A Proposal for a Parallel Watershed Transform Algorithm for Real-Time Segmentation

André Körbes[1], Giovani B. Vitor[2], Janito V. Ferreira[2], Roberto de Alencar Lotufo[1]

[1]Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Av. Albert Einstein - 400
CP 6101 - Campinas, SP, Brasil
korbes@dca.fee.unicamp.br, lotufo@unicamp.br

[2]Universidade Estadual de Campinas
Faculdade de Engenharia Mecânica
Rua Mendeleyev, 200
CP 6122 - Campinas, SP, Brasil
{giovani,janito}@fem.unicamp.br

Abstract

The watershed transform is widely used for image segmentation on computer vision applications. However, sequential watershed algorithms are not suitable for real-time applications, once they are one demanding part of several tasks. This paper proposes a new parallel algorithm for the watershed transform focused on real-time image segmentation using off-the-shelf GPUs. In this sense, this algorithm aims for a speedup by mixing several techniques of the fastest procedures on both sequential and parallel fields. The algorithm has four major steps, processed on SIMD, with implementation details tuned for many-core architectures, without the use of explicit message passing schemes.

1. Introduction

The task of object identification on images demands a previous processing, which are in general based on segmentation by discontinuity or similarity. The segmentation itself is not trivial, once it is one of the hardest tasks on image processing and its precision determines the success or failure of a computer vision application [9].

Some intrinsic problems are the illumination variation over a sequence of images, dynamic change of background with moving camera and/or target, deformation or overlapping of targets, among others, which result in a complex problem. On literature, there are several approaches for real-time segmentation. One of the most used is the frame subtraction, which is very robust, but with a restriction of a static camera, not applicable when both the scene and camera changes through time. Other techniques, such as shape and colour extraction exist, but are also subject to the previous presented problems. An option to manage these problems is to unite more than one technique, aiming for gain in robustness. However, the disadvantage of this process is on the performance, that degrades with the use of more processing, restricting it from running on real-time. With this focus, this paper contributes with a new parallel watershed transform algorithm for the real-time image segmentation task.

The watershed transform is a very data intensive process in the sense of both memory and processor use, demanding sometimes several scans of the image, graph building, complex data structures for maintaining data, etc. These needs affect the performance of sequential algorithms so that real-time processing is not possible. In order to handle these problems, parallelization is an obvious choice. Along with these needs, the spreading of the parallel GPUs (Graphical Processing Unit) with the CUDA (Compute Unified Device Architecture) architecture present a new field for the development of watershed algorithms.

The work on parallel watershed algorithms was introduced by Moga *et al.* for reduction of time of processing for large images [16]. With advances on CPU speed, the focus changed more recently to real-time image segmentation, either on clusters or FPGAs [8, 23]. In this sense, our work, to the best of our knowledge, is innovative as we propose an algorithm for implementation on off-the-shelf video cards.

This work has been inspired by several other algorithms. Among the sequential ones, the most influential is Sun, Yang and Ren's [22], with the point-out and point-in strategy for storing the implicit lower-complete graph of the image. Also, the Union-Find strategy used by Meijster and Roerdink is used for minima labelling [14]. On the parallel field, the strategy of Galilée *et al.* [8] for local decisions through message passing is used, although in a different way, without explicit messages.

This paper is organised as follows: Sec. 2 presents a review on the state of the art of watershed algorithms, both on sequential and parallel fields. Sec. 3 presents a review

on the GPU architecture, with its benefits of processor organisation and memory access. In Sec. 4 our algorithm is proposed and discussed. Sec. 5 presents the results that are expected with the use of this algorithm. Lastly, Sec. 6 discuss the conclusions and perspectives of use along with future works.

2. Watershed Transform

Many definitions for the watershed transform exist in literature [3, 6, 7, 12, 15, 24] that take different approaches on the problem, such as defining connected components via influence zones, shortest-path forests with a custom distance function and locally, by making paths of steepest descent.

Over the years, several algorithms of watershed have been proposed, according to different formal definitions and applying different strategies. On this paper, we only focus on those based on the local condition definition, which requires the less global operations, once it mimics the behaviour of a drop of water on a surface, easing a parallel implementation. Next, we discuss those algorithms on their sequential versions and the work on parallel watershed on the literature.

2.1. Sequential Watershed Algorithms

The recent fastest sequential watershed transform algorithms are the result of the evolution of the *arrowing* technique for watershed of Bieniek and Moga [3, 4] and the union-find one of Meijster and Roerdink [14]. Several algorithms based on this preliminary works have been proposed, using variations of the previous procedures, achieving considerable speedups without loss of precision [6, 11, 20, 22]. These algorithms are all based on evaluating the neighbourhood, intuitively, 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.

