

FAST IMAGE SEGMENTATION BY WATERSHED TRANSFORM ON GRAPHICAL HARDWARE

Giovani Bernardes Vitor

Janito Vaqueiro Ferreira

giovani,janito@fem.unicamp.br

Universidade Estadual de Campinas

Faculdade de Engenharia Mecânica

DMC -Departamento de Mecânica Computacional

Rua Mendeleev, 200

CP 6122 - Campinas, SP, Brasil

André Körbes

korbes@dca.fee.unicamp.br

Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

DCA -Departamento de Engenharia de Computação e Automação Industrial

Av. Albert Einstein - 400

CP 6101 - Campinas, SP, Brasil

Abstract. *The watershed transform is widely used for image segmentation on computer vision applications. However, sequential watershed algorithms are not suitable for fast applications, once they are one demanding part of several tasks. This paper proposes two parallel algorithms for the watershed transform focused on fast image segmentation using off-the-shelf GPUs. In this sense, these algorithms aims for a speedup by mixing several techniques of the fastest procedures on both sequential and parallel fields. Both algorithms has four major steps, the parallel version processed on SIMD, and the hybrid version mixing parallel and sequential approaches. The experimental results obtained show that the hybrid version is faster, taking advantage of the most appropriate hardware for each task.*

Keywords: watershed transform, image segmentation, GPGPU

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 one of the hardest ones in image processing. Some of its inherent problems are illumination variation through image sequences, dynamic change of background where the camera and/or target moves, 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.

There are several methods for instantaneous image segmentation on literature, such as background subtraction and its variations, which are very robust but restricted to static cameras, not applicable to a moving scene and camera, the aim of this work. Extraction by shape and colour also exist, however are more sensitive to the previously mentioned variations. An alternative is to join multiple techniques, aiming for gain in robustness. Nevertheless, this type of process carries a disadvantage on the time required for calculations, exceeding CPU capacity for instantaneous video processing. Given that need for a robust yet fast procedure, parallel implementations seem an obvious choice.

One significant part of some segmentation processes goes through the use of a watershed transform. With the use of special hardware, such as GPUs, it can be calculated faster, once these are highly specialised and optimised devices for graphic processing. This paper aims for the development of a watershed transform algorithm suited for GPU processing, developed with NVIDIA's CUDA platform, in four major steps. Two algorithms are proposed, a fully parallel SIMD algorithm and another hybrid approach, taking advantage of the best core for each necessary task. These proposals are compared with each other and with recently developed sequential watershed algorithms.

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 processing with the CUDA libraries. Sec. 4. exposes both algorithms developed, explaining its operations. Sec. 5. presents the timing results obtained, comparing the algorithms. Lastly, Sec. 6. depicts conclusions obtained with this work.

2. Watershed Transform

Many definitions for the watershed transform exist in literature (Vincent and Soille, 1991; Meyer, 1994; Lotufo and Falcão, 2000; Falcão et al., 2004; Bieniek and Moga, 1998; Cousty et al., 2009) 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 (2000) and the union-find one of Meijster and Roerdink (1998). 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, 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. (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. 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. (2007) proposed an improved algorithm on the sense of pixel visitation by optimising the preliminary works previously mentioned. 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 et al. (2005), Lin et al. (2006), Bieniek and Moga (2000) and Meijster and Roerdink (1998) 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 (Meijster and Roerdink, 1995; Bieniek et al., 1997). Roerdink and Meijster (2000) extensively surveyed the literature on this problem. More recently, the evolution of sequential algorithms along with hardware minimised this problem. 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 (Trieu and Maruyama, 2007; Galilée et al., 2007).

The initial work of Meijster and Roerdink (1995) is based on the work of Vincent and 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 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. (1997) proposed another parallel algorithm with a modified definition, that is the local condition watershed transform. 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. (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 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 fast applications, Trieu and 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.

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 (Cooperman and 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, 2009a).

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 (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, as shown in Fig. 1.

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.

