Advances on Watershed Processing on GPU Architecture

André Körbes, Giovani Bernardes Vitor, Roberto Alencar Lotufo, Janitovaqueiro Ferreira

Advances on Watershed Processing on GPU Architecture

André Körbes[1], Giovani Bernardes Vitor[2], Roberto de Alencar Lotufo[1], and Janito Vaqueiro Ferreira[2]

[1] University of Campinas, School of Electrical and Computer Engineering (Department of Computer Engineering and Industrial Automation), Campinas - SP, Brazil,
korbes@dca.fee.unicamp.br, lotufo@dca.fee.unicamp.br
[2] University of Campinas, School of Mechanical Engineering, Campinas - SP, Brazil,
giovani@fem.unicamp.br, janito@fem.unicamp.br

Abstract. This paper presents an overview on the advances of watershed processing algorithms executed on GPU architecture. The programming model, memory hierarchy and restrictions are discussed, and its influence on image processing algorithms detailed. The recently proposed algorithms of watershed transform for GPU computation are examined and briefly described. Its implementations are analyzed in depth and evaluations are made to compare them both on the GPU, against a CPU version and on two different GPU cards.

1 Introduction

The watershed transform is a well known image segmentation tool, used to compute connected components, generally out of the gradient of the input image. Several algorithms exist to process the transform sequentially [4], [8], and since the development of the first fast transforms, effort is put on research of parallel algorithms and architectures. This has led to the recent development of techniques used on clusters [1] and FPGA [9].

The rise of GPU cards programmable for generic purposes, with low cost and high computational power, and with a trend of improvement of the hardware without severe changes to the architecture and programming model, has established a new platform of interest. These features sets the direction of advances on the watershed algorithms for improving performance, using the concepts and definitions established on literature, exploring the architecture for the procedures that best fit, and showing some new and revisited approaches [3], [13], [12].

This paper works on studying this architecture and the advances made on the watershed processing and the algorithms that use it. The usage of such models influences how the problems must be treated, algorithms designed and programs implemented. The watershed transform is implemented on two forms: one with four steps, with different levels of division of tasks, ranging from the neighborhood to global processing; and one that uses only a neighborhood operation, switching values until the optimal solution is found and stabilizes. These different levels of decomposition are analyzed, and the implementation strategies described.

This paper is organized as follows: Sec. 2 discusses the influence of the GPU architecture on the development of image processing algorithms; Sec. 3 reviews the literature of parallel watershed algorithms and discusses the two algorithms evaluated in this work; Sec. 4 discusses in details the implementation of both algorithms; Sec. 5 discusses the measurements made, and exposes the performance obtained on two GPU cards and a CPU; and lastly, Sec. 6 presents the conclusions drawn from this work.

2 Architecture Influence

Given many threads executing, each one having a small computing power, with some shared resources and high bandwidth for communication, the GPU architecture is well suited for parallel algorithms of fine granularity, where each thread will process one or a few pixels of the output image. Regarding the communication between threads, as they share a common memory, this area is used when any kind of information needs to be transferred between them. Also, whenever using GPU memory to communicate with the host CPU, there is a considerable overhead on the copy operation that may degrade the speedup obtained.

As many image processing algorithms, the watershed transform processes a neighborhood around every pixel. Also, as the blocks of threads divide the image on sub-images that are loaded on shared memory, these must have a border with the pixels from the adjacent blocks. This concept is illustrated on [7] as the apron for each block. The apron is constituted such that blocks are processed with overlapped data, with each block responsible for processing an area of the image, but loading data from adjacent blocks.

The modern GPU architecture evolved to a point where there are several forms of running programs on it, with the most noticeable frameworks and languages being CUDA and OpenCL, which are very similar to C and C++. Both models provide access to the same memory model with similar instructions. The programs developed for this work used CUDA, and the source codes are available on the Internet at `http://www.adessowiki.org`.

3 Parallel Watershed Algorithms

The watershed transform is a very data intensive task, that even with quasi-linear algorithms is time-consuming. Since its introduction by Vincent and Soille [10] effort is put on researching faster algorithms as well as parallel approaches. An extensive survey on parallel approaches is given by Roerdink and Meijster [8]. Since then, faster algorithms have been created, as well as new parallel approaches [9], [1]. On the past few years, other parallel watershed algorithms have been introduced, specifically for GPU architectures [3], [13], [12], using different strategies, designed for the restrictions of the platform.

