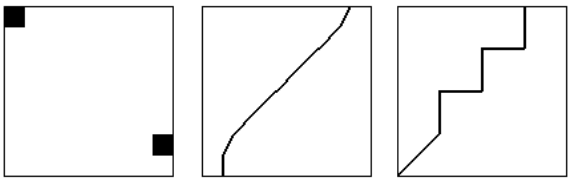

Figure 6: An image of size 128×128 pixels (left), the correct watershed line obtained using 4-connectivity (middle), and the artifacts produced by the asynchronous computation using blocks of cells of size 32×32 (right).

2D cellular automaton [16]. This algorithm follows a computational model in which the grid of cells of the automaton is partitioned into regular regions that are assigned to blocks of threads of the GPU. The block-asynchronous algorithm avoids points of global synchronization that are costly in execution time and efficiently exploits the shared memory, which has lower access times than the global memory. The algorithm obtains a correct segmentation according to the watershed segmentation definition. Thus, when non-minimum plateaus exist in the image, the algorithm gives a correct segmentation; nevertheless, the watershed lines may not match the geodesic distance properly. This situation is visually observed as small irregularities in the watershed lines.

When the data propagation speed is similar for all the cells, the algorithm may give a good approximation of the watershed lines. However, when computed by regions, it presents the problem of data propagation at the region boundaries, which causes artifacts as shown in the example in Fig. 6. Initially data are propagated within the region during the intra-block updating, and later this is performed at the region boundaries during the inter-block updating. The different speed of data propagation in the intra-block (one update per cell is performed at each time step) and inter-block (one update of the boundaries of the region is performed each several steps) updates result in improperly placed watershed lines.

The origin of this problem can be shown by introducing a variable which measures the propagation distance of data from the slopes of the image, as shown in Fig. 7. The region boundaries delay the data propagation. However, this problem can be solved by data correction from the information provided by the inter-block updating process and the measured distances. Accordingly, a procedure for performing this is required.

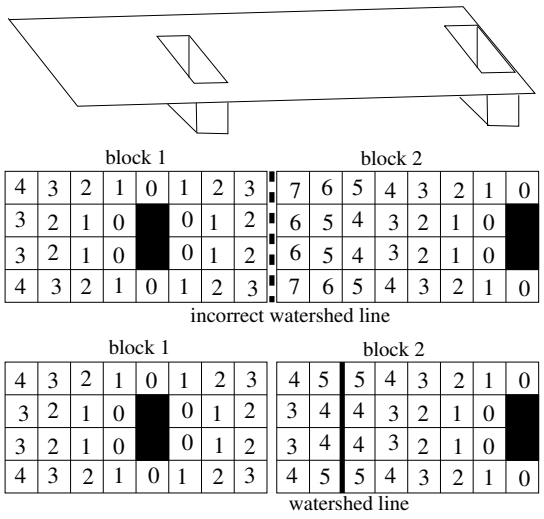

Figure 7: An image of 4×16 pixels displayed as a topographical relief (top), map of distances when the watershed is computed using asynchronous blocks (middle), and map of corrected distances (bottom).

In order to correct the artifacts produced by the asynchronous computation, we propose incorporating the wavefront technique into the algorithm, in a similar way to how it was introduced by Moga et. al [6]. It is necessary to define a distance variable, $d(u)$, that is initiated as,

$$d(u) = \begin{cases} \infty & \text{if } N^F(u) = \emptyset \\ 0 & \text{otherwise,} \end{cases} \quad (6)$$

i.e., the pixels in the neighborhood of a lower border of a plateau are assigned a distance of 0.

Then an iterative process for updating the automaton starts. The distances of pixels in the "MP" state are computed by increasing the distance from a neighbor by one. Thus, at the same time as the data are propagated over the plateau, the distances are computed by the pixels. However, the distances obtained during the intra-block updating can take incorrect values, as shown in Fig. 7. If a pixel is switched to the "NM" state, its distance and its label might need to be corrected, as the pixel could be part of a non-minimum plateau which should be split between two basins. The decision is taken by comparing the variable d of the pixel to those of its neighbors, and in the event that the difference is bigger than one, it must be corrected.

This technique provides watershed lines that match the geodesic distance properly. In Fig. 7, two slopes are symmetrically placed at both sides of a non-minimum plateau that are separated by an even number of pixels; hence, in this case only one solution is possible.

The original CA-watershed algorithm is therefore modified as shown below in Algorithm 4, where we have marked in bold the modifications over the initial Algorithm 1.