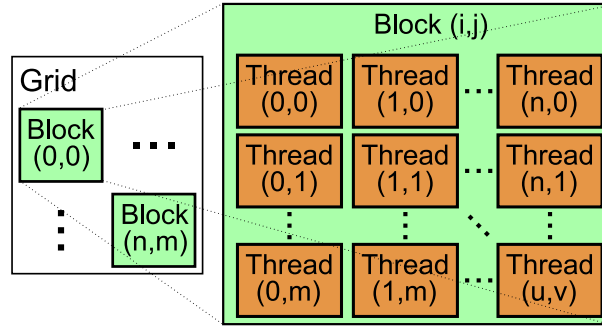


Figure 1: Structuring of data for implementation in parallel

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. 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. Parallel Watershed Algorithm

In this section we propose an algorithm for fully parallel SIMD implementation and a hybrid approach, based on both GPU and CPU. Firstly, the motivations and inspirations are presented. Next, the notation used is explained, and the algorithms 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 instantaneous video analysis. 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 by Bieniek and 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 along with the labelling of minima by neighbourhood evaluation, 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, in a SIMD way, and there is a synchronisation step, where the next statement after the **for all** is only processed after every parallel process has terminated. 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, propagating uniformly from the borders; minima labelling by maximal neighbour address and pixel labelling by flooding from minima. The input image is called I , and the output labelled image is called lab , which is also used for storing addresses.

A similar algorithm has been proposed recently, though with modified strategies for labelling (Körbes et al., 2009).

Algorithm 1: Parallel watershed transform

```

// First Step
1:  $PLATEAU \leftarrow +\infty$ 
2: for all  $p \in D$  do
3:   if  $\exists q \in N(p) : I(q) < I(p)$  and  $I(q) = \min_{q' \in N(p)} I(q')$  then
4:      $lab(p) \leftarrow -q$ 
5:   else
6:      $lab(p) \leftarrow PLATEAU$ 
7:   end if
8: end for

// Second step
9: while  $lab$  is not stable do
10:   $lab' \leftarrow lab$ 
11:  for all  $p \in D : lab(p) = PLATEAU$  do
12:    if  $\exists q \in N(p) : lab(q) \leq 0$  and  $I(q) = I(p)$  then
13:       $lab'(p) \leftarrow -q$ 
14:    end if
15:  end for
16:   $lab \leftarrow lab'$ 
17: end while

// Third step
18: for all  $p \in D : lab(p) = PLATEAU$  do
19:   $lab(p) = p + 1$ 
20: end for
21: while  $lab$  is not stable do
22:  for all  $p \in D : lab(p) > 0$  do
23:    for  $q \in N(p)$  do
24:      if  $lab(q) > lab(p)$  then
25:         $lab(p) \leftarrow lab(q)$ 
26:      end if
27:    end for
28:  end for
29: end while

// Fourth step
30: while  $lab$  is not stable do
31:  for all  $p \in D : lab(p) \leq 0$  do
32:     $q \leftarrow -lab(p)$ 
33:    if  $lab(q) > 0$  then
34:       $lab(p) \leftarrow lab(q)$ 
35:    end if
36:  end for
37: 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 (NVIDIA, 2009b). 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 (NVIDIA, 2009a): hardware-based linear interpolation and border management; potential higher bandwidth, once its cached. An observation is that the shared 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. The kernels were developed in two 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 other 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.

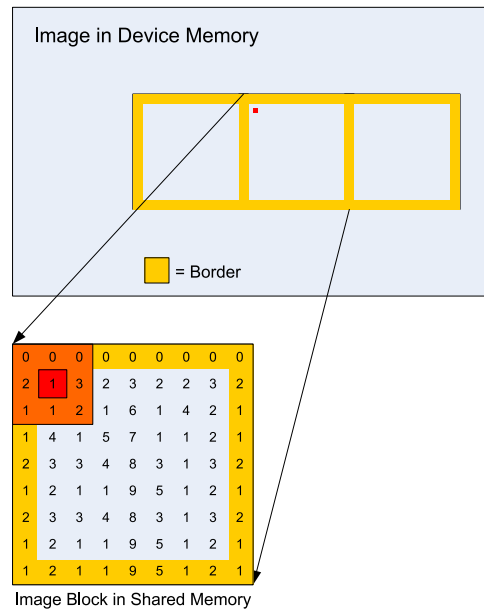


Figure 2: Border behaviour to include neighbours outside of the block

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, were these data is 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).

Along with these specifications, the algorithm is dependant on stabilisation steps, which are verified on the CPU. As the experimental results presented on Sec. 5. show, these steps contribute heavily for degrading the performance of Algorithm 1. Therefore, we propose a second algorithm, on a hybrid approach, were steps 1 and 2 are the same as Algorithm 1, processed on the GPU, and steps 3 and 4 are modified versions, processed on the CPU, achieving better performance. These steps follow general strategies of minima (connected component)

labelling and path labelling found on algorithms such as Bieniek and Moga (1998), Osma-Ruiz et al. (2007) and Lin et al. (2006).

Algorithm 2: Hybrid watershed transform

