Analysis of a Step-Based Watershed Algorithm Using CUDA

Giovani Bernardes Vitor, André Körbes, Roberto de Alencar Lotufo, Janito Vaqueiro Ferreira

Analysis of a step-based watershed algorithm using CUDA

Giovani Bernardes Vitor [1]
giovani@fem.unicamp.br
André Körbes [2]
korbes@dca.fee.unicamp.br
Roberto de Alencar Lotufo [2]
lotufo@unicamp.br
Janito Vaqueiro Ferreira [1]
janito@fem.unicamp.br

[1] Universidade Estadual de Campinas
Faculdade de Engenharia Mecânica
DMC - Departamento de Mecânica Computacional
Rua Mendeleyev, 200
CP 6122 - Campinas, SP, Brasil

[2] Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
DCA - Departamento de Engenharia de Computacão e Automação Industrial
Av. Albert Einstein - 400
CP 6101 - Campinas, SP, Brasil

Abstract

This paper proposes and develops a parallel algorithm for the watershed transform, with application on graphics hardware. The existing proposals are discussed and its aspects briefly analysed. The algorithm is proposed as a procedure of four steps, where each step performs a task using different approaches, inspired by the existing techniques that best adhere to the problem addressed. The algorithm is implemented using the CUDA libraries and its performance is measured on the GPU and compared to a sequential algorithm running on the CPU, achieving an average speedup of two times the execution time of the sequential approach. This work improves on previous results of hybrid approaches and parallel algorithms with many steps of synchronisation and iterations between CPU and GPU.

1. Introduction

The identification of objects on images needs in most cases a pre-processing step, with algorithms based on segmentation by discontinuity or the opposite, by similarity. Segmentation itself is not a trivial task, being among the hardest ones in image processing (Gonzalez & Woods, 2002). Some of its inherent problems are illumination variation through image sequences, dynamic change of background where the camera and/or target move, as well as changes on the topology of the target or its partial or full occlusion, resulting on a higher complexity of the scene and therefore a more difficult problem.

The watershed transform is a useful tool that is a part of some segmentation processes, for tasks of region labelling. Since its introduction by Digabel & Lantuéjoul (1978) constant efforts have been made to improve its performance, in sequential architectures such as the first fast algorithm by Vincent & Soille (1991) until the recent ones by Osma-Ruiz et al. (2007) and Cousty et al. (2009), as well as in parallel, such as the framework developed by Moga et al. (1994) until the algorithm of Galilée et al. (2007). These efforts, combined with the recent dissemination of out of the box SIMD (Single Instruction Multiple Data) parallel architectures through GPUs (Graphics Process Unit) and the languages and libraries that ease its development, suggest a new approach on research for accelerating the watershed implementations.

This paper aims for the development of a watershed transform algorithm suited for GPU processing, developed with NVIDIA’s CUDA (Compute Unified Device Architecture) platform, in four major steps. A fully parallel SIMD algorithm is proposed, taking advantage of the massive parallelism using specialised algorithms for each necessary step. The proposed algorithm is tested on a set of images and compared to the faster sequential algorithm that the authors built, based on the works by Lin et al. (2006) and Osma-Ruiz et al. (2007), that is very similar in function to the algorithm proposed.

The paper is organised as follows: Sec. 2 revises the state of the art on the watershed transform algorithms both on sequential and parallel fields. Sec. 3 goes through the implementation and architecture of GPGPU (General-Purpose computation on Graphics Processing Units) processing with the CUDA libraries. Sec. 4 exposes the algorithm developed, explaining its operation. Sec. 5 presents the timing results obtained, comparing the sequential and parallel algorithms. Lastly, Sec. 6 depicts conclusions obtained with this work.

2. Watershed Transform

Taking different approaches, the watershed transform is defined diversely on the literature, each of which producing a different set of solutions (Vincent & Soille, 1991; Meyer, 1994; Lotufo & Falcão, 2000; Bieniek & Moga, 1998; Cousty et al., 2009). The definitions are based on regional or global elements, such as influence zones and shortest-path forests with max or sum of weights of edges, or on local elements, such as the steepest descent paths, where the neighbour’s information is used to create a path to the corresponding minimum. These definitions are thoroughly revised on the literature (Audigier & Lotufo, 2007; Roerdink & Meijster, 2000; Cousty et al., 2009). In this work the local condition definition is used, henceforth named LC-WT (Local Condition Watershed Transform).

The LC-WT definition was introduced by Bieniek et al. (1997) with the purpose of achieving speedups on the parallel watershed transform, once it mimics the behaviour of a drop of water on a surface, requiring less steps of global processing, and thus requiring less communication and synchronisation. Next, we discuss the algorithms that implement this definition on their sequential and parallel versions.

Sequential Watershed Algorithms

The recent faster sequential watershed transform algorithms are the result of the evolution of the arrowing technique for watershed of Bieniek & Moga (2000) and the union-find of Meijster & Roerdink (1998). The arrowing is the algorithmic representation of the drop of water, where for every pixel of an image, an arrow is drawn from the current to the next pixel creating a path that ultimately leads to a regional minimum. The arrow indicates the direction that a drop of water would slide, considering the image as a surface. For the LC-WT definition, every pixel that does not belong to regional minima will have one and only one arrow. The union-find technique for watershed transform is based on the algorithm for disjoint sets of Tarjan (1983). Given that the regions of the output image form a partition and those are disjoint by definition, the union-find algorithm processes paths to identify the roots - or representatives - for every pixel.