This paper analyzes two algorithms: one inspired by the drop of water paradigm and depth-search approaches, based on [12], named **DW**; and one based on cellular automata to process a shortest-path forest with sum cost function [3], named **CA**. Both algorithms are tuned to better suit the architecture of modern GPUs.

The **DW** algorithm uses different steps of processing, which allows faster processing of operations with different levels of data independence. These steps are briefly: (i) identification of the neighbor with the lowest gray level for each pixel (this corresponds to finding the downstream for each pixel, see [6], [8]); (ii) propagation of the downstream of the plateau border pixels to inner pixels according to the geodesic distance; (iii) labeling of regional minima by union-find; and (iv) labeling of non-minima pixels by the path created to every minima. These steps are organized and designed to maximize performance on a GPU architecture. Step (i) processes each pixel independently, scanning the neighborhood for the lowest gray level; step (ii) is the step less suited for parallelism, caused by the need to uniformly calculate the geodesic distances, creating synchronization barriers on the threads; step (iii) consists of labeling the regional minima pixels using the union-find strategy for merging connected components; step (iv) consists on compressing paths from each pixel to the regional minima associated with the path.

The **CA** algorithm is based on the algorithm of Ford-Bellmann to calculate a shortest-path forest, using a single relaxing procedure, performed until convergence of the solution. Therefore, the proposed implementation consists of a single step, executed until stabilization of the solution. Nevertheless, for the **CA** algorithm produce a correct watersheds transform, its cost function must be adapted to consider the weight of each edge as the topographic distance [6]. Also, as this function does not manage plateaus, either the image must be preprocessed by lower completion, or the cost function must be further modified to consider the lexicographic cost as a secondary component [5]. For greater adherence with the original proposal, the lower completion is used prior to watershed execution.

4 Implementation Details

This section discuss the implementation of watershed algorithms on the SPMD and SIMD models, considering the access to the memory layers of the GPU, that constitutes the problem of communication between blocks. It is also presented the methodology used for development, highlighting the border treatment, the replication of data, and the data flow between memories.

The GPU memory model is hierarchically divided, and programs must consider these different levels and its characteristics to take advantage of its features, ultimately to reach the best performance. Using the CUDA framework, the code that is executed on the GPU does not have access to the CPU RAM memory, being an area of access exclusive of the host code. Therefore, for any processing on the GPU, the data must be transferred between host (CPU) memory and device (GPU) memory. The GPU global memory is the area writeable by the CPU, and it also provides special access modes of texture and constant, which are cached and only readable by the device code. There are three other memory areas available to the device code: the registers, the local and the shared memory. These memories have very limited size, but are located close to each stream multiprocessor of the GPU, and provide high speed of access. Registers and local memory are bound to the scope of each thread, whereas the shared memory, with scope on the block, is used for cooperation and communication between threads, with also a high speed access.

Based on the communication relations between different types of memory, the host and device code that composes the watershed processing steps are modeled to use as most as possible the shared memory area. Initially the image is loaded by the host on a texture memory, and the neighborhood relation data on the constant memory. As the blocks process parts of the image, it is loaded to the shared memory. From this, the results are computed on registers, and finally written on the global memory. This flow is used on all kernels of the algorithms developed, except those that do not use a neighborhood relation, where the global memory is used directly.