```

// First Step
// Same as first step of Algorithm 1
1:  $PLATEAU \leftarrow +\infty$ 
2: for all  $p \in D$  do
3:   if  $\exists q \in N(p) : I(q) < I(p)$  and  $I(q) = \min_{q' \in N(p)} I(q')$  then
4:      $lab(p) \leftarrow -q$ 
5:   else
6:      $lab(p) \leftarrow PLATEAU$ 
7:   end if
8: end for

// Second step
// Same as second step of Algorithm 1
9: while  $lab$  is not stable do
10:   $lab' \leftarrow lab$ 
11:  for all  $p \in D : lab(p) = PLATEAU$  do
12:    if  $\exists q \in N(p) : lab(q) \leq 0$  and  $I(q) = I(p)$  then
13:       $lab'(p) \leftarrow -q$ 
14:    end if
15:  end for
16:   $lab \leftarrow lab'$ 
17: end while

// Third step
18:  $basins \leftarrow 1$ 
19: for  $p \in D$  do
20:   if  $lab(p) = PLATEAU$  then
21:      $lab(p) \leftarrow basins$ 
22:      $basins \leftarrow basins + 1$ 
23:      $QUEUEPUSH(p)$ 
24:     while  $QUEUEEMPTY() = \text{False}$  do
25:        $q \leftarrow QUEUEPOP()$ 
26:       for  $u \in N(q)$  do
27:         if  $lab(u) = PLATEAU$  then
28:            $lab(u) \leftarrow lab(p)$ 
29:            $QUEUEPUSH(u)$ 
30:         end if
31:       end for
32:     end while
33:   end if
34: end for

