A TREE OF SHAPES COMPUTATION ALGORITHM FOR MASSIVELY PARALLEL ARCHITECTURES

Edwin Carlinet, Baptiste Esteban

EPITA Research Laboratory (LRE)
14-16 rue Pasteur, Le Kremlin-Bicêtre, France

ABSTRACT

The tree of shapes of an image (ToS) is a powerful hierarchical representation of image. Being self-dual and contrast invariant, it is well-suited for several image processing tasks such as filtering, segmentation or object detection. It is a must-have tool in the toolbox of image processing practitioners, if only as a pre- or post-processing usage. Nevertheless, the ToS computation is a complex task, so far, limited to CPU. This is a major bottleneck in any deep-learning or real-time image processing pipeline that requires GPUs for speed. This limitation is due to a front propagation algorithm, used in the ToS construction, that is intrinsically sequential and not well-suited for massively parallel architectures. In this paper, we present a new approach to compute, end-to-end, the tree of shapes on massively parallel architectures that outperforms the existing algorithms. The parallelization strategies introduced in this paper can further be used to speed up many propagation-based algorithms such as distance transforms.

***Index Terms*—** Tree of Shapes, Massively Parallel Architectures, Mathematical Morphology, Distance transforms.

1. INTRODUCTION

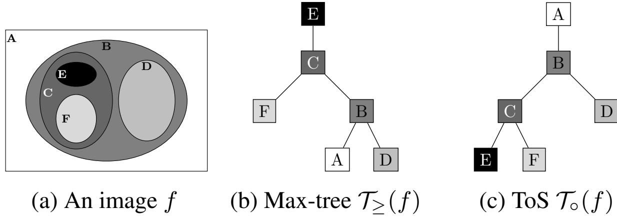
Hierarchical representations of image [1] are widely used tools in image processing. They allow representing objects of interest with different levels of details based on the inclusion relationship of the image regions. Among them, the tree of shapes (ToS) [2] encodes the inclusion relationship of the connected components of an image, and thus its level lines. It is used for several tasks such as image simplification [3], visual saliency [4], feature extraction [5], and image classification [6]. Fig. 1 illustrates a simple ToS-based application where a grain filter serves as a simple pre-filtering to accelerate text detection. Components are selected by their size, and the residual image contains text characters.

The Fast Level Line Transform (FLLT) [2] was the algorithm that builds the ToS by merging the min-tree and max-tree [7]. Latter, the Fast Level Set Transform (FLST) [8] relied on a region-growing approach to flood an image from its extrema and organize its level sets. In [9], the authors propose a top-down approach that starts from the border of the image and follows the level lines. Géraud *et al.* [10] first sort the level lines of an image according to their level of inclusion and then apply a Union-Find procedure to obtain the ToS. [11] and [12] follow a similar approach : they adapt the algorithm from [10] by first building an *order* map and then extract the ToS from this map using a max-tree. It yielded the first linear and parallel ToS algorithms.

Fig. 1: Illustration of a grain filters with the ToS. Left: original image. Middle: image simplified with a grain filter. Right: residual image after filtering.

Nevertheless, none of the approaches discussed above is well-suited for massively parallel architectures. However, the latest present some interesting properties. First, they rely on the max-tree, for which an algorithm has been proposed for such architectures [13]. Then, the order map computation is based on a propagation procedure relying on a priority queue. This algorithmic scheme is used for several image processing tasks such as geodesic distance computation by solving the Eikonal equations in the Fast Marching Methods (FMM) [14]. This scheme is difficult to parallelize as it relies on a particular order to process the image pixels. To tackle this issue, Fast Iterative Methods (FIM) [15] have been proposed, relying on an iterative procedure so that the order of processing does not matter but also removing the need for an external data structure. With these barriers removed, the parallelization of the algorithm becomes straightforward.

In this paper, we propose the first algorithm to compute the tree of shapes on massively parallel architectures. To this aim, we present a novel iterative algorithm to compute the order map of the connected component inclusion in an image. We also present the parallelization strategy employed to effi-



(a) An image f (b) Max-tree $\mathcal{T}_{\geq}(f)$ (c) ToS $\mathcal{T}_{\circ}(f)$

Fig. 2: An image and its max-tree and tree of shapes

cient compute this map on such architectures.