Algorithm 4: Artifact-free CA-watershed

Input: $current\_state(u)$, $f(v)$, $\mathbf{d}(\mathbf{v})$ from the neighbors
Output: $next\_state(u)$, $f(u)$, $\mathbf{d}(\mathbf{u})$
Case ($current\_state$ = "INIT")
  Compute $N^=(u)$, $N^F(u)$, $\mathbf{d}(\mathbf{u})$
  If ($N^F(u) = \emptyset$)
  { $l(u) \leftarrow \min_{v \in N^=(u)}(l(v))$; $next\_state(u) \leftarrow$ "MP"; }
  Else
  { $f(u) \leftarrow f(N^F(u))$; $next\_state(u) \leftarrow$ "NM"; }
Case ($current\_state$ = "MP")
  For any neighbor $v \in N^=(u)$:
  If ($h(v) < h(u)$)
  { $N^F(u) = \{v\}$; $f(u) \leftarrow f(v)$; $\mathbf{d}(\mathbf{u}) = \mathbf{d}(\mathbf{v}) + \mathbf{1}$;
    $next\_state(u) \leftarrow$ "NM"; }
  Else
  { $l(u) \leftarrow \min(l(u), l(v))$; }
Case ($current\_state$ = "NM")
  For its flooding neighbor $v = N^F(u)$:
  $f(u) \leftarrow f(v)$;
  For any neighbor $\mathbf{v} \in \mathbf{N}^=(\mathbf{u})$:
  If $(\mathbf{d}(\mathbf{u}) - \mathbf{d}(\mathbf{v}) > \mathbf{1})$
  { $\mathbf{l}(\mathbf{u}) \leftarrow \mathbf{l}(\mathbf{v})$; $\mathbf{N^F}(\mathbf{u}) = \{\mathbf{v}\}$; $\mathbf{d}(\mathbf{u}) = \mathbf{d}(\mathbf{v}) + \mathbf{1}$; }

Regarding the implementation on GPU, the storage requirements are also similar to those for the previous GPU proposals, but an additional double buffer of integers is also required for storing the distances $d(u)$. The buffer is initialized in the first stage and it must be moved between shared and global memory at each inter-block updating. The amount of data per pixel are two buffers of 8 bytes of packed information (see Fig. 3) and other two buffers of integers to store the distance values.

4.4. Asynchronous watershed proposal based on a 3D cellular automaton

As explained in Section 2.2 the cells of an automaton may be arranged in three dimensions in order to process a 3D volume. In this case the connectivity of the automaton needs to be adapted to connect a cell to its surrounding neighbors, which may range from a minimum of 6 (connecting a cell to its left, right, top, bottom, forward and backward neighbors) up to a maximum of twenty six neighbors.

The previous 2D watershed proposals in GPU can be easily adapted to process a 3D volume of data, especially when only a low number of neighbors is considered. In this section we describe the changes made to the previous 2D asynchronous artifact-free algorithm described in Section 4.3 to process a volume. The new 3D implementation also consists of two kernels which are configured to work in blocks of $8 \times 8 \times 4$ threads, with each one operating on a different 3D region in the volume. In the first stage the input data are read from global memory and the cells are initialized by accessing to their neighbors. The hybrid iterative process described in Section 4.2 performs the inter-block and intra-block updates in the second stage.

The same memory requirements per pixel as for the 2D artifact-free asynchronous algorithm can be applied in this case. If we consider a 6-neighbour connectivity, the data required in global memory per each voxel can be compressed in 8 bytes, as explained in Section 4.1 and showed in Fig. 3 including one byte that stores the connectivity allowing to store up to 8 neighbors. The storage requirements per voxel are two buffers (input and output buffers) of 8 bytes of global memory for the packed

data and two additional buffers of 4 bytes each storing the distance value.

The shared memory allocated for each 3D region is extended with a border of size one in each dimension, so the borders with the adjacent regions are overlapped in the same way as in the 2D case. So, compared to the 2D algorithm, the shared memory requirements are much higher. More resources in terms of registers are also required as it will be detailed in Section 5.4.

5. Results

We have evaluated our proposals on a PC with an Intel Core i7 with four cores at 2.80 GHz and 8 GB of RAM. Each core has separate L1 caches for instructions and data, and a unified L2 cache. The unified L3 cache is common to all the cores, as shown in Table 1. The GPU is a NVIDIA GeForce GTX580, consisting of 16 SMs, each one with 32 SPs. The GPU memory size is 1536 MB and its cache architecture consists of a unified L1 cache per SM and a L2 unified cache of 768 KB shared by all the SMs. The L1 cache of the GPU can be configured as 16 KB, being in this case 48 KB the size of the shared memory, indicated in Table 1 as "L1 opt1", or vice versa, indicated as "L1 opt2". We have chosen one configuration or the other depending on the shared memory requirements of each GPU proposal. The code has been compiled using the gcc version 4.4.3 with OpenMP 3.0 support under Linux for the CPU implementation and using the nvcc and the toolkit 4.0, also under Linux, for the GPU implementations.

The performance results analyzed are expressed in terms of execution times and speedups. The execution times were obtained as the average of twenty executions. The speedups were calculated with respect to the best OpenMP multithreaded parallel implementation of the synchronous cellular automaton. The OpenMP code is optimized by using four threads for running on the four available cores of the CPU and the work scheduling is static in order to evenly distribute the workload among the threads and to achieve a high locality in the data accesses. Therefore, the 2D image or the 3D volume is divided into four consecutive horizontal strips or volumes and each thread processes one of them. The need to access data outside the region assigned to each thread is not a problem in the OpenMP implementation, as all the threads access the same memory space. The algorithm includes a synchronization barrier at each step of the updating stage.