The work by Cousty *et al.* introduced one of the fastest sequential watershed algorithms due to its linear complexity [6]. However, its constraints and style of switching between breadth-first and depth-first propagation compose a hard problem for a parallel version. On the other side, its elegant design with the use of sets instead of queues for pixel storage and labelling suggests a possible path for parallelization.

Osma-Ruiz *et al.* proposed an improved algorithm on the sense of pixel visitation by optimising the preliminary works previously mentioned [20]. 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, such as locks.

The algorithms of Sun, Yang and Ren [22], Lin *et al.* [11], Bieniek and Moga [4] and Meijster and Roerdink [14] are very similar in the sense of problem approach. Their difference is on which definition is applied, considering that Lin's and Meijster and Roerdink's procedure applies a label to distinguish pixels where the path of steepest descent is ambiguous between two or more minima. With the use of explicit loops over every pixel depending only on local information, these algorithms are good candidates for parallelization, even though not being the fastest sequential ones.

2.2. Parallel Watershed Algorithms

Given that the watershed transform is a very demanding task, early studies have been developed on parallel algorithms. Initially, the focus was on typical image processing systems, where the segmentation represented a time consuming operation [2, 13]. Roerdink and Meijster extensively surveyed the literature on this problem [21]. More recently, the evolution of sequential algorithms along with hardware minimised this problem. However, for real-time 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 [8, 23].

The initial work of Meijster and Roerdink [13] is based on the work of Vincent and Soille [24], 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 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.

Bieniek *et al.* proposed another parallel algorithm with a modified definition, that is the local condition watershed transform [2]. In fact, this proposal settles an architecture for a parallelization, as it approaches the problem on how it should divide the image, generate unique labels 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.* 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 [8]. 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 de-

fined as a single procedure with state management and message passing, for status updating of the pixels. This way, pixels that cannot be processed with only local information, the plateaus, wait for neighbourhood data messages to arrive in order to decide which label is taken.

Following the line of work on real-time applications, Trieu and Maruyama [23] developed an algorithm for FPGA architectures based on the sequential algorithm of Sun, Yang and Ren [22]. 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.

On the next section the GPU processing is discussed on its implications on parallel programming and the algorithm proposed on this paper.

3. GPU Processing

The performance gain in the capacity of GPU (Graphics Process Unit) devices in the last years is no longer exclusive to graphics processing, as it becomes an excellent alternative for general purposes, 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. This graphics card 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, and as such requires a style of programming called SIMD (Single Instruction Multiple Data). For this card, execution times of up to 100 times faster than the CPU time in classical programs as a multiplication of matrices have been obtained [5]. 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 [18].

To explore the potential of the GPU, a different paradigm of programming should 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 [1] 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, as shown in Fig. 1.

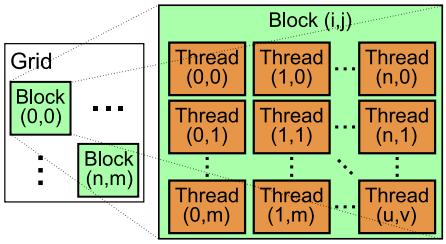

Figure 1. Structuring of data for implementation in parallel [18]

A note to consider is that the data within the block must perform the same process, as a SIMD architecture. This simple and necessary routine for GPU programming does not always fit into some problems because they can not 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 structuring involving sequential and parallel programming, where some sections are sequential and others parallel, depending on the problem.

3.1. CUDA

The CUDA architecture is a language binding to the C/C++ language for general purpose parallel implementation. CUDA consists of a runtime library extended from C. Its main abstraction is based on the hierarchy of thread groups, memory sharing and synchronisation. Fig. 2 presents the programming platform for CUDA and C/C++ proposed by Halfhill [10]. As a complex topic out of the scope of this paper, the reader is referred to the work of Nickolls *et al.* [17] for a detailed view on CUDA programming and modelling.

4. Proposed Algorithm

In this section our proposed algorithm is exposed and explained. Firstly, the motivations and inspirations are presented. Next, the notation used is explained, and the algorithm depicted. Lastly, implementation details foreseen are pointed out.

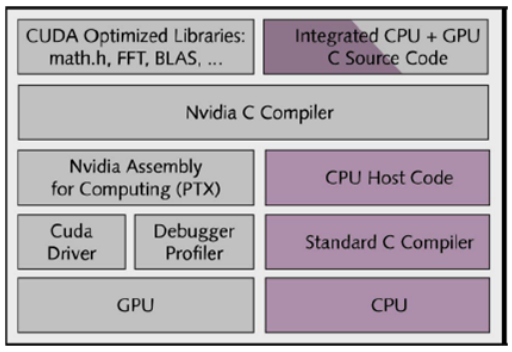

Figure 2. NVIDIA's CUDA platform for parallel processing on NVIDIA GPUs. Key elements are common C/C++ source code with different compiler forks for CPUs and GPUs; function libraries that simplify programming; and a hardware-abstraction mechanism that hides the details of the GPU architecture from programmers [10]

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 real-time applications. Thus, the parallel approach seem 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.