// Fourth step
35: for  $p \in D$  do
36:   if  $lab(p) \leq 0$  then

```

```

37:      $q \leftarrow p$ 
38:     while  $lab(q) \leq 0$  do
39:          $q \leftarrow -lab(q)$ 
40:     end while
41:      $u \leftarrow p$ 
42:     while  $u \neq q$  do
43:          $v \leftarrow u$ 
44:          $u \leftarrow -lab(u)$ 
45:          $lab(v) \leftarrow lab(q)$ 
46:     end while
47: end if
48: end for

```

The procedures used for minima and path labelling on Algorithm 2 optimise performance given some conditions explained here. For most images, regional minima are composed by more than a few pixels, making the stabilisation step for minima labelling on Algorithm 1 take several turns, while on Algorithm 2 this analysis is done only once, and pixels receive its final labels as soon as visited. The same is valid for step 4, where long paths on Algorithm 1 take its length in number of turns to complete the labelling, while on Algorithm 2 pixels are visited only twice to receive its final labels. However, for step 2 a sequential approach would not improve the performance, as the border propagation is done similarly, though with a queue that restricts the set of pixels that are processed. Another consideration is that on every turn until stabilisation on Algorithm 1, each pixel is visited and processed, slowing its convergence. Along with these, CUDA is well suited for hybrid approaches, as mentioned on Sec. 3.1, and that improves the performance as seen on Sec. 5..

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 proposed on Algorithms 1 and 2, but without the watershed label, for matters of equivalence of definitions. 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 4Gb 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	0,1821	0,7256	3,1650	11,9489	45,8002	171,0729
Step 2	0,2751	1,1016	4,3633	17,8430	72,6984	306,1431
Step 3	0,0386	0,1897	0,4629	1,6862	4,4622	17,0799
Step 4	0,0554	0,1597	0,9337	3,5642	15,5848	70,0675
Total	0,5511	2,1765	8,9249	35,0422	138,5456	564,3633

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, specially on larger images. Moving to the fully parallel implementation on the GPU, developed according to Algorithm 1, Table 2 shows that the bottleneck on step 1 is completely mitigated, whereas for step 2 the performance depends on image

size, though showing small improvements. However, for steps 3 and 4, the algorithm fails on achieving speedups, in fact, these steps contribute heavily for a poor overall performance.

	64x64	128x128	256x256	512x512	1024x1024	2048x2048
Step 1	0.8645	0.9912	1.5086	3.3537	10.0368	37.8606
Step 2	0.2746	0.6246	2.2040	9.8523	56.3876	358.0210
Step 3	0.0671	0.1438	0.7173	4.1557	25.6360	405.1396
Step 4	0.2757	0.6314	2.0385	12.2949	21.6591	763.2722
Total	1.4820	2.3909	6.4683	29.6566	113.7195	1564.2934

Table 2: Parallel algorithm processed on the GPU

The observation that steps 1 and 2 of the parallel algorithm are more suited for GPU, and that steps 3 and 4 are more suited for the CPU, led to the hybrid algorithm, proposed on Algorithm 2. With the union of these steps, the resulting implementation achieved the time measurements of Table 3

	64x64	128x128	256x256	512x512	1024x1024	2048x2048
Step 1	0.8645	0.9912	1.5086	3.3537	10.0368	37.8606
Step 2	0.2746	0.6246	2.2040	9.8523	56.3876	358.0210
Step 3	0.0386	0.1897	0.4629	1.6862	4.4622	17.0799
Step 4	0.0554	0.1597	0.9337	3.5642	15.5848	70.0675
Total	1.2330	1.9652	5.1091	18.4564	86.4713	483.029

Table 3: Hybrid algorithm processed on the GPU+CPU

With the data collected on these three tables, a comparison may be done, checking the effective gain in time performance. This evaluation is seen on Table 4, where the algorithms are put aside and the relative gain is calculated for the average total time for each image size tested.

	CPU	GPU	GPU + CPU	Gain (%)
64x64	0.5511	1.4820	1.2330	-123.75%
128x128	2.1765	2.3909	1.9652	9.71%
256x256	8.9249	6.4683	5.1091	42.75%
512x512	35.0422	29.6566	18.4564	47.33%
1024x1024	138.5456	113.7195	86.4713	37.59%
2048x2048	564.3633	1564.2934	483.0290	14.41%

Table 4: Comparison between algorithms