Table 1: CPU and GPU cache memory hierarchies of the test platforms.

	CPU Intel Core i7	
L1	64 KB × 4 cores =	256 KB
L2	256 KB × 4 cores =	1024 KB
L3		8192 KB
	GPU GTX580	
L1 opt1	16 KB × 16 SMs =	256 KB
L1 opt2	48 KB × 16 SMs =	768 KB
L2		768 KB
L3		–

Table 2: CPU–GPU data transfer times.

Image size	512^2	1024^2	2048^2
Transfer time	0.0022s	0.0082s	0.0321s

The tests executing the GPU proposals were run on two images and one volume at different resolutions. The CPU–GPU data transfers are carried out at the beginning, for transferring the image to the GPU, and at the end, for sending the results back to the CPU.

In the following sections we analyze the different GPU proposals for the CA-watershed: synchronous, block-asynchronous and artifact-free block-asynchronous. First in Section 5.1 we have checked the correctness by comparing the number of segmented regions obtained by the GPU algorithms over 2D images to the number of regions obtained by a sequential watershed algorithm. Then the performance is analyzed in terms of executions times and speedups for 2D images. The main differences in performance among the synchronous and the block-asynchronous algorithms are explained in Section 5.2 mainly by inspecting the number of synchronization points required by the different proposals. We study the artifact-free block-asynchronous results in detail in Section 5.3 in a similar way as in Section 5.1. Section 5.4 describes the performance results obtained for a 3D implementation of the artifact-free block-asynchronous algorithm. Finally, we compare the results to other works in Section 5.5.

5.1. GPU performance results for 2D images

This section describes the performance results obtained by the three different GPU proposals when they are applied over 2D images of different sizes.

First, we have evaluated and compared the synchronous implementation and the two block-asynchronous proposals in the GPU in terms of execution times and speedups. For the GPU tests we have measured execution times calculated as the sum of the data transfer times between CPU and GPU and the computation times.

Table 2 shows the data transfer times in seconds for the different image resolutions used in our experiments. The test images used were Lena and a computed tomography scan of a human head (CT Scan), as representatives of processing small and large plateaus, respectively (see Fig. 8 (left)). The processing of large plateaus (regions of uniform grey values) makes it necessary to propagate the labels through large regions of the image. This requires more computation time than processing small plateaus. Thus the selected images represent two very different cases regarding to computational cost of the watershed.

Fig. 8 (right) shows the result of applying watershed to the original images without any additional processing, although, when the watershed is applied to a particular image domain, it is usual to previously apply techniques, such as a gradient

Table 3: Number of regions generated by the algorithms using Lena and the CT Scan of a human head.

Image size	512^2	1024^2	2048^2
Lena	24958	25139	28521
CT Scan	6221	7300	13381

function, to enhance image edges, and a denoise filter or markers selection to reduce the over-segmentation [3, 4]. We have validated our segmentation results by comparing the number of segmented regions obtained by a sequential watershed algorithm with the number of regions generated by the GPU implementations. All the proposals obtain the same number of regions, as shown in Table 3. The difference between the number of regions at different resolutions is due to the process of scaling the image. The CT Scan image presents large plateaus and therefore the number of regions is lower than for the Lena image.

For the GPU proposals, the L1 cache is maximized to 48 KB, with the remaining 16 KB being for the shared memory corresponding this configuration to "L1 opt2" in Table 1. The reasons for selecting this configuration are: in the GPU synchronous implementation because shared memory is not used, and in the block-asynchronous proposals because only 15 KB of shared memory are required for the 6 blocks that are simultaneously active per SM. As described in Section 4.1, 8 bytes per pixel are required in the case of the block-asynchronous proposal. For a 16×16 thread block, the shared memory required has been extended with a border of size one, so the algorithm needs $18 \times 18 \times 8$ bytes of shared memory, i.e. 2.5 KB per block. Therefore, considering 6 blocks per SM, a total of 15 KB of shared memory per SM are used.

For the case of the artifact-free asynchronous proposal the number of simultaneously active blocks per SM is reduced to 4. In this case the limiting factor is the number of 32768 registers available per SM. The proposal requires 26 registers per thread, which gives a total of $16 \times 16 \times 26 = 6656$ registers per block, so there are enough registers in each SM for only 4 blocks. Regarding shared memory use, each pixel requires 12 bytes in this proposal, so $18 \times 18 \times 12 = 3888$ bytes per block are required, which gives a total of 15.2 KB of shared memory required for the 4 blocks in each SM, so the configuration "L1 opt2", i.e. 16 KB of shared memory are sufficient.

Table 4 shows a summary of the performance results obtained. The speedups of the GPU proposals considering only computation times are displayed in Fig. 9. For both images all GPU proposals obtain high speedups for all the image sizes. As shown in Fig. 9, the speedups also scale well with the size of the image; i.e. from 20.4x and 17.5x for the synchronous implementation for the Lena and CT Scan 512×512 images, respectively, up to 51.7x and 115.5x with the block-asynchronous proposal when the 2048×2048 images are processed. When the image size increases, so does the amount of computational work, thus the hundreds of available threads are better exploited. The artifact-free block-asynchronous proposal also obtains better speedups than the synchronous one: 30x and 74.3x for the Lena and CT Scan 2048×2048 images.