This paper is organized as follows: first, we recall the definition of the tree of shapes and its computation algorithm in section 2. Then, we present the distance transform as an order map in section 3 and its parallelization in section 4. We compare the performance of our parallel algorithm with the sequential one in section 5. We finally conclude in section 6.

2. SEQUENTIAL TREE OF SHAPES COMPUTATION

2.1. Definitions

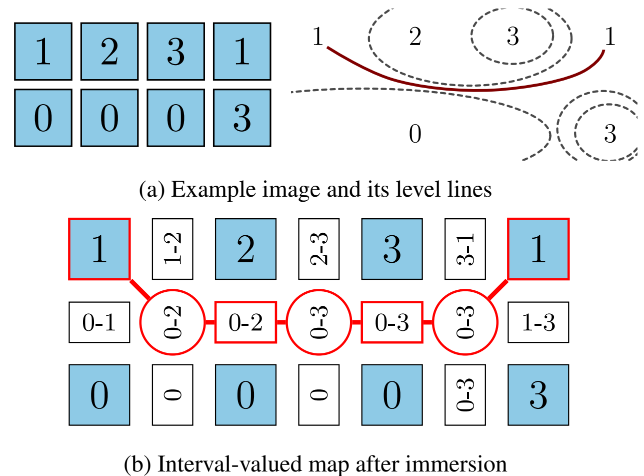
Let $f : \Omega \to \mathcal{V}$ be an image defined on a domain $\Omega \subset \mathbb{Z}^n$ and whose value set $\mathcal{V}$ is endowed with an order relationship $\leq$. Let $[f \geq \lambda] = \{p \in \Omega \mid f(p) \geq \lambda\}$ be the upper thresholds set and $[f \leq \lambda] = \{p \in \Omega \mid f(p) \leq \lambda\}$ be the lower thresholds set, with $\lambda \in \mathcal{V}$. The set of upper connected components is denoted by $\mathfrak{C}^{\geq} = \cup_{\lambda \in \mathcal{V}}\{X \mid X \in \mathcal{CC}([f \geq \lambda])\}$ and the set of lower connected components by $\mathfrak{C}^{\leq} = \cup_{\lambda \in \mathcal{V}}\{X \mid X \in \mathcal{CC}([f \leq \lambda])\}$, with the operator $\mathcal{CC}(X)$ returning the set of connected components in a set X. The max-tree $\mathcal{T}_{\geq}$ (resp. the min-tree $\mathcal{T}^{\leq}$) encodes the inclusion relationship of the components in $\mathfrak{C}^{\geq}$ (resp. $\mathfrak{C}^{\leq}$).

A shape $\mathcal{C}$ is a connected component of f with its holes filled and belongs to the set $\mathfrak{C}^{\circ} = \{\text{Sat}(X) \mid X \in \mathfrak{C}^{\leq}\} \cup \{\text{Sat}(X) \mid X \in \mathfrak{C}^{\geq}\}$ with $\text{Sat}(X)$ the hole-filling operator. The tree of shapes $\mathcal{T}_{\circ}$ encodes the inclusion relationship of the shapes of $\mathfrak{C}^{\circ}$, and thus the level lines of f. Fig. 2 illustrates a max-tree $\mathcal{T}_{\geq}$ and a tree of shapes $\mathcal{T}_{\circ}$ computed on an image.

2.2. Construction algorithm

This section describes the different steps to compute the tree of shapes based on the algorithm presented in [11]. To ensure the root of the tree encloses the whole image, the image is padded with a border whose value is usually set to the median value of the image bordering pixels.