The analysis of Table 4 must highlight two side-effects of GPU+CPU hybrid processing. The first one is for the smaller size, where this approach took more than twice the time than the sequential algorithm. This is due to memory transfer bandwidth and the GPU clock, significantly slower than the CPU clock. This effect is reduced with the increase of image size, until the larger one tested, which falls into another problem. For this image, the second step is very time consuming, due to the stabilisation, reducing the gain. However, for average sizes of images - typically those produced by video cameras - the hybrid approach shows a gain of an average of 40%, meaning a reduction of time of almost half of the sequential algorithm. Lastly, we present a graph on Fig. 3, showing the increase of gain with the increase of image size, on average sizes.

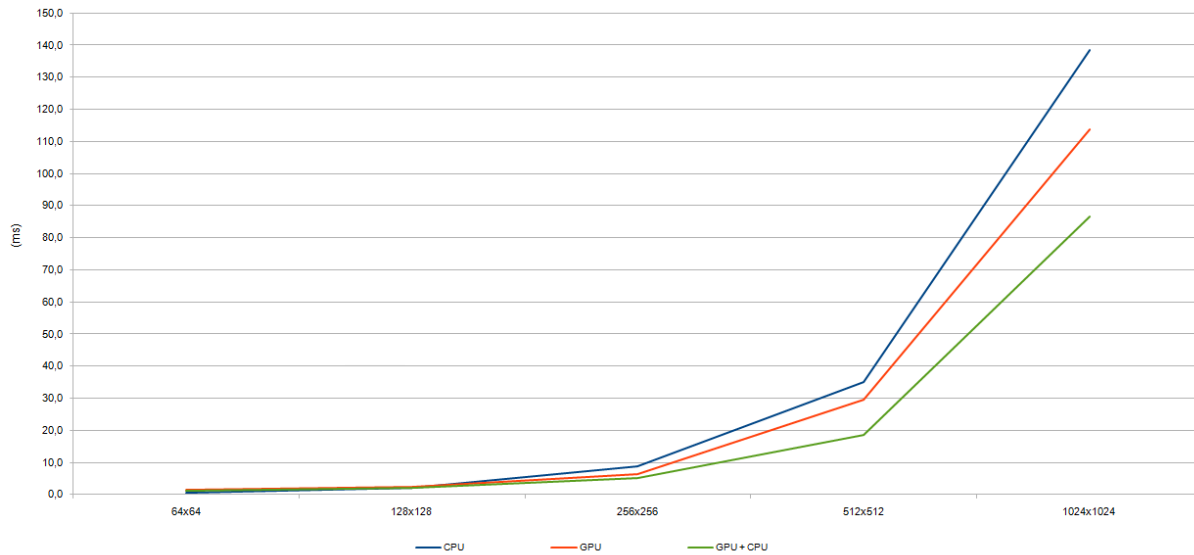


Figure 3: Graphic comparison of average time between algorithms

6. Conclusion

In this paper we proposed two parallel watershed transform algorithms for implementation on many-core architectures, such as GPUs, and together with a CPU, taking advantage of the best hardware for each task. We reviewed the state of the art on both sequential and parallel algorithms and proposed a new one inspired on some techniques presented. Based on the results obtained on this fully parallel algorithm in comparison with a sequential version, a hybrid version was proposed, with the first two steps operating on the GPU and the last two on the CPU. Our algorithms are described along with the discussion of some of the implementation issues. With the hybrid algorithm, we achieved a significant speedup on images of average sizes (up to 1024x1024 pixels), being almost twice faster than the sequential algorithm. As the scope of this project lies on video analysis, with images of sizes closer to 512x512, the average time obtained of 18ms allows the segmentation to work on a rate of 30fps with extra time for other processing necessary, such as filtering.

References

- Baggio, D. L., 2007. Gpu based image segmentation livewire algorithm implementation. Master's thesis, Technological Institute of Aeronautics, São 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*, vol. 33, n. 6, pp. 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*, vol. 31, n. 8, pp. 1362–1374.
- 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*, vol. 26, n. 1, pp. 19–29.
- Galilée, B., Mamalet, F., Renaudin, M., & Coulon, P.-Y., 2007. Parallel asynchronous watershed algorithm-architecture. *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, n. 1, pp. 44–56.
- Halfhill, T. R., 2008. *Parallel Processing with CUDA: Nvidia's high-performance computing platform uses massive multithreading*. NVIDIA. http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf.
- Körbes, A., Lotufo, R., Vitor, G. B., & Ferreira, J. V., 2009. A proposal for a parallel watershed transform algorithm for real-time segmentation. In *Proceedings of Workshop de Visão Computacional WVC'2009*.
- Lin, Y., Tsai, Y., Hung, Y., & Shih, Z., 2006. Comparison between immersion-based and toboggan-based watershed image segmentation. *IEEE Transactions on Image Processing*, vol. 15, n. 3, pp. 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*, vol. 38, n. 1, pp. 113–125.
- Nickolls, J., Buck, I., Garland, M., & Skadron, K., 2008. Scalable parallel programming with cuda. *ACM Queue*, vol. 6, n. 2, pp. 40–53.
- NVIDIA, 2009a. *CUDA Programming Guide, 2.1*. http://developer.download.nvidia.com/compute/cuda/2.1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf.
- NVIDIA, 2009b. *CUDA Technical Training. Volume I: Introduction to CUDA Programming*. <http://www.nvidia.com/docs/IO/47904/VolumeI.pdf>.
- 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*, vol. 40, n. 3, pp. 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.*, vol. 41, n. 1-2, pp. 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*, vol. 26, n. 9, pp. 1266–1274.
- 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*, vol. 2, n. 4, pp. 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*, vol. 13, n. 6, pp. 583–598.