5.2. Block-asynchronous approach versus synchronous approach

In this section we analyze the reason why the block-asynchronous performance results are better than for the synchronous approach. The discussion is based on analyzing the

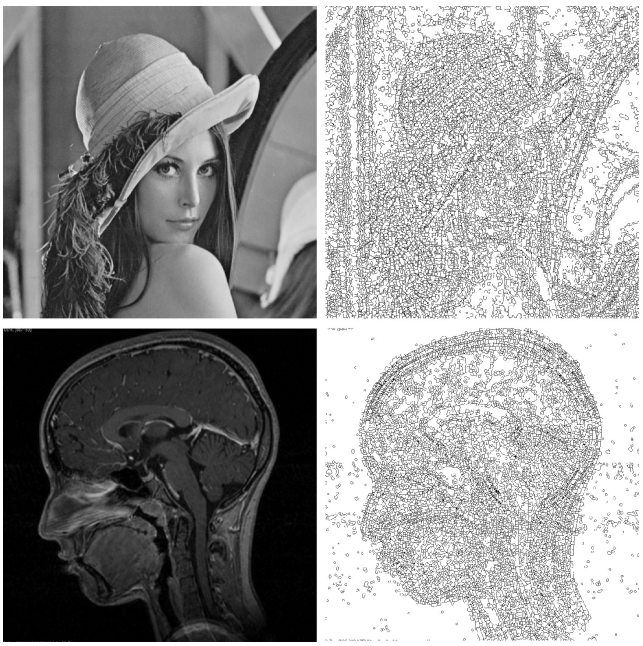

Figure 8: Images used in the tests and watershed results obtained.

Table 4: Performance results for 2D Lena and CT Scan images including data transfer times

Lena	512^2	1024^2	2048^2
time			
CPU OpenMP	0.0351s	0.1990s	1.2452s
GPU Sync.	0.0039s	0.0188s	0.0916s
GPU Async.	**0.0030s**	**0.0131s**	**0.0562s**
GPU A-F Async.	0.0034s	0.0158s	0.0736s
speedup			
GPU Sync.	9.0x	10.6x	13.6x
GPU Async.	**11.7x**	**15.2x**	**22.2x**
GPU A-F Async.	10.3x	12.6x	16.9x
CT Scan	512^2	1024^2	2048^2
time			
CPU OpenMP	0.4941s	2.8793s	15.0919s
GPU Sync.	0.0305s	0.1436s	0.6992s
GPU Async.	**0.0093s**	**0.0381s**	**0.1628s**
GPU A-F Async.	0.0126s	0.0522s	0.2353s
speedup			
GPU Sync.	16.2x	20.1x	21.6x
GPU Async.	**53.1x**	**75.6x**	**92.7x**
GPU A-F Async.	39.2x	55.2x	64.1x

characteristics of the two algorithms, the test images in number and size of the plateaus and on the number of global and local synchronizations required.

We focus the test on the images at a resolution of 2048×2048 pixels as the behaviour of processing large plateaus is better appreciated in this case. When the image presents large plateaus the computational cost of the watershed transform increases because the labels must be propagated through large regions of the image. Comparing the execution times of the synchronous and block-asynchronous approaches (see Table 4), a speedup of 1.6x is obtained for the image of Lena while the speedup increases up to 4.3x for the CT Scan image. The improvement of the block-asynchronous proposal versus the synchronous implementation is better for the second image; although, as shown in Table 4, processing large plateaus takes more time: 0.1628s for the CT Scan image while the Lena image only requires 0.0562s. The reason is that for the block-asynchronous proposal the intra-block updating allows the labels to propagate faster among regions, especially in images with large plateaus. If a region is entirely within a plateau, the labels have to be propagated from side to side of that region. In this situation, only one inter-block update and w intra-block updates are needed, where w is the width of the region. The synchronous implementation would need w inter-block updates, with the consequent penalty for transferring data from and to global memory at each step, with each one of those steps corresponding with a global synchronization.