Several algorithms based on this preliminary works have been proposed, using variations of the previous procedures, achieving considerable speedups without loss of precision (Sun et al., 2005; Lin et al., 2006; Osma-Ruiz et al., 2007; Cousty et al., 2009). These algorithms are all based on evaluating the neighbourhood to identify the direction of sliding of a drop of water on a surface until it reaches a minimum, and label the regions where the drops falls into the same minimum. However, these procedures are performed using different types of data structures and approaches to the intrinsic model, such as processing the watershed on a graph built upon the image but where pixel values are addressed to edges and cuts are made on these in order to separate regions (Cousty et al., 2009).

The work by Cousty et al. (2009) introduced one of the fastest sequential watershed algorithms due to its linear complexity. However, its constraints and style of switching between breadth-first and depth-first propagation compose a hard problem for a parallel version. Osma-Ruiz et al. (2007) proposed an improved algorithm on the sense of pixel visitation by optimising on previous works. Nevertheless, these improvements required a massive use of queues for synchronisation of pixel visitation. This feature, interesting for a sequential algorithm, in order to minimise memory access, demands several strategies of flow control on a parallel implementation, translated into synchronisation steps. The algorithms of Sun et al. (2005), Lin et al. (2006), Bieniek & Moga (2000) and Meijster & Roerdink (1998) are very similar in the sense of problem approach. Their difference is that Lin's and Meijster and Roerdink's procedure solution is unique, where Sun, Yang and Ren's and Bieniek and Moga's solution depend on scan order. These algorithms are divided in clear steps, where an analysis to identify independence of data to process simultaneously works better. Also, as these steps might be seen as a pipeline, different algorithms might be applied when the used approach does not fit on a parallel architecture.

Even though different, these implementations fall on common problems, such as regional minima detection, plateau resolution, paths construction and pixel labelling. These problems are the same for both sequential and parallel watershed algorithms, once they are parts of the main problem to be solved, i.e. the application of the definition. There are several ways to address these problems, as noted before on (Körbes & Lotufo, 2009), with inner consequences to the other steps of the algorithm. The plateau problem in particular is discussed in depth by Roerdink & Meijster (2000), with a proposal of pre-processing the image. However, the algorithm applied for that matter works with the same principles as the others referenced on this section, thus creating an unnecessary overhead on the whole process.

Parallel Watershed Algorithms

The watershed transform is a useful tool for segmenting images on computer vision processes, and since its introduction studies on parallelism are performed. Initially, the focus was on typical image processing systems, where the segmentation represented a time consuming operation (Meijster & Roerdink, 1995; Bieniek et al., 1997). Roerdink & Meijster (2000) extensively surveyed the literature on this subject. More recently, the evolution of sequential algorithms along with hardware minimised the performance issue. However, for fast applications (e.g. surveillance and navigation) the sequential algorithms are not fast enough, and the focus of works on parallel watershed algorithms have changed to this area using specialised architecture such as FPGAs and clusters (Trieu & Maruyama, 2007; Galilée et al., 2007).

The initial work of Meijster & Roerdink (1995) is based on the work of Vincent & Soille (1991), in the sense of definition and problem approach. The algorithm proposed relies on a graph transformation, performing the watershed on three steps: convert the image into a graph, where each vertex is a connected component of same grey level - a plateau; perform the watershed on the graph; convert the graph into the output image. The most important task, the watershed itself, is done by performing a breadth-first search from the minima, on an iterative flooding. The speedup is a consequence of the reduction of nodes on the graph and of the independence of those.

Bieniek et al. (1997) proposed another parallel algorithm with a modified definition, which is the LC-WT. In fact, this proposal settles a framework for parallelisation, as it approaches the problem on how it should divide the image, generate unique labels for each regional minimum and merge regions, depending on a sequential algorithm executed on each region. The steps of the algorithm are: split the image on regions; for each region find the regional minima and generate labels; set temporary labels to pixels of the borders of regions; run a sequential watershed algorithm on each region; merge the regions processing the pixels with temporary labels.

Galilée et al. (2007) introduces a new algorithm for parallel watershed transform, stating to be the first that does not require minima detection as a first step prior to definition of catchment basins. However, a first sequential step for attributing a distinct label for each pixel is necessary, which is in fact its address in raster scan. The algorithm itself is defined as a single procedure with state management and message passing, for status updating of the pixels. This way, pixels that cannot be processed only with local information wait for neighbourhood data messages to arrive in order to decide which label is taken.

Following the line of work on fast applications, Trieu & Maruyama (2007) developed an algorithm for FPGA architectures based on the sequential algorithm of Sun et al. (2005). The use of queues and stacks for synchronisation on this algorithm is substituted by a process of stabilisation of geodesic distance values and labelling; reading the memory always sequentially, on raster or anti-raster scan order. The plateau resolution is done by explicitly calculating the geodesic distances. Minima are labelled independently on a pixel basis, with merging of these sub-regions deciding for the minor label. Previous works on parallel watershed algorithms for GPU architectures exists (Kauffmann & Piche, 2008; Körbes et al., 2009; Vitor et al., 2009) and emphasise the problem that stabilisation steps introduce when not appropriately designed. A hybrid approach also is considered, using the fastest processing unit for each of the steps of the algorithm. However, entirely parallel proposals are possible to improve the slower steps using adequate algorithms, especially when considering the labelling of minima and regions.

Another approach to the problem is to take it as a shortest path graph problem. For that matter, there are several sequential algorithms that are suited for parallelisation. Kauffmann & Piche (2008) developed a modification of the Ford-Bellman algorithm for calculation of such paths and propagation of labels. This algorithm works as a single kernel, executed on each pixel of the image, thus massively parallel, until stabilisation, with synchronisation after every iteration. On the next section the GPU processing is discussed on the context of its implications on parallel programming and on the algorithm proposed on this paper.