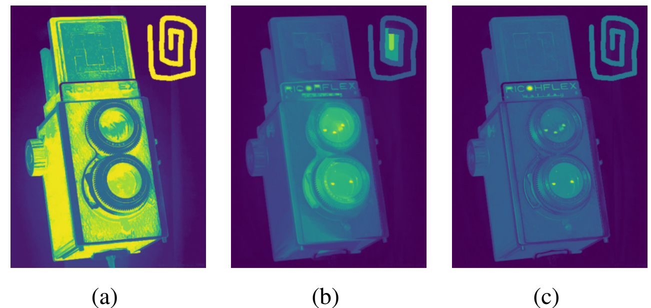
Immersion. The *immersion* step consists of embedding the image in a set-valued map on a Khalimsky grid. The set-valued map aims at modeling the level lines passing between the pixels of the image by using intermediate pixels (0 and 1-faces). This step is illustrated in Fig. 3. Original pixels are represented by blue squares, the 1-faces are represented by rectangles, and the 0-faces by circles. 1-faces and 0-faces



(a) Example image and its level lines

(b) Interval-valued map after immersion

Fig. 3: Illustration of the immersion and level set distances. The top-left and top-right "one" pixels are not connected in (a) but they belong the same level set. In the interval-valued map (b), the level line "One" can be materialized and connects "physically" the two pixels "One": the distance between these two pixels is thus null.



(a) (b) (c)

Fig. 4: Order map (a) from algorithm 1 (a) and distance transform images (b,c) from algorithm 2. (b) is the result after one round while (c) is the result obtained after the algorithm has converged (150 rounds).

have interval to represent all possible level lines passing between the adjacent original pixels. For instance, considering the saddle point $\begin{smallmatrix}3 & 1\\ 0 & 3\end{smallmatrix}$, the level lines from 0 to 3 could pass in the center (depending on the interpolation), but only the level lines from 1 to 3 could pass between the two top pixels. The interval-valued map is the foundation of the ToS algorithm as it enables to *physically* follow the level lines in the image.

Propagation. The order map is then computed using the propagation algorithm displayed in algorithm 1. This procedure traverses the set-valued map F previously computed, starting from a root point p_{∞} whose associated value is v_{∞}, and creates a mapping from the components in the image and their level lines to their order of appearance in the ToS in a top-down traversal. To this aim, it relies on a hierarchical queue Q with two operations: $\text{PUSH}(Q, v, p)$ enqueues

Algorithm 1 Propagation algorithm

```
function PROPAGATION(F, p∞, v∞)
    Q ← ∅
    Ord(p) ← −1 forall p
    λ_old ← v∞, d ← 0
    PUSH(Q, v∞, p∞)
    while Q not empty do
        (λ, p) ← POP(Q, λ_old)
        if λ_old ≠ λ then
            d ← d + 1
        Ord(p) ← d
        for n ∈ N(p), Ord(n) = −1 do
            (m, M) ← F(n)
            λ' ← clamp λ in [m, M]
            PUSH(Q, λ', n)
            Ord(n) ← 0
        λ_old ← λ
    return Ord
```

a point p in the queue at level v and $\text{POP}(Q, v)$ returns the point p at the nearest level λ of v. Thus, all the image pixels are traversed, and the components are labelled. Each time a new level is encountered, meaning that the current component has been completed, the ordered counter d is increased by one. The order map ORD computed during the propagation on the *camera* image is shown in fig. 4a where values range from 0 to 5700.

Max-tree computation. Finally, a max-tree is built on the order image. A linear or quasi-linear algorithm can be used depending on the maximum value of the order-map. Once the tree obtained, some post-processing are usually applied: "unimmersing" to go back to the original domain, and attaching the nodes to their original grayscale value.

To make this process massively parallel, the only challenge is to parallelize the propagation algorithm. Indeed, the immersion algorithm is trivially parallelizable; this is a min/max interpolation, i.e. a 2x2 stencil pattern operation. The max-tree computation has been shown to be massively parallelizable [13]. It remains the propagation algorithm, and this is going to be discussed in the next section.