This algorithm is inspired in both sequential and parallel previous algorithms presented on Sec. 2. Among the sequential ones, the most influential is Sun, Yang and Ren's [22], which inspired for the use of the point-out and point-in strategy for finding the steepest descent paths. Also, the Union-Find strategy used by Meijster and Roerdink is used for finding and labelling the minima's connected components [14]. The strategy of Galilée *et al.* [8] of waiting for neighbourhood data to split non-minima plateaus is used, although in a different way, without explicit message passing.

Algorithm 1 presents our proposal for a parallel watershed transform. On the course of it, the statement **for all** denotes that every iteration can be processed in parallel and there is a synchronisation step, where the next statement after it is only processed after every parallel process has terminated, and the usual **for** denotes a step where only a single processor works. The **Arrow** procedure indicate through a number which neighbour is being selected, whereas **Pointed** retrieves the set of neighbours that point to a specific one. Fig. 3 exemplifies this process, showing the numbers and behaviour of both functions.

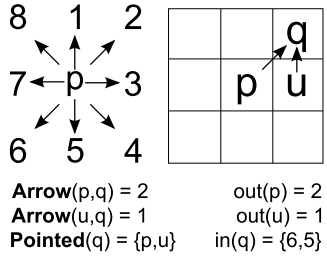

Figure 3. Sample behaviour of procedures Arrow and Pointed with the data stored on `out` and `in`

Also the regular procedures of Union-Find are used, namely **MakeSet**, **Union**, **FindRoots** and **FindRoot**, where they respectively create a set, merge two sets, locate the representative members of all the sets and locate the set representative member of any member of it. These procedures, specially Union, must be modified from its original versions in order to ensure sync between parallel calls.

The algorithm is divided in four major steps: find the lowest neighbour of each pixel (direct path of steepest descent); find the nearest border of internal pixels of plateaus; minima labelling and pixel labelling by flooding from minima. The input image is called I, and the output labelled image is called lab. Two working images are used: out and in, which are of the same dimension of I [1].

Algorithm 1: Parallel watershed transform

1: */* First Step */*
2: **for all** $p \in D$ **do**
3: **if** $\exists q \in N(p) \mid I(q) < I(p)$ **and** $I(q) = min_{\forall q' \in N(p)} I(q')$ **then**
4: $out(p) \leftarrow \mathbf{Arrow}(p, q)$
5: $in(q) \leftarrow in(q) \cup \mathbf{Arrow}(q, p)$
6: **else**
7: $P \leftarrow P \cup p$
8: **end if**
9: **end for**
10: */* Second step */*
11: **while** P is not stable **do**
12: **for all** $p \in P$ **do**
13: **if** $\exists q \in N(p) \mid out(q) > 0$ **and** $I(q) \leq I(p)$ **then**
14: $out'(p) \leftarrow \mathbf{Arrow}(p, q)$
15: $in(q) \leftarrow in(q) \cup \mathbf{Arrow}(q, p)$

1 As an implementation detail, we follow the same strategy of Sun, Yang and Ren for storing the values on in, that is to use binary flags to indicate the direction [22].

```
16:          P ← P \ p
17:          W ← W ∪ p
18:        end if
19:      end for
20:      for all p ∈ W do
21:        out(p) ← out'(p)
22:      end for
23:      W ← ∅
24:    end while
25:    /* Third step */
26:    for all p ∈ P do
27:      MakeSet(p)
28:    end for
29:    for all p ∈ P do
30:      for q ∈ N(p) | I(q) = I(p) do
31:        Union(p, q)
32:      end for
33:    end for
34:    R ← FindRoots()
35:    labels ← 0
36:    for p ∈ R do
37:      labels ← labels + 1
38:      lab(p) ← labels
39:    end for
40:    for all p ∈ P do
41:      r ← FindRoot(p)
42:      lab(p) ← lab(r)
43:    end for
44:    /* Fourth step */
45:    while P ≠ ∅ do
46:      for all p ∈ P | in(p) > 0 do
47:        for q ∈ Pointed(p) do
48:          lab(q) ← lab(p)
49:          W ← W ∪ q
50:        end for
51:      end for
52:      P ← W
53:    end while
```

For the parallel implementation of the algorithm, some parameters must be defined. The first one is the number of threads per block, in order to divide the processing and obtain the best performance. The image block size must be a multiple of the warp size, which is characterised as a minimum group of 64 threads, processed as SIMD by CUDA many-core architecture [19]. Empirically, it is suggested the use of 256 threads as a good choice, balancing between memory latency, registers and threads.