3. GPU Processing

The performance gain in the capacity of GPU devices in the last years is no longer restricted to graphics processing, as it becomes an excellent alternative for general purposes, known as GPGPU, where the CPU's known speed limits are broken with the insertion of this many-core programming paradigm. Also, the range of applications of parallel nature, the gain in speed compared to conventional computers, the simplicity and practicality in use and acquisition of this technology has drawn great attention from the scientific community.

As an example, the GeForce 8800, a card of the NVIDIA's G8 series of graphics cards has 128 units of calculation (called multiprocessors), distributed on 8 vector processors. It is an architecture similar to that found in clusters of CPUs, but confined to a single hardware device, with a style of programming called SIMD (Single Instruction Multiple Data), or, data level parallelism. For this card, execution times of up to 100 times faster than the CPU time in classical programs such as a multiplication of matrices have been obtained (Cooperman & Kaeli, 2009). A study by NVIDIA presented a chart comparing the computational power of an Intel CPU, measured in peak GFlops, to the NVIDIA graphics cards, where the most modern GPU architecture delivers performance up to 6 times higher than the CPUs (NVIDIA, 2009). However, this chart must be evaluated carefully, as it is problem dependent, once the parallelism level is dependent on how data are managed. Also, for some problems there might not be any speedup, if its nature is synchronous or the algorithm is not suited for the architecture. That is, before implementation on GPU, the algorithm must be analysed and rearranged to suit on a data parallel hardware (Kirk & Hwu, 2010).

To explore the potential of the GPU, a different paradigm of programming must be used, called programming flow. Data are packaged in streams and the arithmetic calculations are kernels operating on them. Excerpts of programming that have enormous arithmetic rates can be shared in order to use the most of the GPU. Baggio (2007) characterises the structure of algorithms as follows: (1) the parallel sections of the program are identified and implemented with a kernel, which is a share of the GPU to process arrays of data in parallel on different processors; (2) the organisation of these data should follow a hierarchy for the best arrangement of processing cores.

A note to consider is that the data within the block must perform the same process, as SIMD architecture. This simple and necessary routine for GPU programming does not always fit into some problems because they cannot be arranged in parallel, thus these may not benefit of the acceleration of GPUs. On the other hand when working together with CPU and GPU, the gains are significant. On this line, NVIDIA developed an architecture called CUDA that enables the development involving sequential and parallel programming, where some sections are sequential and others parallel, depending on the problem.

CUDA

The CUDA architecture is a language binding to the C/C++ language for general purpose parallel implementation. CUDA consists of a run-time library extended from C. Its main abstraction is based on the hierarchy of thread groups, memory sharing and synchronisation. Key elements of CUDA are: common C/C++ source code with different compiler forks for CPUs and GPUs; function libraries that simplify programming; and a hardware-abstraction mechanism that hides the details of the GPU architecture from programmers (Halfhill, 2008). As a complex topic out of the scope of this paper, the reader is referred to the work of Nickolls et al. (2008) for a detailed view on CUDA programming and modelling.

4. Algorithm Proposal

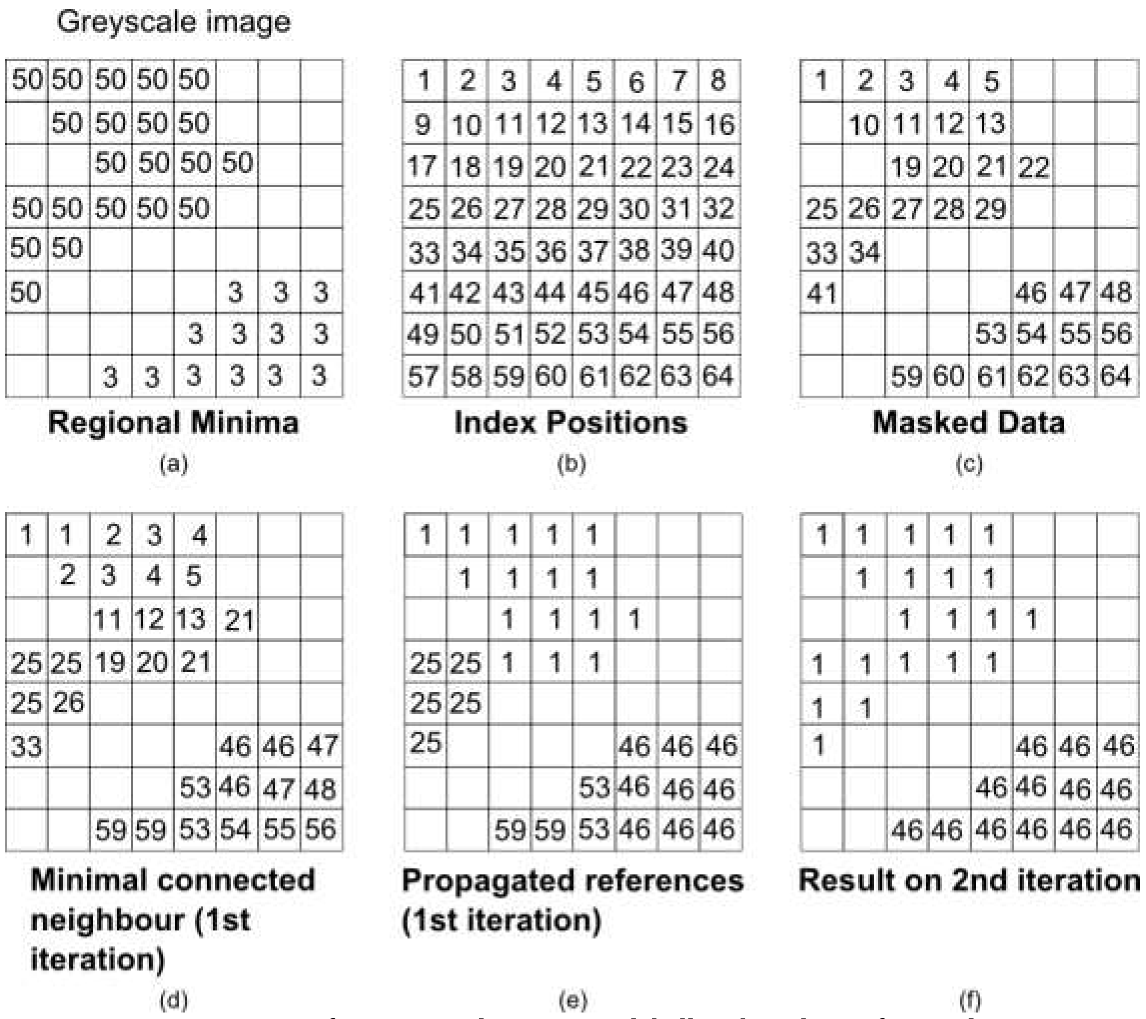
In this section we propose an algorithm for fully parallel SIMD implementation. Firstly, the motivations and inspirations are presented. Next, the notation used is explained, and the algorithm depicted and explained. Lastly, implementation details are pointed out as well as performance considerations.