3. A DISTANCE TRANSFORM AS AN ORDER MAP

The order map computed during the propagation step of the ToS is not unique. Any image levelling (that preserves the ordering relation between any pair of neighboring pixels) can be used as an order map. Here we propose to use the distance transform from the border on an interval-valued image. Instead of counting the number of level-lines (as done in algorithm 1), we instead sum the differences between the level-lines on the path. This computes the gray-weighted distance transform [16] on the interval-valued image F. More formally, let $\Pi(x)$ be the set of paths from the border to the

Algorithm 2 A pass of the Distance Transform algorithm

```
1:  function ITERATION_M(in F, in out 𝓕, in out D)
2:      changed ← false
3:      for each pixel p do                    ▷ forward or backward
4:          (m, M) ← F(p)
5:          for each neighbor n in mask M do
6:              λ ← clamp 𝓕(n) in [m, M]
7:              d_new ← |𝓕(n) − λ| + D(n)
8:              if d_new < D(p) then
9:                  D(p) ← d_new; 𝓕(p) ← λ
10:                 changed ← true
11:     return changed
```

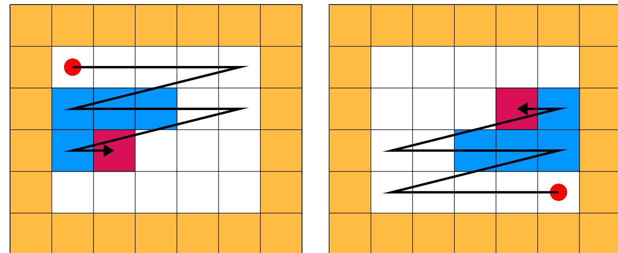


Fig. 5: Iterative Distance Transform masks used in forward (left)and backward (right) scans.

pixel x in the image, the distance of x is defined as:

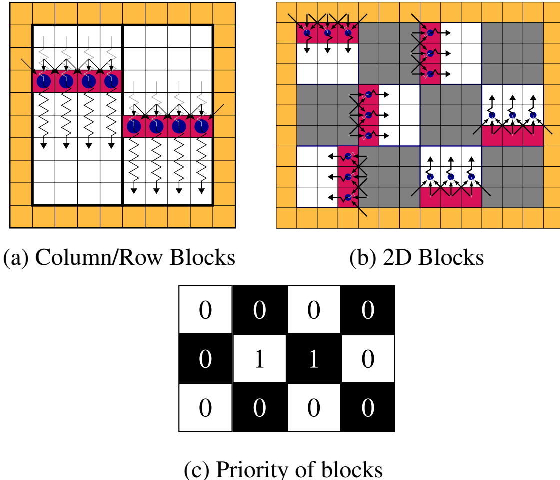
$$D(x) = \min_{\mathcal{F}, \pi \in \Pi(x)} \sum_{i=1}^{n-1} |\mathcal{F}(\pi_i) - \mathcal{F}(\pi_{i+1})|$$

where π_i is the i-th pixel of the path π and $\mathcal{F}$ is a projection of the interval-valued image F.

This distance could be implemented using the Fast Marching Method [14] or using an iterative algorithm [17]. The latter is more suited for parallelization. The basic idea is to iterate algorithm 2 using two scans (forward and backward) until no pixel is updated (see fig. 5). The forward and backward use two 3x3 kernels $\mathcal{M}$ to update the distance of a pixel based on its neighbors. $D(p)$ holds the current distance to the border, and $\mathcal{F}(p)$ holds the value chosen in the interval $F(p)$ to reach this distance. Compared to the regular geodesic distance transform, we only change the update rule. When observing the neighbor, we project (clamp) its value on the interval $F(p)$ and compute their difference (this is the distance between the two level lines). This difference is then added to the current distance to the border. If the new distance is smaller than the old one, we update the distance $D(p)$ and the chosen level line value $\mathcal{F}(p)$.