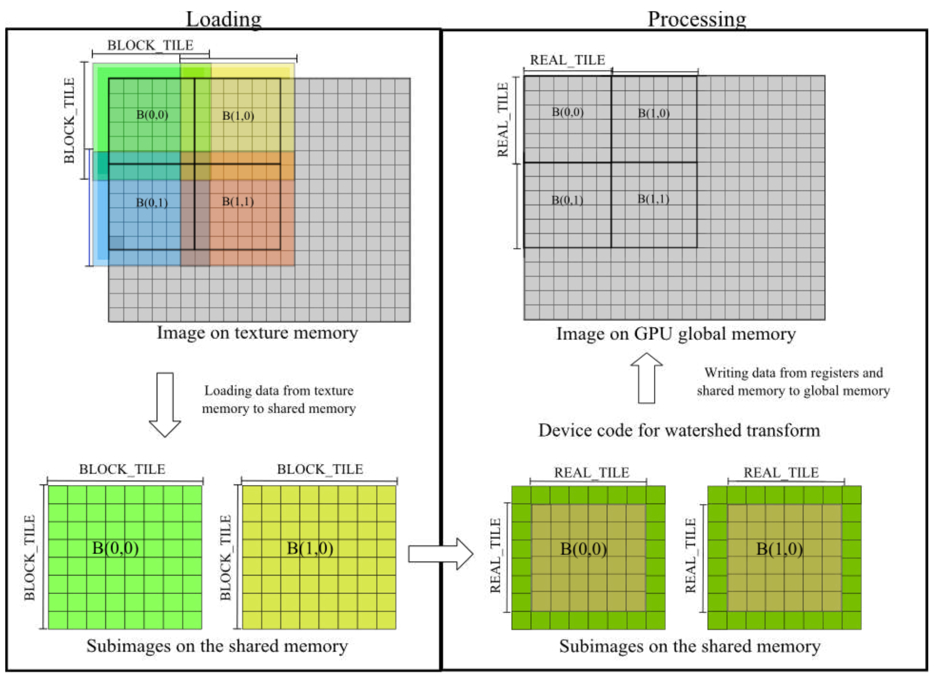
With the decomposition of the image on sub-images arises the problem of management of the borders of each block, as each pixel demands data from the neighbors that may be contained on another block, to complete its computation. The strategy used is to load an overlapping area, called the apron, as described on [11], [7], according to the neighborhood relation used. The borders values are either the values of pixels of the adjacent block, or in the case of image borders, a constant and predefined mask value is used.

The overlapping scheme, with borders loaded for every block by the device code, is presented on Fig. 1. This scheme is divided on two phases: the first one loads the data to the shared memory, while the second processes the shared memory and stores the results back on global memory. The loading is performed on blocks of $BLOCK\_TILE$ width and height, which corresponds to the size of blocks of threads used on the CUDA device code invoking. The processing is performed on fewer threads, on a block of $REAL\_TILE$ width and height, which corresponds to the dimensions of the block of data that is actually processed, without overlapping, ignoring border pixels.

4.1 Algorithm DW

The **DW** algorithm is based on the four steps described on Sec. 3 and extensively detailed on [12]. To implement this algorithm, six programs of device code (kernels) were created:

– *donwstreamCalc*: calculates the downstream for each pixel and then uses the data already loaded on memory to propagate it to plateaus internal pixels, until stability inside the block. This kernel is executed once and is associated to the first and second steps of the algorithm.
– *plateauPropagate*: continues the propagation of the downstream to plateaus internal pixels inside the block, respecting the geodesic distance to the borders. As the distance propagation may require block communication, this kernel is invoked until stabilization and is associated with the second step.



Loading
Processing
BLOCK_TILE
B(0,0)
B(1,0)
B(0,1)
B(1,1)
Image on texture memory
Loading data from texture memory to shared memory
Subimages on the shared memory
REAL_TILE
Image on GPU global memory
Writing data from registers and shared memory to global memory
Device code for watershed transform

Fig. 1. Overlapping of data loaded on shared memory to process border pixels correctly. It is used extra threads inside each block to load data, which are discarded and do not perform the computation.

– *mergeRegions*: creates connected components with the minima pixels, using the index as the reference for the group. The regional minima may spread across different blocks, so this kernel is invoked until stability, paired with the pathCompression kernel. This kernel is associated with the third step of the algorithm.
– *pathCompression*: updates the reference of each pixel directly to the connected component root. This avoids unnecessary memory accesses when labeling the image and merges connected components that spread across blocks. This kernel is associated with the third and fourth steps of the algorithm. On the third step this is invoked until stabilization of the solution, and only once for the fourth step.
– *labelPixel*: updates the label of each pixel, according to the reference of its group. For regional minima processing, this connects minima that spread on more than a single block, unifying the labels. This kernel is associated with the third step of the algorithm.
– *indexAdjust*: updates the references matrix to allow the pathCompression kernel to work on all pixels, rather than only on the regional minima, as a preparation for the fourth step.