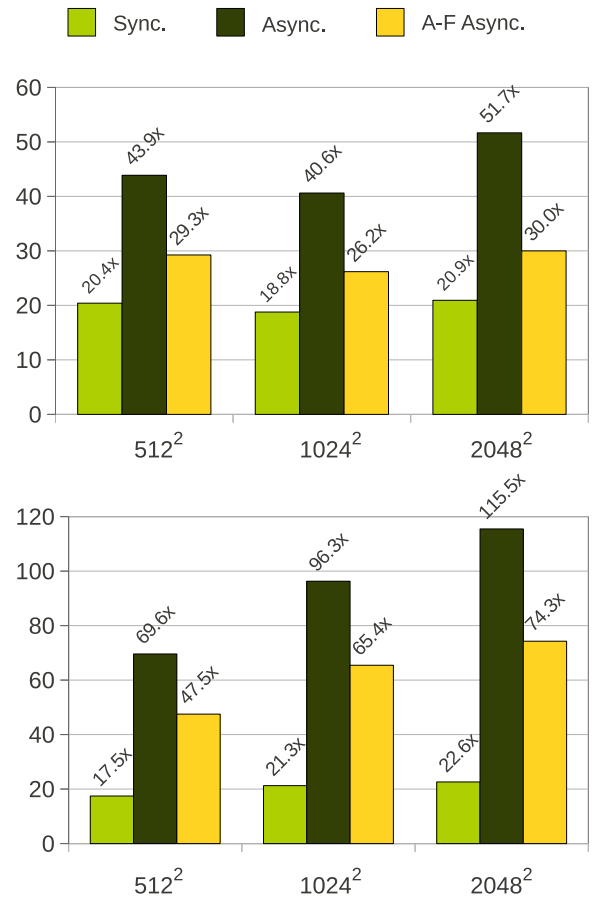

Figure 9: Speedup of the GPU proposals using the image of Lena (top) and a the CT Scan (bottom), not including data transfer times.

Table 5: Number of updates per pixel for the different GPU implementations for 2048 × 2048 images.

Lena	GPU Sync.	GPU Async.
inter-block	114	16
intra-block min.	—	22
intra-block max.	—	195
intra-block avg.	—	108.5
CT Scan	GPU Sync.	GPU Async.
inter-block	1156	76
intra-block min.	—	88
intra-block max.	—	1248
intra-block avg.	—	668

The block-asynchronous approach reduces the number of synchronizations among thread blocks and increases data reuse thanks to the inter- and intra-block updating scheme.

The decrease in the number of synchronizations for the block-asynchronous proposals is illustrated in Table 5, where the number of inter-block and intra-block updates are summarized for the synchronous and the block-asynchronous implementations and the test images. Only the values for the block-asynchronous implementation are shown as the numbers are the same for the artifact-free proposal. For the synchronous implementations (in CPU as in GPU) only inter-block updates take place in the sense that after each update of all the pixels of the image one global synchronization operation is required. Observing, for example, the values for the CT Scan image in the table, the number of inter-block updates (i.e. the number of global synchronizations required) is 1156 for the synchronous implementation. For the block-asynchronous cases the number of inter-block updates decreases to 76 and the total number of asynchronous intra-block updates per block summing up all the iterations ranges from 88 to 1248, depending on the block, with 668 being the average value over all the blocks. Hence, the number of updates per pixel is 1156, with the same number of corresponding global synchronizations for the synchronous implementation, and an average of 668 local synchronizations with only 76 global synchronizations for the block-asynchronous algorithms. In the case of the Lena image, a similar decrease is observed.

The block-asynchronous proposal also exploits the resources available in the GPU better, using the shared memory, which is faster than the global memory.

5.3. Artifact-free block-asynchronous results

The artifact-free block-asynchronous algorithm is the best GPU solution for computing the CA-watershed as it obtains the most correct watershed lines as well as greater speedups. In this section, details of its GPU implementation are discussed and the performance results are analyzed in detail by inspecting the time-breakdown for the different test images.

The artifact-free block-asynchronous algorithm gives the correct watershed solution at the cost of increasing the computational cost by adding computations that include conditionals in the cases where non-minimum plateaus are present. In addition, this implementation requires an additional double buffer for sorting the distances, as described in Section 4.3. The speedups over the synchronous version are high, as shown in Fig. 9, but lower than for the plain block-asynchronous proposal. The speedups for Lena and CT Scan 2048×2048 images are 30x and 74.3x, while for the block-asynchronous proposal the values were 51.7x and 115.5x, respectively. The main reason for the bigger speedups for the block-asynchronous implementation is that, as a consequence of the number of registers required, as we have explained in Section 5.1, the number of active blocks per SM are 4 for the artifact-free block-asynchronous implementation and 6 in the case of the plain block-asynchronous one. The ratio in the number of concurrent blocks per SM is therefore $6/4 = 1.5$, which is approximately the ratio observed in the execution times; i.e. the block-asynchronous approach is 1.5 times faster than the artifact-free one. In addition, the artifact-free block-asynchronous proposal uses two buffers for the distances which need to be loaded from global to shared memory at each inter-block update. Thus, the overhead is mainly introduced by the hardware limitations of the GPU. That is, if it could allocate up to six concurrent blocks per SM the performance would be higher.

Fig. 10 shows in detail a breakdown of the execution times and the time percentages corresponding to data transfer and to computation for the artifact-free block-asynchronous proposal with the CT Scan image. The time percentage for computation increased in proportion to the image size, in particular, from 82.5% to 86.3% for resolutions of 512×512 and 2048×2048 pixels, respectively. A similar trend can also be observed in Fig. 11 for the Lena image. This is due to the fact that, while the data transfer time increases linearly by a factor roughly equal to the scaling factor of the image, the computation time increases by a factor proportional to the increase in the size of the image. This behaviour is related to the number of regions in the image which, as shown in Table 3, increases with the size of the images. It is interesting to note that, for the CT Scan image, computations take the greatest percentage of time as this image includes large plateaus that require the propagation of the labels through large regions of the image thus increasing the computation time.

