An alpha-tree algorithm for massively parallel architectures

Edwin Carlinet, Quentin Kaci [*] and Nicolas Blin [†]
EPITA Research Laboratory (LRE), Le Kremlin-Bicêtre, France
edwin1.carlinet@epita.fr

Abstract—**The alpha-tree, also known as the quasi-flat zone hierarchy is a widely used representation of images in Mathematical Morphology. This structure organizes the regions according to a similarity criterion into a tree, that eases the multiscale analysis of images. Many alpha-tree algorithms exist and computing this structure efficiently is still an active field of research. Indeed, the alpha-tree is commonly used in remote sensing where there is an urge for fast processing of large terabytes images. In this paper, we propose the first massively parallel alpha-tree algorithm that leverages concurrent union-find data structures to exploit the SIMT (Single Instruction Multiple Threads) programming model of GPUs. Our algorithm outperforms the State-of-the-Art parallel CPU algorithms by a factor of 10 on average on desktop computers and servers. It also opens new perspectives for using Mathematical Morphology methods on GPU pipelines.**

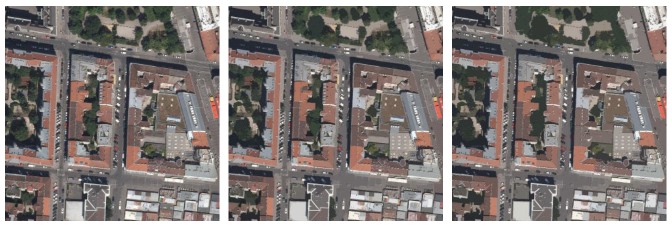


Fig. 1: Fine to coarse image simplification with the α-tree constrained with a maximal intra-cluster distance ω (a.k.a α-ω filtering [6]). Left: original image (859411 regions). Middle: $\alpha = 100, \omega = 60$ (100386 regions). Right: $\alpha = 100, \omega = 100$ (72902 regions).

I. Introduction

Mathematical morphology provides several hierarchical structures to represent images. They can be divided into two groups: the component trees and the hierarchies of partitions. In the first class, min and max-trees [1, 2] relies on the connected components of the threshold sets. Such components are either nested or disjoint and so, can be organized into an inclusion tree. The second class of representations is based on a sequence of nested partitions (any two partitions are related by the refinement relation). This set of partitions consequently forms a hierarchical structure. This closely relates to the problem of hierarchical clustering in machine learning which requires the combination of spatial relation and pixel similarity during region merging. Binary Partition Trees [3], hierarchies of watersheds [4, 5], and the quasi-flat zones hierarchies provide mathematical morphology-based solutions to this problem.

The quasi-flat zone hierarchy [6, 7], also known as the alpha-tree, has become increasingly popular for image processing over the last few decades for two main reasons.

Firstly, the impressive results showcased in the Berkeley segmentation challenge[8] have contributed to the growing trendiness of the hierarchy of partitions. Cousty et al. [9] has since proven the existence of a bijection between any hierarchy of segmentation and the alpha-tree of a (ultrametric) contour map. Thanks to this link, the alpha-tree is not only a representation of the image but also becomes a tool to construct and manipulate other hierarchies of segmentations at the pixel level. In simpler terms, it enables the transition between the tree representation and the image representation (saliency map) of the hierarchy, allowing transformations to be performed in the most adapted workspace [10, 9]. Figure 2 illustrates these two equivalent representations.

Secondly, the alpha-tree's capability to handle multi-channel images provides a distinct advantage over other representations. For instance, contrary to component trees, the alpha-tree does not depend on an arbitrary *total* ordering relation among image pixels for which there is no consensus. The alpha-tree utilizes a pixel similarity metric to define the tree's α-levels. Distance between vectors (e.g. the L2 distance in CIELab for colors) are generally more accepted than color orderings. Consequently, the alpha-tree is better suited for encoding RGB, multi-spectral, and hyper-spectral images [11], making it ideal for natural image processing and multi-spectral imaging. An example of image simplification with the alpha-tree hierarchy is shown in figure 1 where the number of regions is divided by a factor of 10 while preserving the important structures. Since it is a hierarchy of segmentation, the alpha-tree was obviously used in the context of color image segmentation as in [12] for natural scene segmentation, in [13] for medical photographic images, or in [14] to extend the grayscale Maximally Stable Extremal Region (MSER) [15] to color images. While its usage for natural scene processing has been declining for the last years in favor of deep-learning approaches, it is still used for remote sensing [16, 17, 18] to process very large images with thousands of channels.