The minima and paths labeling problem is addressed in many ways on both sequential and parallel algorithms, being a usual bottleneck, where common strategies have not obtained success on the GPU [12]. For that matter, these steps on the current approach are performed using the labeling algorithm of [2] , based on a reference list for path compressing and representative propagation. This algorithm is implemented through the kernels *mergeRegions*, *pathCompression* and *labelPixel* presented before.

To better understand the bottlenecks of this algorithm, each kernel and API call is timed, and a profile of the amount of time spent on each step is produced, shown on Fig. 2. The first kernel executed, downstreamCalc, takes around 60% of the total time of execution. This is expected, as this step calculates both the downstream and solves small plateaus that are contained inside a block. The next kernel invokes, for plateau propagation, take around 16% of the total time, with 8 calls, each with a smaller computation time, until stabilization. The regional minima and catchment basin labeling are solved with a small percentage of the total time, indicating an algorithm highly adapted for the task in parallel. The first kernel may look as a severe bottleneck, compared to the second. However, it was observed that this kernel solves many small plateaus, reducing the amount of work to be done on the plateau propagation kernel.

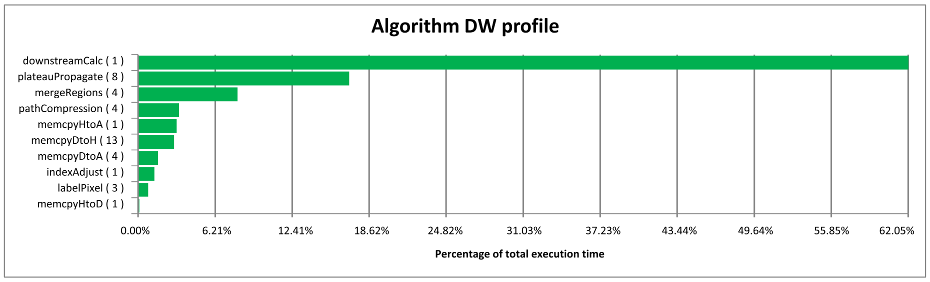


Fig. 2. Percentage of the total execution time taken by each kernel and API call of the DW algorithm.

Clearly the division of the algorithm on steps described on Sec. 3 does not match exactly the kernels described on this section, as these are designed to minimize the amount of time consumed on memory copies and maximize the processing on each pixel on a single call. However, those steps divide the problem logically, leading to other solutions, such as the sequential algorithm that is used on the comparisons of the next section. This implementation is also based on the drop of water principle and is divided on the aforementioned steps, and is not parallelized on CPU threads, thus running on a single core. The solutions provided by both algorithms are equivalent and valid.

4.2 Algorithm CA

The **CA** is based on the algorithm described on Sec. 3 and detailed on the work of Kauffmann and Piche [3]. This algorithm, inspired on the algorithm of Ford-Bellmann for calculating shortest-path forests with the sum path cost function, iterates performing changes on the solution whenever a path of less cost is found, until stabilization. To apply this concept on the watershed transform, the weight of each edge of the inner graph must be associated with the topographical distance, as described by Meyer [6]. To address the issue of the cost on each edge depending on the minimal value of the neighborhood, the implementation is based on two kernels:

– *initialization*: initializes the cost for each pixel, depending on whether they are a regional minima or not. Also, scans the neighborhood to store the minimal value, necessary to compute the cost on each edge, during the processing stage.
– *iterate*: process each pixel looking for paths of less cost. This kernel process each block until stabilization of the sub-image solution, and also is invoked as many times as necessary for stabilization of the whole image solution.

To use this algorithm, it is necessary to provide markers, either defined by the user or the regional minima. However, as the kernel that propagates the solution is invoked as many times as necessary, the greatest path on the inner graph will bind the execution time. Thus, the performance is proportional to the ratio of the area of catchment basins by the area of regional minima. To implement this algorithm efficiently, the greatest improvement is the preprocessing of the cost, which leads to several less memory accesses.