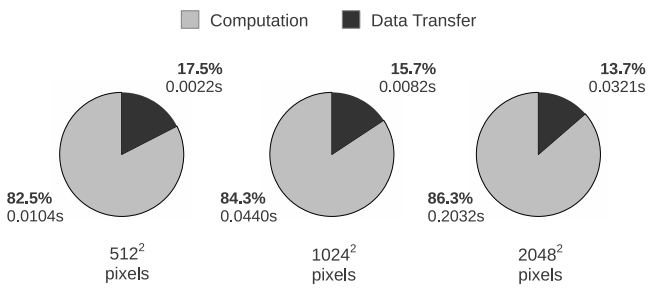

Figure 10: Time breakdown for the artifact-free block-asynchronous proposal when the CT scan image is processed.

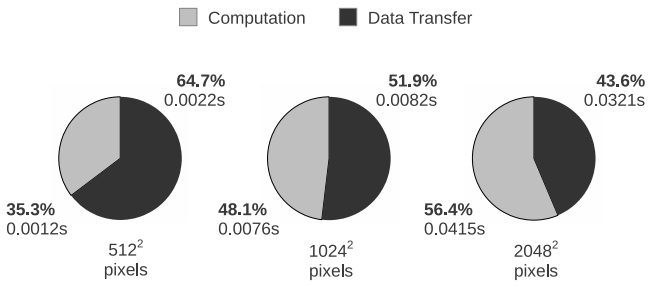

Figure 11: Time breakdown for the artifact-free block-asynchronous proposal when the Lena image is processed.

Table 6: CPU–GPU data transfer times for the 3D BrainWeb volumes.

Volume size	$45^2 \times 54$	$90^2 \times 108$	$181^2 \times 217$
Transfer time	0.0013s	0.0073s	0.0581s

5.4. GPU performance for 3D images

In this section results for the proposal based on a 3D cellular automata are analyzed. In order to assess the quality of our proposals we have used a volume downloaded from the BrainWeb Simulated Brain Database [24]. This database contains a set of realistic MRI data volumes produced by a MRI simulator. The aim of this analysis is to study the performance of the best of our GPU algorithms, the artifact-free block-asynchronous one, in a real environment with a heavier dataset. The test volume shown in Fig. 12 consists of a range of grey values representing the background and the region of interest, the grey matter. Noise or other features which may produce over-segmentation when the watershed is calculated are not present.

As in the case of the previous proposals, we have measured the execution times calculated as the sum of the data transfer times between CPU and GPU and the computation times. These data transfers, which are carried out at the beginning and the end of the codes, are represented in Table 6.

The correctness of the algorithm has been validated by comparing the number of segmented regions. The numbers are

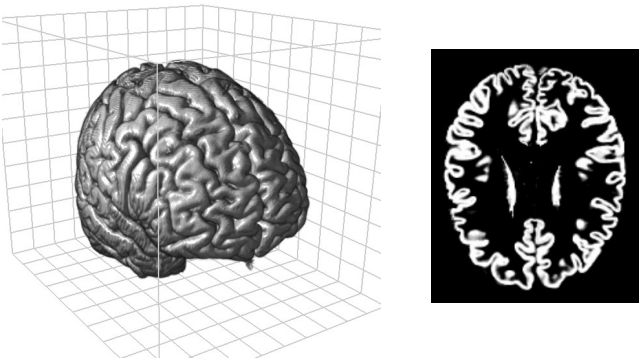

Figure 12: 3D volume used in the tests (left) and a section of the volume (right). Images produced using Voreen (voreen.uni-muenster.de)

Table 7: Number of regions generated by the watershed for 3D volumes from BrainWeb.

Volume size	$45^2 \times 54$	$90^2 \times 108$	$181^2 \times 217$
BrainWeb	1115	5669	14348

Table 8: Performance results for 3D BrainWeb volume including data transfer times.

BrainWeb	$45^2 \times 54$	$90^2 \times 108$	$181^2 \times 217$
time			
CPU OpenMP	0.1337s	2.2611s	37.7378s
GPU Sync.	0.0084s	0.0820s	1.1227s
GPU Async.	**0.0044s**	**0.0451s**	**0.5907s**
GPU A-F Async.	0.0050s	0.0540s	0.7304s
speedup			
GPU Sync.	15.9x	27.6x	33.6x
GPU Async.	**30.4x**	**50.2x**	**63.9x**
GPU A-F Async.	26.5x	41.8x	51.7x

shown in Table 7. The 3D proposals for GPU obtain the same number of regions as the sequential version on CPU.

Before executing the algorithms, the shared memory/L1 distribution of the GPU must be chosen. For this proposal the shared memory/L1 configuration is different from the case of the 2D images. In the 3D case only 4 blocks can be simultaneously active in each SM. The reason is that each SM provides a maximum of 32768 registers, and this proposal requires 32 registers per thread, where $8 \times 8 \times 4 = 256$ is the number of threads per block, and therefore 8192 the number of registers required per block. Regarding the shared memory, each block operates on $10 \times 10 \times 6 = 600$ voxels, which requires $600 \times 10 = 6000$ bytes of shared memory that include the distance values of the artifact-free block-asynchronous version. 24000 bytes are required for the 4 active blocks per SM; hence, the shared memory/L1 configuration must be 48 KB for shared memory and 16 KB for the L1 cache (case "L1 opt1" in Table 1).