4. PARALLEL DISTANCE TRANSFORM

Parallelizing the distance transform(s) is not a novel idea... In 1968, [18] proposed a parallel version of distance transforms,



(a) Column/Row Blocks (b) 2D Blocks

(c) Priority of blocks

Fig. 6: Parallel strategies for the iterative distance transform. (a) Two blocks of 4 threads during the top-down pass. Each thread processes a column. Threads of the same block are synchronized after each line update. (b) Image split in 3x3 blocks colored with a chessboard pattern. This round only updates the white blocks. Blocks are independent, a block of threads updates a block of the image by repeating the four passes until convergence. (c) The priority assigned to each block of (b) based on the distance to the border. Priorities are used to schedule blocks in the task-based implementation.

the idea is to iteratively update the distance of each pixel based on its neighbors until convergence, in **parallel**. To avoid data races, the algorithm runs with a double buffer strategy and needs to synchronize after each round. Let n be the number of pixels in the image, and P the number of processors. This approach has a maximal parallelism, since with $P = n$ number of processors, all pixels can be updated in parallel at each round. In practice, the number of processors is much less than n, and very few pixels need to be updated at each round. In other words, very few processors are making progress at each round.

More recently, [19, 20] have proposed a more efficient parallelization strategy that augments the work by processor (or work per thread). Instead of updating a single pixel at each round, each processor is responsible for processing a whole column or a whole row of the image. During a pass, each processor updates the distance based on the distances of its neighbors from the previous row/column as illustrated in fig. 6 (a). While the algorithm from [17] requires two passes at each round, this new approach requires now fours passes: top-down, left-right, bottom-up, right-left. This strategy has been shown to be more efficient than the previous one, even if the level of parallelism is lower.

We adopted this strategy to parallelize our iterative distance transform algorithm as a baseline. We now explain how it can be further optimized.

Block-based parallelization *(Block)*. Instead of processing the image row by row or column by column, we propose to compute the distance transform block by block taking advantage of the cache hierarchy (fast CUDA shared-memory). We associate *status* to each block: *active* or *inactive*. An active block is a block that is adjacent to a block that has been updated during the previous round. To avoid data-race between values that are read/written by adjacent blocks, we use an odd-even communication strategy. Blocks have colors using a chessboard pattern. At each round, we update all *white active* blocks, then all *black active* blocks, and so on... as illustrated in fig. 6b The algorithm is then as follows:

procedure BLOCKDISTANCETRANSFORM
    color ← white
    Initialize blocks status to *inactive* except *white* border blocks
    while Not converged **do**
        for each active *color* block, **in parallel do**
            Iterate alg. 2 until convergence
            if *N/E/W/S* border has been updated **then**
                Set the *N/E/W/S* neighboring block to *active*
            Set the block to *inactive*
        Swap current *color*

Queue-based parallelization *(Queue)*. Using the classical CUDA programming model, the block-based approach has two drawbacks. First, at each round, each block is scheduled even if inactive (it runs but exits immediately). Second, blocks are scheduled in any order and if the image is complex, there might be a lot of active blocks at each round. As we compute a distance transform from the border, we want to prioritize the blocks close to the border as they are the most likely to get their final state. On the contrary, the most "inner" blocks are likely to be invalidated more often. We thus propose to switch to CUDA "task-based" parallelization with priority queue to schedule the blocks. The priority is *based* on the distance of the block to the border as shown on fig. 6c. Each "CUDA block" (*worker*) retrieves a *physical* block from the queue, processes it, and pushes its neighbors to the queue as previously. The number of workers is set according to the number of Streaming Multiprocessors (SM) of the GPU, and each worker will process a maximum predefined number of blocks (#border blocks / #workers). Thus, it *may not* process all *active* blocks at each round, only the most important ones that are likely to converge and not be invalidated anymore. Note that the priority is not "just" the distance of the block from the border. The value is mapped to [0-63] in order to get one queue per priority that leverages fast concurrent lock-free queue updates. Compared to the block-based approach, the queue-based approach requires more advanced lock-free programming techniques.

5. PERFORMANCE COMPARISON

We have conducted a performance comparison of the proposed distance transform algorithms and also compared

Method	MPixels/s	#Rounds	Speed-up
CPU	12.3 ± 1	1	1
GPU - Naïve	1.1 ± 0.5	1130 ± 480	0.1
GPU - Block	25.7 ± 8.3	1835 ± 802	1.9
GPU - Queue	85.9 ± 19.8	2196 ± 843	6.7

Table 1: Performance comparison of the distance transform with different parallelization strategies.

against the CPU propagation algorithm.