As presented on Sec. 2, there are several approaches for watershed algorithms. Even though the evolution of techniques achieved a great speedup in comparison with the first fast transforms, those are still not enough for applications such as automatic navigation. Thus, the parallel approach seems obvious to achieve an even greater speedup. Along with the recent development on GPU parallel processing, presented on Sec. 3, with further optimisation for image processing, a new field for the watershed transform is seen, taking advantage of this massive many-core architecture. Also, experiments with hybrid approaches already shown a speedup (Vitor et al., 2009).

This algorithm is inspired in both sequential and parallel previous algorithms presented on Sec. 2. Among the sequential ones, the most influential is by Bieniek & Moga (2000), which inspired on storing the pixel addresses for finding the steepest descent paths on a single step. The strategy of Galilée et al. (2007) of waiting for neighbourhood data to split non-minima plateaus is used. On this case, since there is no message system to exchange data between concurrent threads on the GPU, this communications is handled through synchronisation steps. Iterating until the plateaus are resolved, this algorithm requires memory copies to and from the GPU until it is stable. Other approaches are available for solving the plateau problem, however they all tend to use some kind of ordered propagation, even when transforming the image in the lower completion process. Some algorithms, such as the watershed cut of Cousty et al. (2009) choose to not split plateaus evenly, propagating over these almost randomly.

Another relevant step of any watershed algorithm is the minima and paths labelling. This problem is itself addressed in many ways on both sequential and parallel algorithms, being a usual bottleneck, where common strategies fail on the GPU as noted before (Vitor et al., 2009). For that matter, these steps on the current approach are performed using the labelling algorithm of K.A.Hawick et al. (2009), based on a reference list for path compressing and representative propagation. This algorithm itself is inspired on the union-find approach for path compressing (Tarjan, 1983), though applied in parallel.

Fig. 1 shows an example of the steps applied iteratively for connected component labelling. On (a) it is shown the greyscale image with only its regional minima values. Fig. 1 (b) shows the index positions, used for calculation of the reference list, and shown on (c) as the masked data to be processed for the regional minima. Fig. 1 (d) shows the first step of the first iteration, where every pixel looks on its neighbourhood (inside the mask) for the smallest reference, and store as its own reference. Next, those indexes are propagated, compressing the path from every pixel to the minor reference on the same connected component. As seen on (e), a single iteration does not merge the whole component, thus, a second one is necessary, processing the same steps to set the same reference for every pixel, as shown on (f).



Greyscale image

Regional Minima (a)

50	50	50	50	50			
	50	50	50	50			
		50	50	50	50		
50	50	50	50	50			
50	50						
50					3	3	3
				3	3	3	3
		3	3	3	3	3	3

Index Positions (b)

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Masked Data (c)

1	2	3	4	5			
	10	11	12	13			
		19	20	21	22		
25	26	27	28	29			
33	34						
41					46	47	48
				53	54	55	56
		59	60	61	62	63	64

Minimal connected neighbour (1st iteration) (d)

1	1	2	3	4			
	2	3	4	5			
		11	12	13	21		
25	25	19	20	21			
25	26						
33					46	46	47
				53	46	47	48
		59	59	53	54	55	56

Propagated references (1st iteration) (e)

1	1	1	1	1			
	1	1	1	1			
		1	1	1	1		
25	25	1	1	1			
25	25						
25					46	46	46
				53	46	46	46
		59	59	53	46	46	46

Result on 2nd iteration (f)

1	1	1	1	1			
	1	1	1	1			
		1	1	1	1		
1	1	1	1	1			
1	1						
1					46	46	46
				46	46	46	46
		46	46	46	46	46	46

Figure 1: Steps for connected component labelling based on reference list

Algorithm 1 presents our proposal for a parallel watershed transform. On the course of it, the statement **for <condition> in parallel do** denotes a kernel execution on GPU architecture, indicating that each iteration can be processed independently, in a SIMD way. The algorithm is divided in four major steps: (1) find the lowest neighbour of each pixel (direct path of steepest descent); (2) find the nearest border of internal pixels of plateaus, propagating uniformly from the borders; (3) minima labelling by minor neighbour address and (4) pixel labelling by path compression. The input image is called I, with a domain D and the output labelled image is called **lab**, which is also used for working. As noted before, a similar algorithm has been proposed recently, though with modified strategies for labelling (Körbes et al., 2009), that had shown worst results than a sequential approach (Vitor et al., 2009).