Table 8 shows a summary of the performance results obtained in terms of execution times and speedups. As for the 2D case, all the 3D GPU proposals obtain speedups for all the volume sizes, and the speedup values increase with the volume size as the computational load also increases. The performance results for the block-asynchronous proposals are always better than for the synchronous implementation; approximately twice as good. The performance results for both, the block-asynchronous and the artifact-free block-asynchronous approaches are very similar.

5.5. Comparison to other works

Proposals of different watershed algorithms on the GPU using shaders [8] and CUDA [9–11] have been presented in the last few years. In [9] a new algorithm is presented based on the introduction of a chromatic function for establishing the order in which the voxels are processed. The experiments are carried out over volume data sets obtaining maximum speedups of 7x on a Nvidia GTX295 when compared to the sequential proposal of the algorithm, even when large volume data sets of up to $600 \times 600 \times 600$ are considered. In our case, the largest volume considered was $181 \times 217 \times 181$, 9 times smaller, achieving speedup values of around 64x on a GTX580. Taking into account that the speedups of our block-asynchronous proposals for 3D increase with the volume size, as shown in Table 6, and that our experiments have proved that the block-asynchronous proposals scale by a factor of 2x–4x among the GTX295 and the GTX580 GPUs, we can conclude that our proposals outperform the results in [9].

The algorithm presented in [10] is inspired by the drop of water paradigm and performs a component labelling and a path compression approach [25]. Its results are compared to a GPU synchronous algorithm for the watershed based on a CA [8] outperforming it. Given that the experiments in [10] are performed in a older GPU than in our case, we have executed them on our GTX580, obtaining similar speedup results to the obtained with our block-asynchronous proposal of the CA-watershed described in this paper.

The main advantage of our proposals is the simplicity of the CA computing model. They are simple and easy to understand because the CA present only three states and the projection over a regular structure of independent computing cores is immediate. Furthermore, our CA-watershed proposals for GPUs can easily be extended to introduce preprocessing or postprocessing steps, such as, for example, calculating the gradient or performing region merging.

6. Conclusions

A block-asynchronous strategy to compute the cellular automata based watershed on the GPU was studied. The implicit parallelism of CA consisting in independent cells that evolve following a set of rules, perfectly matches the computing model of modern GPUs. The block-asynchronous proposal relaxes the synchronization requirements thus exploiting the computing capabilities of GPUs to the maximum. Moreover, this implementation also matches the computing requirements of multi-CPU computers, and therefore it could be also adapted to these architectures.

As a consequence of the asynchronous computation by blocks, the watershed lines obtained do not match the geodesic distance properly. An artifact-free block-asynchronous algorithm that applies the wavefront method in order to correct this problem has also been proposed.

A drawback of the proposed algorithms is a small increase in the number of operations with respect to the original watershed algorithm, specially for the artifact-free proposal. Nevertheless, very good performance results are obtained, specially for images with large plateaus where the block-asynchronous approach is better exploited, achieving a maximum speedup of 115.5x at a resolution of 2048×2048 pixels.

In our future work, we plan to modify the CA-watershed to include other image operations, such as pre- or postprocessing stages. A multi-GPU implementation that can be efficiently applied to large multidimensional images is also among our objectives.

Acknowledgments

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, cofounded by the FEDER funds of European Union, under contract TIN 2010-17541, and by Xunta de Galicia, Program for Consolidation of Competitive Research Groups ref. 2010/28. Pablo acknowledges financial support from the Ministry of Science and Innovation, Government of Spain, under a MICINN-FPI grant.

References

[1] Vincent, L. and Soille, P., Watersheds in digital spaces: An efficient algorithm based on immersion simulations, IEEE Trans Pattern Anal Mach Intell, **13**(6) pp. 583–598, 1991.

[2] Meyer, F., Topographic distance and watershed lines, Signal Processing, **38**(1), pp. 113–125, 1994.

[3] Grau, V., Mewes, A.U.J., Alcaniz, M., Kikinis, R. and Warfield, S.K., Improved watershed transform for medical image segmentation using prior information, Medical Imaging, IEEE Trans. on, **23**(4), pp. 447–458, 2004.

[4] Meyer, F. and Beucher, S., Morphological segmentation, Journal of Visual Communication and Image Representation, **1**(1), pp. 21–46, 1990.

[5] Bieniek, A. and Moga, A connected component approach to the watershed segmentation. In Mathematical Morphology and its Applications to Image and Signal Processing, Heijmans, H. J. A. M. and Roerdink, J. B. T. M., Eds. Kluwer Acad. Publ., Dordrecht, pp. 215–222, 1998.

[6] Moga, A.N., Cramariuc, B., and Gabbouj, M., Parallel watershed transformation algorithms for image segmentation, Parallel Computing, **24**, pp. 1981-2001, 1998.