The bench dataset contains 639 grayscale aerial pictures (with sizes ranging from 2430×3500 to 3289×3500) from the database of the Netherlands Institute of Military History [1]. The CPU is an Intel Xeon Silver 4200 @ 2.20GHz and the GPU is an NVidia Quadro RTX 8000, the timings do not include memory transfers between host and device memories.

The results are reported in table 1. It shows that the naïve GPU distance transform is much slower than the CPU propagation. This was expected because it requires many rounds to converge (with 4 passes over the entire image at each). The block-based parallelization is much faster than the naïve one, due to data locality and cache reuse (processing the blocks in shared memory). The queue-based is the fastest, even if it requires more rounds to converge, each round is much faster because it only processes the most important blocks. The speed-up reaches 84x compared to the naïve GPU implementation and 6.7 compared to the CPU propagation.

However, this benchmark is **unfair** to the CPU algorithms for two reasons. First, we did not use the parallel version from [12]. In their work, the authors did not benchmark the contribution of the parallelization of each step (they consider the speed-up of the whole pipeline). In our tests, the parallelization of algorithm 1 with many threads did not scale well because the work in the queues is unbalanced. The speed-up, at the level of the propagation, was not significant. The performance could probably be improved by a better workload balancing, but we did not investigate this. Second, the GPU algorithms and parallelization strategies proposed in this paper can be implemented on the CPU using a thread-level parallelism **and** a data-level parallelism, i.e. using suitable SIMD instructions. However, the implementation is much less straightforward, and the compiler auto-vectorization has failed in issuing the SIMD code required for such implementation. Nevertheless, even if the CPU version could be improved, the speed-up would be at best the number of cores of the CPU, so our GPU implementation would still be challenging to beat.

Table 2 shows the distribution of the computation time of the ToS algorithm on CPU (sequential) and GPU with the Queue-based distance transform. We do not want to claim that the GPU version outperforms the CPU version, even if it does in this experiment (it should be benchmarked against an optimized parallel version). Our GPU distance transform is a direct replacement of the propagation algorithm, but does not gain as much as the immersion/max-tree steps. It is now by far the bottleneck of the ToS algorithm. However, even if we do not get a 10x speed-up on this step, the overall speed-up is still significant and the ToS can now be fully computed on the GPU without any memory movement to the Host. The last point was our main motivation in this work as the memory transfer that goes forth and back between the host and the device in image processing pipelines is often the bottleneck.

[1] `https://beeldbank.nimh.nl/`

Method	ToS on CPU	ToS on GPU
Immersion	82 (2%)	0.22 (<1%)
Propagation/DT	2921 (56%)	440 (97%)
Max-tree	2186 (42%)	9.32 (2%)
Total	5189	450

Table 2: Distribution of the computation time (in ms) of the sequential ToS algorithm on CPU and the massively parallel ToS on GPU.

6. CONCLUSION

We proposed the first algorithm to compute the tree of shapes **end-to-end** on massively parallel architectures. It enables using advanced morphological filters on GPU pipeline without any memory transfer to the host.

To this aim, we rely on an iterative algorithm to compute the order map of the shapes of an image instead of the propagation one. Furthermore, we exposed new parallelization strategies of this algorithm that significantly speeds up the computation by a factor of 84 compared to the classical ones. We have also shown that our approach outperforms the state-of-the-art CPU algorithms for the tree of shapes computation. Even if the CPU algorithms could probably be further optimized and parallelized, our GPU implementation would still be challenging to beat.

While this algorithm was designed to compute a distance transform specific to the tree of shapes, it can actually be used for any distance transform computation. In particular, the parallelization strategies proposed in this paper can be used to compute the geodesic distance transform, the chamfer distance transform, or any other distance transform that can be computed iteratively. As a perspective, we plan to adapt and benchmark our approach against the state-of-the-art GPU geodesic distance transforms from the FastGeodis framework [20].

7. REFERENCES

[1] Petra Bosilj, Ewa Kijak, and Sébastien Lefèvre, "Partition and Inclusion Hierarchies of Images: A Comprehensive Survey," *Journal of Imaging*, vol. 4, no. 2, pp. 33, Feb. 2018.