5 Performance Measurements

This section presents the experiments performed with the watershed algorithms discussed on this paper, divided on three subsections: first, the algorithms are measured and compared one against the other on the GPU; second, algorithm **DW** is compared against a sequential implementation also based on the drop of water principle; third the algorithm **DW** is timed on two different GPU cards, to evaluate the impact of the evolution of hardware.

The hardware used is: an NVIDIA GTX 295 GPU card with 240 cores running CUDA 2.30; an NVIDA GTX 470 GPU card with 448 cores running CUDA 3.20; and a CPU AMD Phenom II X3 CPU of 2.6Ghz clock and 7.5Mb of cache with 4 GB of RAM. The measurements of GPU algorithm execution also consider memory transfer from CPU to GPU. For the **CA** algorithm, the computation of regional minima is discarded. This computation is not measured as it is out of scope of this work and would implicate on an additional processing that may not be useful if it is supplied markers obtained from other methods, such as user input. The algorithms were run on the images lena, cameraman, peppers and baboon. These images have different profiles, with varying number of regional minima and extension of plateaus. The measurements were averaged, and the standard deviation presented indicates the variation on each image size. The images were resized to 64x64, 128x128, 256x256, 512x512, 1024x1024 and 2048x2048.

5.1 GPU comparison

This experiment intends to compare the algorithms **DW** and **CA** executing on the GTX 295 card. Tab. 1 shows the average times obtained for each image size, and the standard deviation (STD) for both algorithms. Clearly, the algorithm **DW** produces results much faster and with smaller variation of time. The large standard deviation observed for **CA** is a direct consequence of its single-step propagation that will take as long to execute as the longer path on the image.

Image size	**Alg. DW (ms)**	**STD**	**Alg. CA (ms)**	**STD**
64x64	1.065	0.064	1.099	0.094
128x128	1.224	0.126	1.871	0.320
256x256	2.431	0.624	4.872	0.946
512x512	5.801	1.035	19.290	5.224
1024x1024	19.918	3.866	109.532	33.655
2048x2048	88.262	23.322	657.115	253.307

Table 1. Comparison of times of algorithm DW and CA running on GTX295

The data of Tab. 1 is also shown as a chart, on Fig. 3, where the difference of times measured is better visualized. This chart presents the measured times in milliseconds (ms) by the total number of pixels on the image.

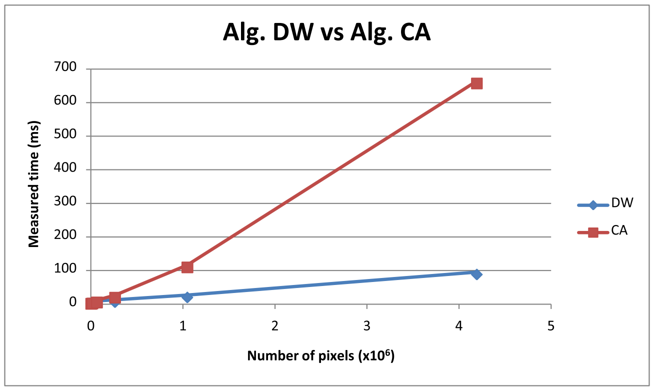


Fig. 3. Chart comparison between algorithms DW and CA running on GTX295.

The results of this experiment may also be compared with the results of the algorithm of Wagner and Godehardt [13], which was executed on a GTX280, a card very similar to the GTX295. This comparison is possible only for the 1024x1024 image size, which has nearly the same number of pixels of the cube of dimension 100 reported by the authors, with an execution time of 550ms.

5.2 GPU and CPU comparison

This experiment compares the algorithm **DW** running on the GTX 470 card against the CPU version. The measurements obtained on the executions are shown on Tab. 2. The speedup calculated is the ratio of the CPU by the GPU time. It is seen that for images smaller than 256x256, the speedup is less than two, and given that the absolute times measured are very small, the usage of a GPU algorithm may not prove useful. However, for images larger than that, the speedup is increased up to 6.5, and the absolute times make this acceleration even more useful, which can be seen on the chart of Fig. 4.