Algorithm 1: Parallel watershed transform (GPU-WS)

```
// First Step
PLATEAU = 0
for p in D in parallel do
    if  exists q in N(p):  I(q) < I(p) and I(q) = min{I(u), u in N(p)} then
        lab(p) = -q
    else
        lab(p) = PLATEAU
    end if
end for

// Second step
while lab is not stable do
    Tlab = lab
    for p in D: lab(p) = PLATEAU in parallel do
        if  exists q in N(p):  lab(q) < 0 and I(q) = I(p) then
            Tlab(p) = - q
        end if
    end for
    lab = Tlab
end while

// Third step
for p in D: lab(p) = PLATEAU in parallel do
    lab(p) = p
end for
while lab is not stable do
// Propagation of minimal index
    for p in D: lab(p) > PLATEAU in parallel do
        q = min{lab(u):  u in N(p) and I(u) = I(p)}
        if  q < lab(p) then
            lab(lab(p)) = q
        end if
    end for
// Representative propagation/Path Compressing
    for p in D: lab(p) > PLATEAU in parallel do
        label = lab(p)
        if  label != p then
            do
                ref = label
                label = lab(ref)
            while label != ref
            lab(p) = label
        end if
    end for
end while

// Fourth step
for p in D: lab(p) = PLATEAU in parallel do
    lab(p) = |lab(p)|
end for
for p in D in parallel do
    label = lab(p)
    if  label != p then
        do
            ref = label
            label = lab(ref)
        while label != ref
        Wait()
        lab(p) = label
    end if
end for
```

Using this algorithm, an image is taken as a grid divided in 16x16 pixels blocks, resulting in up to 256 threads. The image loaded on the CPU is copied to the texture memory of the GPU, where it is then processed via kernel access. The kernels were developed in three ways. The first is for step 1, which is based on texture memory access with block division as mentioned above, outputting its results to global memory. For the further steps, the second kernel demands a special approach, once pixel evaluation is highly guided by its neighbours. In effect, each block must include a border, containing neighbour pixels of the block, for correct processing. Fig. 2 shows the modelling used for each image block.

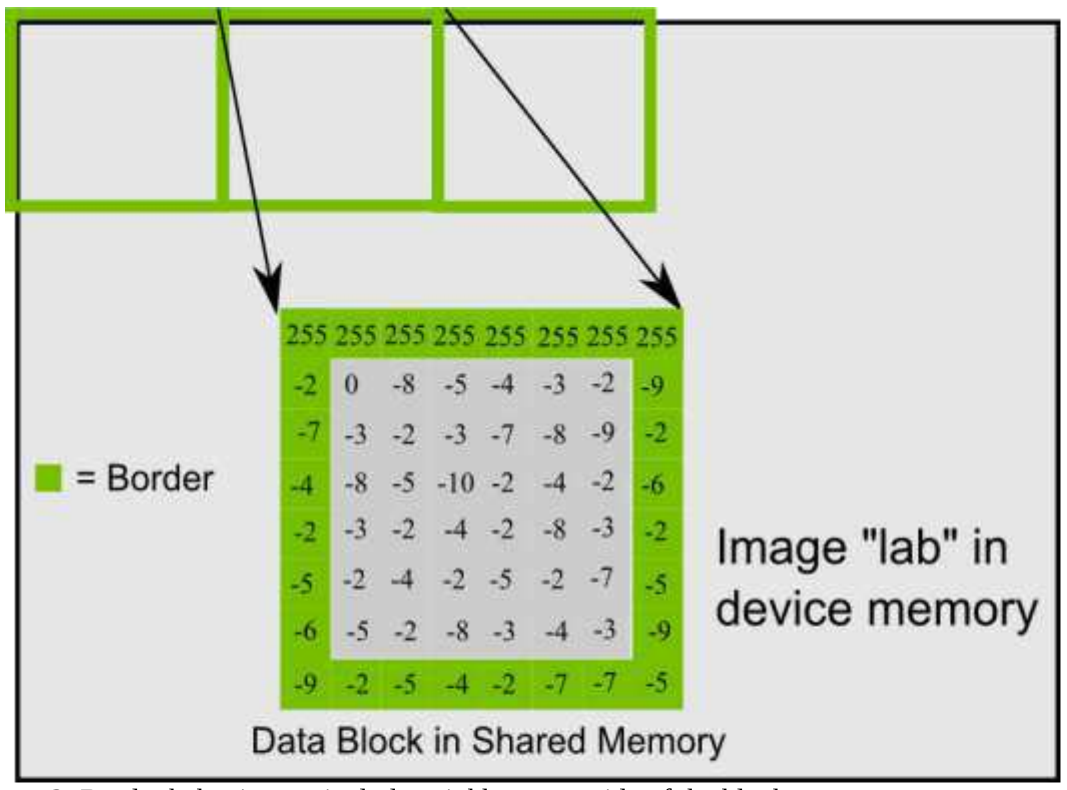


Figure 2: Border behaviour to include neighbours outside of the block

Considering coalescent data access, the block dimension is kept on 16x16, though with a processing range of 14x14 centred on it. At that point, each block processes 196 threads, with other 60 inactive threads, where these data are processed by adjacent blocks. Once the data is loaded from the global memory to shared memory, the results are stored on output global memory, ensuring that border results do not overlap (Podlozhnyuk, 2007).

The third step was developed using two kernels, where the first propagates the minimal index for the neighbours on the connected component, and the second transforms the image into the inner reference list. This list is processed for equivalence of indexes until stabilisation, that is, the minor label is propagated through the connected path obtained on the first kernel. These two kernels iterate until every minima is merged with its pixels, with these iterations controlled on the CPU. Fig. 1 (d)-(f) shows these iterations of each kernel. Along with these specifications, the algorithm is dependent on stabilisation steps, which are verified on the CPU. As the experimental results presented on Sec. 5 show, these steps contribute heavily for increasing the performance of Algorithm 1, compared to a previous work (Vitor et al., 2009). The last step use one kernel to solve the equivalence list, propagating the label of the regional minima through the connected path between the pixels obtained on the first step. Fig. 3 shows each of the steps applied on a sample image. On this image, there are two regional minima, with levels 0 and 10. On the first step, every pixel locates its lower neighbour, as is indicated by the arrows. The plateau's inner pixels are then resolved on step 2, with the created paths indicated by four new arrows. On step 3, the regional minima are labelled, and finally on step 4 the labels of the minima are propagated to every other pixel.