Another parameter is the memory access. For this development, the texture memory is proposed, given its benefits compared to both global and constant memory. Some of these are [18]: hardware-based linear interpolation and border management; potential higher bandwidth, once its cached. An observation is that the global memory is used to store intermediary results, given that the texture memory is read-only.

Based on this approach, 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. As some parts of the algorithm need mutual exclusion, mostly the disjoint sets functions, CUDA also provides atomic functions, ensuring that other threads do not interfere on the memory addresses until completion of the operation [18].

5. Expected Results

The proposed algorithm and implementation are expected to increase the performance of the watershed transform, reducing the time of processing, taking advantage from the many-core architecture of off-the-shelf GPUs. This increase is expected to enable real-time processing that requires image segmentation, such as target tracking on navigation applications, with little specific hardware effort, as the computation is done locally on the GPU. As a proposal, there are no performance results available, such as speedup comparisons nor sample results. Also, with the spreading of many-core GPUs, the proposed algorithm may enable the transference of common segmentation tasks - such as the watershed transform - from the CPU to the GPU.

6. Conclusions and Perspectives

In this paper we proposed a parallel watershed transform algorithm for implementation on many-core architectures, such as GPUs, with details on the CUDA architecture. We reviewed the state of the art on both sequential and parallel algorithms and proposed a new one inspired on some techniques presented. Also the architecture of the GPUs with CUDA were presented. Our algorithm is described along with the discussion of some of the expected implementation issues. With this algorithm, we expect to achieve a significant increase on the performance compared with the fastest sequential algorithms. As future works, we intend to continue on the work on parallel algorithms specialised to many-core architectures, such as morphological reconstruction, watershed from markers, etc.

References

[1] D. L. Baggio. Gpu based image segmentation livewire algorithm implementation. Master's thesis, Technological Institute of Aeronautics, São José dos Campos, 2007.

[2] A. Bieniek, H. Burkhardt, H. Marschner, M. Nölle, and G. Schreiber. A parallel watershed algorithm. In *Proceed-*

ings of 10th Scandinavian Conference on Image Analysis (SCIA97), pages 237–244, 1997.

[3] A. Bieniek and A. Moga. 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*, pages 215–222, Norwell, MA, USA, 1998. Kluwer Academic Publishers.

[4] A. Bieniek and A. Moga. An efficient watershed algorithm based on connected components. *Pattern Recognition*, 33(6):907–916, 2000.

[5] G. Cooperman and D. Kaeli. Gpu programming – syllabus. `http://www.ccs.neu.edu/course/csu610/#syllabus`, June 2009.

[6] J. Cousty, G. Bertrand, L. Najman, and M. Couprie. Watershed cuts: Minimum spanning forests and the drop of water principle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008, to appear.

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

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

[9] R. C. González and R. E. Woods. *Digital Image Processing*. Prentice Hall, 2 edition, 2002.

[10] T. R. Halfhill. Parallel processing with cuda: Nvidia's high-performace computing platform uses massive multithreading. *Microprocessor Report*, 2008. `http://www.nvidia.com/docs/IO/55972/220401\_Reprint.pdf`.

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

[12] R. Lotufo and A. Falcão. 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, pages 341–350. Kluwer Academic Publishers, June 2000.

[13] A. Meijster and J. B. T. M. Roerdink. *A Proposal for the Implementation of a Parallel Watershed Algorithm - CAIP'95*, volume 970 of *Lecture Notes in Computer Science*, pages 790–795. Springer Berlin / Heidelberg, 1995.

[14] A. Meijster and J. B. T. M. Roerdink. A disjoint set algorithm for the watershed transform. In *Proc. IX European Signal Processing Conf EUSIPCO '98*, pages 1665–1668, 1998.

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

[16] A. Moga, T. Viero, B. Dobrin, and M. Gabbouj. Implementation of a distributed watershed algorithm. In J. Serra and P. Soille, editors, *Mathematical morphology and its applications to image processing*, volume 2, pages 281–288. Kluwer Academic Publishers, 1994.

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, 2008.

[18] NVIDIA. Cuda programming guide, 2.1. Technical report, 2009. `http://developer.download.nvidia.com/compute/cuda/2\_1/toolkit/docs/NVIDIA\_CUDA\_Programming\_Guide\_2.1.pdf`.

[19] NVIDIA. Cuda technical training. volume i: Introduction to cuda programming. Technical report, 2009. `http://www.nvidia.com/docs/IO/47904/VolumeI.pdf`.

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

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

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

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

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