[2] P. Monasse and F. Guichard, "Fast computation of a contrast-invariant image representation," *IEEE Transactions on Image Processing*, vol. 9, no. 5, pp. 860–872, May 2000.

[3] Yongchao Xu, Thierry Géraud, and Laurent Najman, "Hierarchical image simplification and segmentation based on mumford-shah-salient level line selection," *Pattern Recognition Letters*, vol. 83, pp. 278–286, Nov. 2016.

[4] Minh Ôn Vũ Ngọc, Nicolas Boutry, Jonathan Fabrizio, and Thierry Géraud, "A minimum barrier distance for multivariate images with applications," *Computer Vision and Image Understanding*, vol. 197-198, pp. 102993, Aug. 2020.

[5] Petra Bosilj, Ewa Kijak, and Sébastien Lefèvre, "Beyond mser: Maximally stable regions using tree of shapes," in *Procedings of the British Machine Vision Conference 2015*. 2015, BMVC 2015, pp. 169.1–169.13, British Machine Vision Association.

[6] Gabriele Cavallaro, Mauro Dalla Mura, Jon Atli Benediktsson, and Antonio Plaza, "Remote sensing image classification using attribute filters defined over the tree of shapes," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 54, no. 7, pp. 3899–3911, July 2016.

[7] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, Apr. 1998.

[8] Vicent Caselles and Pascal Monasse, *Geometric Description of Images as Topographic Maps*, Springer Berlin Heidelberg, 2010.

[9] Yuqing Song, "A Topdown Algorithm for Computation of Level Line Trees," *IEEE Transactions on Image Processing*, vol. 16, no. 8, pp. 2107–2116, Aug. 2007.

[10] Thierry Géraud, Edwin Carlinet, Sébastien Crozet, and Laurent Najman, "A Quasi-linear Algorithm to Compute the Tree of Shapes of nD Images," in *Mathematical Morphology and Its Applications to Signal and Image Processing*. 2013, pp. 98–110, Springer Berlin Heidelberg.

[11] Edwin Carlinet, Sébastien Crozet, and Thierry Géraud, "The Tree of Shapes Turned into a Max-Tree: A Simple and Efficient Linear Algorithm," in *2018 25th IEEE International Conference on Image Processing (ICIP)*. Oct. 2018, pp. 1488–1492, IEEE.

[12] Sébastien Crozet and Thierry Géraud, "A first parallel algorithm to compute the morphological tree of shapes of nd images," in *2014 IEEE International Conference on Image Processing (ICIP)*, 2014, pp. 2933–2937.

[13] Nicolas Blin, Edwin Carlinet, Florian Lemaitre, Lionel Lacassagne, and Thierry Geraud, "Max-Tree Computation on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3520–3531, Dec. 2022.

[14] J A Sethian, "A fast marching level set method for monotonically advancing fronts.," *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, Feb. 1996.

[15] Won-Ki Jeong and Ross T. Whitaker, "A Fast Iterative Method for Eikonal Equations," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2512–2534, Jan. 2008.

[16] Ron Kimmel, Nahum Kiryati, and Alfred M Bruckstein, "Sub-pixel distance maps and weighted distance transforms," *Journal of Mathematical Imaging and Vision*, vol. 6, pp. 223–233, 1996.

[17] Pekka J Toivanen, "New geodosic distance transforms for gray-scale images," *Pattern Recognition Letters*, vol. 17, no. 5, pp. 437–450, 1996.

[18] Azriel Rosenfeld and John L. Pfaltz, "Distance functions on digital pictures," *Pattern Recognition*, vol. 1, no. 1, pp. 33–61, 1968.

[19] Antonio Criminisi, Toby Sharp, and Khan Siddiqui, "Interactive geodesic segmentation of n-dimensional medical images on the graphics processor," *Radiological Society of North America (RSNA)*, 2009.

[20] Muhammad Asad, Reuben Dorent, and Tom Vercauteren, "Fastgeodis: Fast generalised geodesic distance transform," *The Journal of Open Source Software*, vol. 7, no. 79, pp. 4532, 2022.