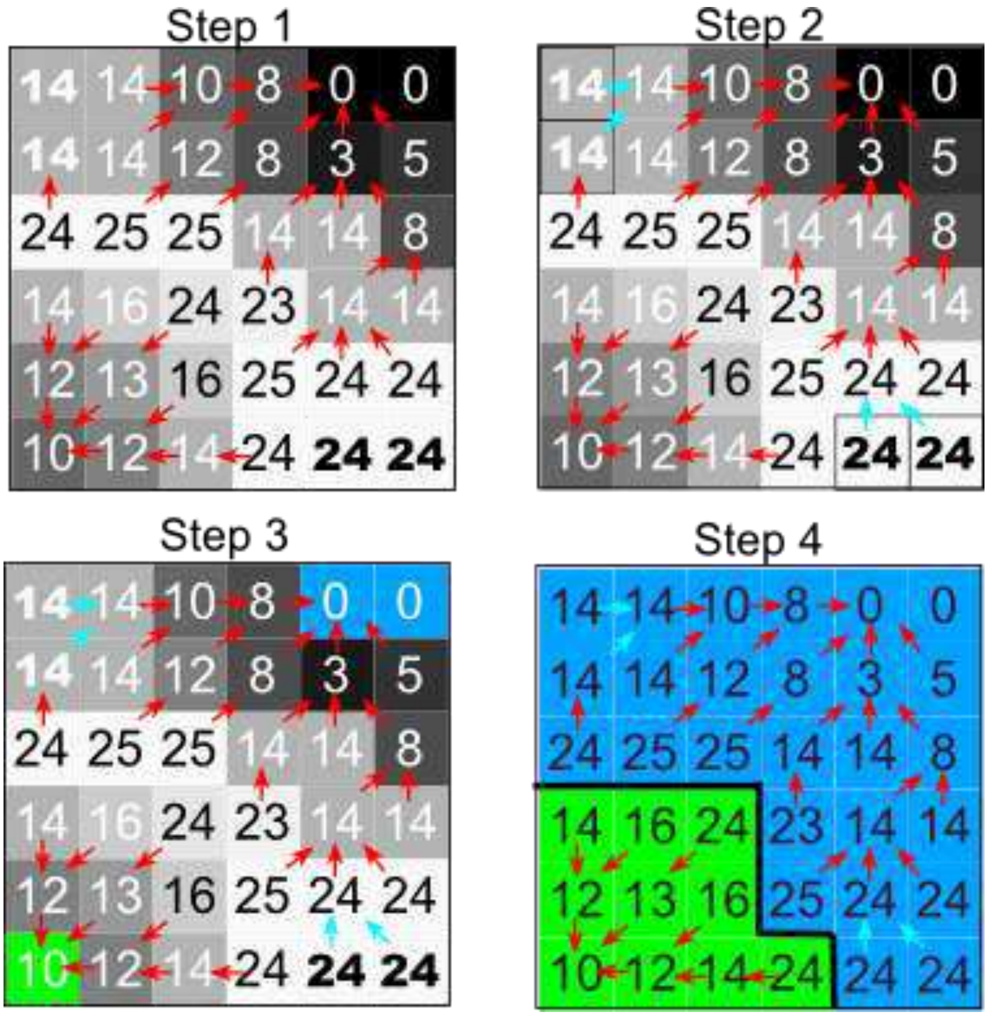


Figure 3: Representation of the processing performed by each of the steps of the proposed algorithm

5. Experimental Results

The algorithm was tested on several images on sizes ranging from 64x64 to 2048x2048, and had its performance measured on each step. The data presented following is the average of the experiments, scaled to milliseconds (ms). The sequential algorithm implemented is a modified version of Lin et al. (2006), which operates also on four steps very similar in purpose to those of Algorithm 1, but without the watershed label, as the authors originally proposed, for matters of equivalence of definitions and better performance. Table 1 shows the performance for the sequential algorithm, processed on the CPU. These results were obtained on a computer with an AMD Phenom II X3 (2.6Ghz 7.5Mb Cache) with 4 GB RAM and a GeForce GTX295 with 1792Mb. Only one GPU core was used for the GPU processing.

	64x64	128x128	256x256	512x512	1024x1024	2048x2048
Step 1	0.18	0.72	3.16	11.94	45.80	171.07
Step 2	0.27	1.10	4.36	17.84	72.69	306.14
Step 3	0.03	0.18	0.46	1.68	4.46	17.07
Step 4	0.05	0.15	0.93	3.56	15.58	70.06
Total	0.55	2.17	8.92	35.04	138.54	564.36

Table 1: Sequential algorithm processed on the CPU

Table 1 shows that for the sequential algorithm, the first two steps are severe bottlenecks, greatly degrading the performance, especially on larger images. Moving to the parallel implementation on the GPU, developed according to Algorithm 1, Table 2 shows that the bottleneck on step 1 is completely mitigated, once it is processed on single kernel execution, whereas for step 2 the performance depends on image size, though showing small improvements. Also the memory copies to and from the GPU must be considered, as it is a time consuming task that does not exist for the sequential algorithm. This is easily seen as the sum of each step of the algorithm on Table 2 does not match the total, where this difference corresponds to memory allocation and copying on the GPU between global and texture memories and also between CPU and GPU memories.