Image size	GPU (ms)	STD	CPU (ms)	STD	Speedup
64x64	0.646	0.043	0.311	0.003	0.481
128x128	0.776	0.068	1.237	0.059	1.594
256x256	1.213	0.156	4.838	0.337	3.988
512x512	3.365	0.285	18.744	1.414	5.571
1024x1024	10.889	0.873	72.087	3.510	6.620
2048x2048	46.487	8.200	307.067	14.208	6.605

Table 2. Comparison of times between GPU GTX470 and CPU implementations

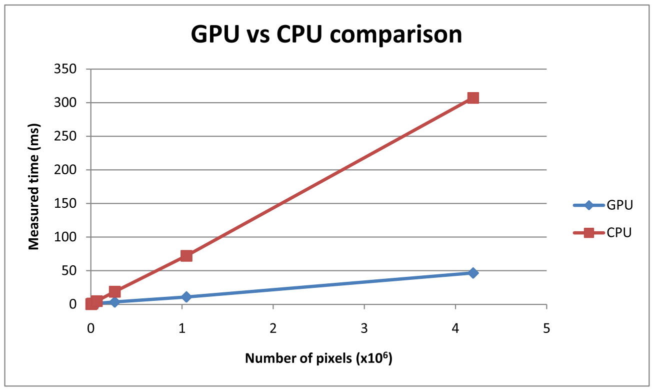


Fig. 4. Chart comparison of times between GPU and CPU implementations.

5.3 GPU evolution comparison

This experiment intends to show how the evolution of GPU cards impact the execution of the same code. On Tab. 3 it is shown the execution times of the algorithm **DW** on both cards. The speedup is calculated as the ratio of the measurement on the GTX 295 by the timing of GTX 470. The average of the speedup of the images is 1.780. This might be associated with the increase of the number of cores from 240 to 448, which gives a ratio of 1.86. However, it must be noted that there are several other improvements on the architecture of GTX 470, such as enhancing cache capabilities and the operation of atomic functions.

Image size	**GTX 295 (ms)**	**STD**	**GTX 470 (ms)**	**STD**	**Speedup**
64x64	1.065	0.064	0.646	0.043	1.649
128x128	1.224	0.126	0.776	0.068	1.578
256x256	2.431	0.624	1.213	0.156	2.004
512x512	5.801	1.035	3.365	0.285	1.724
1024x1024	19.918	3.866	10.889	0.873	1.829
2048x2048	88.262	23.322	46.487	8.200	1.899

Table 3. Comparison of times between two different GPU cards

5.4 Algorithm scalability comparison

This experiment focuses on how the algorithms **DW** and **CA** perform on the same images, filtered to have less regional minima. To evaluate this, the four images were used on the size of 512x512 and filtered to have different number of regional minima, ranging from 562 to 24848. However, as the images have different profiles, only the image **Lena** is used to show the effect of filtering, as its results were the most affected. The times measured are presented on the chart of Fig. 5. This chart shows the measured times by the number of regional minima. This chart confirms the first experiment, that algorithm **DW** is faster that algorithm **CA**, and shows that the algorithm **DW** is more stable, with most times bound between 10 and 20 ms, with a standard deviation of 4.4 ms, while algorithm **CA** is bound between 20 and 50 ms, with a standard deviation of 9.3 ms. The effect of severe variation of running time observed on algorithm **CA** is not observed for every image tested. Nevertheless, such variation was not observed for algorithm **DW** for any of the images.

6 Conclusion

This paper presented an analysis of modern watershed algorithms designed for GPU architectures with some considerations on the implementation and design of such algorithms. The issues of implementing a parallel watershed algorithm

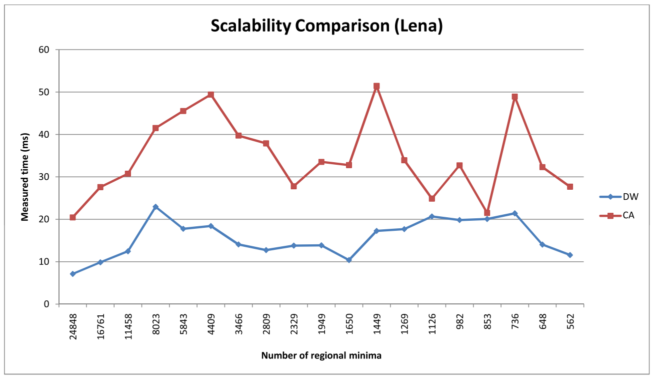


Fig. 5. Chart comparison of measured times by the number of regional minima on image Lena of size 512x512.

were discussed, and two implementations using the CUDA framework were detailed.