[7] Roerdink J. B. T. M. and Meijster A., The watershed transform: definitions, algorithms and parallelization strategies, Fundam. Inf., **41**(1), pp. 187–228, 2000

[8] Kauffmann, C. and Piche, N., Cellular automaton for ultra-fast watershed transform on gpu, in Proc. of the 19th Int. Conf. on Pattern Recognition, pp. 1–4, 2008.

[9] Wagner, B., Müller, P. and Haase, G., A Parallel Watershed-Transformation Algorithm for the GPU, in Proc. of the Workshop on App. of Discrete Geometry and Math. Morphology, pp. 111–115, 2010.

[10] Körbes, A., Vitor, G.B., Lotufo, R. and Ferreira, J.V., Advances on watershed processing on GPU architecture, in Mathematical Morphology and Its Applications to Image and Signal Processing, Springer, pp. 260-271, 2011.

[11] Hučko, M. and Šrámek, M., Streamed Watershed Transform on GPU for Processing of Large Volume Data, in Proceedings of Spring Conf. on Computer Graphics (SCCG), 2012.

[12] Chua, L.O., Yang, L., Cellular neural networks: applications, Circuits and Systems, IEEE Transactions on, **35**(10), pp 1273–1290, 1998.

[13] Priego, B., Souto, D., Bellas, F. and Duro, R. J., Unsupervised Segmentation of Hyperspectral Images through Evolved Cellular Automata, Advances in Knowledge-Based and Intelligent Information and Engineering Systems, **243**, pp. 2160–2169, 2012.

[14] Sipper, M., Tomassini, M. and Capcarrere, M., Evolving asynchronous and scalable cellular automata, in Proc. Int. Conf. on Artificial Neural Networks and Genetic Algorithms, Springer-Verlag, pp. 67-68, 1998.

[15] Kauffmann, C. and Piche, N., A cellular automaton framework for image processing on gpu, in Pattern Recognition, Peng-Yeng, Y., Ed. inTech, pp. 353–376, 2009.

[16] Galilée, B., Mamalet, F., Renaudin, M. and Coulon, P.-Y, Parallel asynchronous watershed algorithm-architecture, IEEE Trans. on Parallel and Distributed Systems, **18**(1), pp. 44–56, 2007.

[17] Kirk, D. B., and Wen-mei, W. H., Programming Massively Parallel Processors: A Hands-on Approach, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[18] Quesada-Barriuso, P., Heras, D.B. and Argüello, F., Efficient GPU asynchronous implementation of a watershed algorithm based on cellular automata, in Proc. IEEE Int. Symp. on Parallel and Distributed Processing with Applications, pp. 79-86, 2012.

[19] Quesada-Barriuso, P., Argüello, F. and Heras, D.B, Efficient segmentation of hyperspectral images on commodity GPUs, Advances in Knowledge-Based and Intelligent Information and Engineering Systems, **243** pp. 2130–2139, 2012.

[20] Sarkar, P. A brief history of cellular automata, ACM Computing Surveys (CSUR), **32**(1), pp. 80–107, 2000.

[21] Nehaniv, C. L., Evolution in asynchronous cellular automata, in Proc. of the eighth Int. Conf. on Artificial life, MIT Press, pp. 65–73, 2003.

[22] Anzt, H., Tomov, S., Dongarra, J. and Heuveline, V., A Block-Asynchronous Relaxation Method for Graphics Processing Units, Technical report, Innovative Computing Laboratory, University of Tennessee, UT-CS-11-687, 2011.

[23] Adachi, S., Peper, F. and Lee, J.: Computation by asynchronously updating cellular automata, J. Stat. Phys., **114** pp. 261-289, 2004.

[24] Cocosco, C.A., Kollokian V., Kwan, R.K.-S, Evans, A.C., BrainWeb: Online Interface to a 3D MRI Simulated Brain Database, in NeuroImage, Proceedings of 3-rd International Conference on Functional Mapping of the Human Brain, **5**(4) part 2/4, S425 1997.

[25] Hawick, K. A., Leist, A. and Playne, D. P., Parallel graph component labelling with GPUs and CUDA, Parallel Computing, **36**(12), pp. 655–678, 2010.

Pablo Quesada-Barriuso received his B.S. in Computer Science and his M.S. in Graphics, Games and Virtual Reality in 2007 and 2010 respectively. He joined the Computer Architecture Group of the University of Santiago de Compostela as a research assistant, and he currently pursuing a Ph.D. in Computer Science. His main research interests include image processing, parallel algorithms and GPUs.

Dora B. Heras received a M.S. degree in Physics in 1994 and a Ph.D. in 2000. She is currently an Associate Professor in the Department of Electronics and Computer Engineering at the same University. Her research interests include parallel and distributed computing, software optimization techniques for emerging architectures, computer graphics for high performance computing and image processing.

Francisco Argüello received the B.S. and Ph.D. degrees in Physics from the University of Santiago, Spain in 1988 and 1992, respectively. He is currently an associate professor in the Department of Electronic and Computer Engineering at the University of Santiago, Spain. His current research interests include signal and image processing, computer graphics, parallel and distributed computing, and quantum computing.