	64x64	128x128	256x256	512x512	1024x1024	2048x2048
Step 1	0.05	0.08	0.19	0.60	2.31	9.18
Step 2	0.22	0.52	1.53	6.00	26.61	186.80
Step 3	0.24	0.67	1.52	4.02	18.95	79.81
Step 4	0.02	0.04	0.06	0.16	0.69	3.75
Total	1.52	1.94	3.54	12.62	50.85	284.96

Table 2: Parallel algorithm processed on the GPU with time measurements in ms

With the data collected on these tables, a comparison may be done, checking the effective speedup in time performance. This evaluation is seen on Table 3, where the algorithms are put aside and the speedup ratio between GPU and CPU is calculated for the average total time for each image size tested.

	CPU (ms)	GPU (ms)	Speedup
64x64	0.55	1.52	0.36
128x128	2.17	1.94	1.11
256x256	8.92	3.54	2.52
512x512	35.04	12.62	2.78
1024x1024	138.54	50.85	2.72
2048x2048	564.36	284.96	1.98

Table 3: Comparison between CPU and GPU algorithms using the speedup as the ratio of CPU by GPU measured time.

The analysis of Table 3 must highlight some points of GPU parallel processing. One is the fact that for small images, such as 64x64 and even 128x128 pixels, the memory operations produce an overhead that eliminates or greatly minimises advantages of a parallel algorithm. Once image sizes increase, this effect is reduced and the speedup is more sensible. For average sizes of images - typically those produced by video cameras - the parallel approach shows an average speedup of 2.7, meaning a reduction of time of more than half of the sequential algorithm. These results are better than the previous obtained with a hybrid algorithm (Vitor et al., 2009). Nevertheless, the algorithm might be further improved, especially on step 2, where a different approach for the plateau resolution could increase the speedup.

Noting that a considerable speedup has been achieved, it is also necessary to compare this work with the one by Kauffmann & Piche (2008). For that matter, the total number of pixels is used as the base for comparison. However, as the authors were interested in 3D images, with orders of tens of millions of pixels, there is data available only for the size of 512x512 and 1024x1024 pixels. There must be used some caution when comparing these data. Given that the GPUs are in constant evolution, with increasing performance, this must be taken into consideration. Kauffmann & Piche (2008) used a Radeon X1950 Pro with 512MB of memory whilst in this paper was used a GTX295 with 1792MB of memory. In order to make a reasonable comparison, and since both GPUs have the same internal clock of 595 MHz, the number of cores is used as a basis. The Radeon X1950 Pro have 48 pixel shaders (AMD, 2010), and the GTX295 have 240 CUDA cores (NVIDIA, 2010). Thus, the measure obtained on the proposed algorithm is multiplied by this ratio, simulating its operation on 48 cores. The measures of both algorithms are presented on Table 4, where GPU-WS is the current proposal, and CA-WS is the data of Kauffmann & Piche (2008).

	GPU-WS (ms)	CA-WS (ms)
512x512	12.62 x 5 = 63.1	600
1024x1024	50.85 x 5 = 254.25	1177

Table 4: Performance data of algorithm 1 and the algorithm of Kauffmann & Piche (2008)

Once all the algorithms compared here yield very similar results, due to it’s conceptually equivalence regarding watershed transform definition, there is no need to show the produced images.

6. Conclusion

This paper presents a watershed transform algorithm with implementation on the GPU using the CUDA run-time libraries. With this implementation, a speedup of more than two times was obtained in comparison with a CPU algorithm. A significant part of this improvement on previous works is due to the labelling algorithm utilised, reducing greatly the iterations necessary for the task completion. However, more effort is necessary as there is room to improve on the step that splits plateaus, which still demands a stabilisation that degrades performance when the image has large areas of such type. A comparison with another watershed algorithm based on the GPU was made, and even though based on different platforms, shows a reasonable improvement. We believe that the spreading of such many-core architectures provides the environment for great improvements on the image segmentation fields, using complex and global algorithms, such as the watershed one.

As future works, we intend to continue on the work on parallel algorithms specialised to many-core architectures, such as morphological reconstruction and watershed from markers.

References

AMD (2010). Radeon X1950 Specifications. http://www.amd.com/us/products/desktop/graphics/other/Pages/x1950-specifications.aspx

Audigier, R. & Lotufo, R. A. (2007). Watershed by image foresting transform, tie-zone, and theoretical relationships with other watershed definitions. In ISMM’2007 Proceedings, volume 1 Sao José dos Campos: Universidade de São Paulo (USP) Instituto Nacional de Pesquisas Espaciais (INPE).

Baggio, D. L. (2007). Gpu based image segmentation livewire algorithm implementation. Master’s thesis, Technological Institute of Aeronautics, Sao José dos Campos.

Bieniek, A., Burkhardt, H., Marschner, H., Nölle, M., & Schreiber, G. (1997). A parallel watershed algorithm. In Proceedings of 10th Scandinavian Conference on Image Analysis (SCIA97) (pp. 237–244).

Bieniek, A. & Moga, A. (1998). A connected component approach to the watershed segmentation. In ISMM ’98: Proceedings of the fourth international symposium on

Mathematical morphology and its applications to image and signal processing (pp. 215–222). Norwell, MA, USA: Kluwer Academic Publishers.

Bieniek, A. & Moga, A. (2000). An efficient watershed algorithm based on connected components. Pattern Recognition, 33(6), 907–916. Cooperman, G. & Kaeli, D. (2009). Gpu programming – syllabus. http://www.ccs.neu.edu/course/csu610/#syllabus.

Cousty, J., Bertrand, G., Najman, L., & Couprie, M. (2009). Watershed cuts: Minimum spanning forests and the drop of water principle. IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(8), 1362–1374.