On the performance measurements, three scenarios were used, comparing algorithms on the GPU, the GPU against the CPU and GPU cards of different configurations. The comparison on the GPU showed that having steps specialized for each task, and less dependence on stabilization produces faster results. The comparison with the CPU showed that studies of GPU algorithms - i.e. algorithms that use the SPMD model - may lead to reasonable speedups. The comparison of evolution of cards showed that using the number of cores of a board is a reasonable measure of normalization of times, and that the evolution of hardware may further reduce the execution times and restrictions that currently exist.

Also, it has been observed that, because of the levels of speedup achieved, especially when considering the evolution of the GPUs, the cost of implementation of these algorithms is rewarded. In fact, the usage of CUDA and/or OpenCL technologies, with comprehension of the general architecture, enables the development of programs that are not strictly bound to the hardware, and may run on several cards with the benefits of speedup. As a consequence, future works are seen on the investigation of more algorithms focused on the SPMD model, with implementations not dependent on special hardware configuration and also suitable for 3D volumes.

References

1. Galilée, B., Mamalet, F., Renaudin, M., Coulon, P.Y.: Parallel asynchronous watershed algorithm-architecture. IEEE Transactions on Parallel and Distributed Systems 18(1), 44–56 (2007)
2. Hawick, K.A., Leist, A., Playne, D.P.: Parallel graph component labelling with GPUs and CUDA. Parallel Computing 36(12), 655–678 (Dec 2010)

3. Kauffmann, C., Piche, N.: A cellular automaton for ultra-fast watershed transform on GPU. In: 19th International Conference on Pattern Recognition. pp. 1–4. IEEE Computer Society, Tampa, Florida, USA (Dec 2008)
4. Körbes, A., Lotufo, R.: Analysis of the watershed algorithms based on the breadth-first and depth-first exploring methods. In: SIBGRAPI'09. pp. 133–140. IEEE Computer Society, Rio de Janeiro, Brazil (Oct 2009)
5. Körbes, A., Lotufo, R.: On watershed transform: Plateau treatment and influence of the different definitions in real applications. In: Proceedings of the 17th International Conference on Systems, Signals and Image Processing. pp. 376–379. Rio de Janeiro, Brazil (Nov 2010)
6. Meyer, F.: Topographic distance and watershed lines. Signal Processing 38(1), 113–125 (1994)
7. Podlozhnyuk, V.: Image Convolution with CUDA. NVIDIA (June 2007), `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/convolutionSeparable.pdf`
8. Roerdink, J.B.T.M., Meijster, A.: The watershed transform: definitions, algorithms and parallelization strategies. Fundam. Inf. 41(1-2), 187–228 (2000)
9. Trieu, D.B.K., Maruyama, T.: Real-time image segmentation based on a parallel and pipelined watershed algorithm. Journal of Real-Time Image Processing 2(4), 319–329 (December 2007)
10. Vincent, L., Soille, P.: Watersheds in digital spaces: An efficient algorithm based on immersion simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence 13(6), 583–598 (1991)
11. Vitor, G.: Rastreamento de alvo móvel em mono–visão aplicado no sistema de navegação autônoma utilizando GPU. Master's thesis, Universidade Estadual de Campinas, Campinas, São Paulo, Brazil (Mar 2010)
12. Vitor, G., Ferreira, J., Körbes, A.: Fast image segmentation by watershed transform on graphical hardware. In: Proceedings of the 30^{o}CILAMCE. Armação dos Búzios, Brazil (Nov 2009)
13. Wagner, B., Godehardt, M.: Cell reconstruction on stream computing architectures. In: ISMM'2009 Abstract Book. pp. 45–48. University of Groningen (2009)