Cousty, J., Bertrand, G., Najman, L., & Couprie, M. (2009, to appear). Watershed cuts: Thinnings, shortest-path forests and topological watersheds. IEEE Transactions on Pattern Analysis and Machine Intelligence.

Digabel, H. & Lantuéjoul, C. (1978). Iterative algorithms. In J.-L. Chermant (Ed.), Proc. Second European Symp. Quantitative Analysis of Microstructures in Material Science, Biology and Medicine (pp. 85–99). Stuttgart, Germany: Riederer Verlag.

Falcão, A. X., Stolfi, J., & Lotufo, R. A. (2004). The image foresting transform: theory, algorithms, and applications. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 26(1), 19–29.

Galilée, B., Mamalet, F., Renaudin, M., & Coulon, P.-Y. (2007). Parallel asynchronous watershed algorithm-architecture. IEEE Transactions on Parallel and Distributed Systems, 18(1), 44–56.

Gonzalez, R. C. & Woods, R. E. (2002). Digital Image Processing. Prentice Hall, 2 edition.

Halfhill, T. R. (2008). Parallel Processing with CUDA: Nvidia's high-performace computing platform uses massive multithreading. NVIDIA.
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf.

K.A.Hawick, A.Leist, & D.P.Playne (2009). Parallel Graph Component Labelling with GPUs and CUDA. Technical report, Institute of Information and Mathematical Sciences, Massey University, Auckland, New Zealand. http://www.massey.ac.nz/~kahawick/cstn/089/cstn-089.html.

Kauffmann, C. & Piche, N. (2008). A cellular automaton for ultra-fast watershed transform on gpu. In Pattern Recognition, 2008. ICPR 2008. 19th International Conference on (pp. 1–4). Tampa, Florida, USA: IEEE Computer Society.

Kirk, D. B. & Hwu, W. (2010). Programming Massively Parallel Processors: A Hands-on Approach, chapter Parallel Programming and Computational Thinking. Morgan Kaufmann: Burlington, MA, USA.

Körbes, A. & Lotufo, R. (2009). Analysis of the watershed algorithms based on the breadth-first and depth-first exploring methods. In SIBGRAPI'09 (pp. 133–140). Rio de Janeiro, Brazil: IEEE Computer Society.

Körbes, A., Vitor, G., Ferreira, J., & Lotufo, R. (2009). A proposal for a parallel watershed transform algorithm for real-time segmentation. In Proceedings of Workshop de Visao Computacional WVC'2009 Sao Paulo, Brazil.
Available on http://iris.sel.eesc.usp.br/wvc2009/WVC2009 CD. rar.

Lin, Y., Tsai, Y., Hung, Y., & Shih, Z. (2006). Comparison between immersion-based and toboggan-based watershed image segmentation. IEEE Transactions on Image Processing, 15(3),632–640.

Lotufo, R. & Falcão, A. (2000). The ordered queue and the optimality of the watershed approaches. In Proceedings of the 5th International Symposium on Mathematical Morphology and its Applications to Image and Signal Processing, volume 18 (pp. 341–350).: Kluwer Academic Publishers.

Meijster, A. & Roerdink, J. B. T. M. (1995). A Proposal for the Implementation of a Parallel Watershed Algorithm - CAIP'95, volume 970 of Lecture Notes in Computer Science, (pp. 790–795). Springer Berlin / Heidelberg.

Meijster, A. & Roerdink, J. B. T. M. (1998). A disjoint set algorithm for the watershed transform. In Proc. IX European Signal Processing Conf EUSIPCO '98 (pp. 1665–1668).

Meyer, F. (1994). Topographic distance and watershed lines. Signal Processing, 38(1), 113–125.

Moga, A., Viero, T., Dobrin, B., & Gabbouj, M. (1994). Implementation of a distributed watershed algorithm. In J. Serra & P. Soille (Eds.), Mathematical morphology and its applications to image processing, volume 2 (pp. 281–288).: Kluwer Academic Publishers.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with cuda. ACM Queue, 6(2), 40–53.

NVIDIA (2009). CUDA Programming Guide, 2.1.
http://developer.download.nvidia.com/compute/cuda/2 1/toolkit/docs/NVIDIA CUDA Programming Guide 2.1.pdf.

NVIDIA (2010). GeForce GTX 295 Specifications.
http://www.nvidia.com/object/product_geforce_gtx_295_us.html

Osma-Ruiz, V., Godino-Llorente, J. I., Sáenz-Lechón, N., & Gómez-Vilda, P. (2007). An improved watershed algorithm based on efficient computation of shortest paths. Pattern Recognition, 40(3), 1078–1090.

Podlozhnyuk, V. (2007). Image Convolution with CUDA. NVIDIA. http://developer. download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/ convolutionSeparable.pdf.

Roerdink, J. B. T. M. & Meijster, A. (2000). The watershed transform: definitions, algorithms and parallelization strategies. Fundam. Inf., 41(1-2), 187–228.

Sun, H., Yang, J., & Ren, M. (2005). A fast watershed algorithm based on chain code and its application in image segmentation. Pattern Recognition Letters, 26(9), 1266–1274.

Tarjan, R. E. (1983). Data structures and network algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Trieu, D. B. K. & Maruyama, T. (2007). Real-time image segmentation based on a parallel and pipelined watershed algorithm. Journal of Real-Time Image Processing, 2(4), 319–329.

Vincent, L. & Soille, P. (1991). Watersheds in digital spaces: An efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence, 13(6), 583–598.

Vitor, G., Ferreira, J., & Körbes, A. (2009). Fast image segmentation by watershed transform on graphical hardware. In Proceedings of the 30oCILAMCE Armaçao dos Buzios, Brazil.