The high computational cost of processing large images keeps the development of these algorithms an active research area. Most of the proposed algorithms so far have utilized Kruskal's Minimum Spanning Tree (MST) algorithm [19, 20], adopting a bottom-up approach (from leaves to root). Recent works involved in optimizing its computation have adopted two strategies: (a) using parallel architectures to distribute the work among processors [21, 22], or (b) using a different

[*][†] Q. Kaci is now with Seoul Robotics, and N. Blin is now with NVIDIA

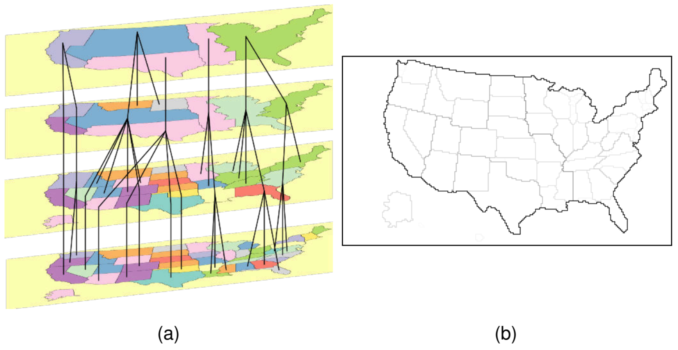


Fig. 2: A hierarchy of partition (left) and its alternative representation as a *saliency map* (right). Both representations are equivalent. The stronger the contour is, the longer the edge will remain in the hierarchy. Thresholding the saliency map gives one a partition of the hierarchy.

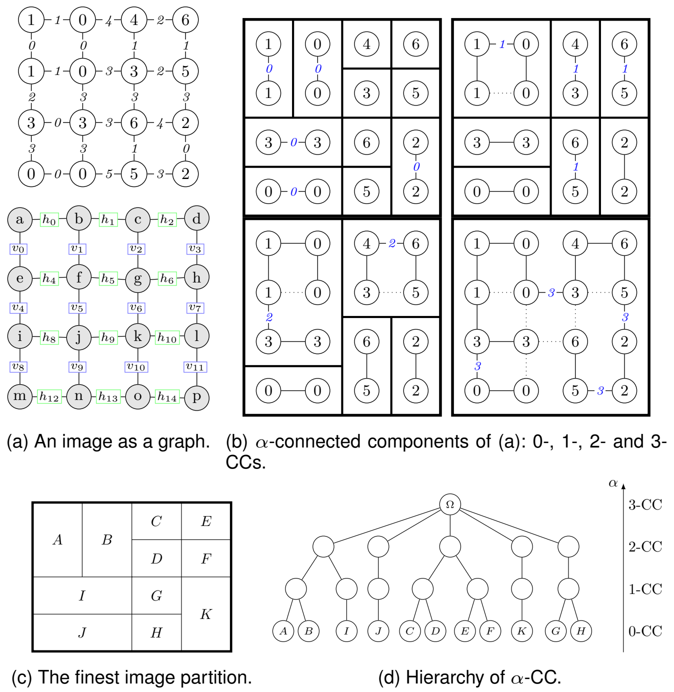

(a) An image as a graph.
(b) α-connected components of (a): 0-, 1-, 2- and 3-CCs.
(c) The finest image partition.
(d) Hierarchy of α-CC.

Fig. 3: The alpha-tree hierarchy illustrated on a sample graph [26]. The α-connected components of (a) form a sequence of partitions (b) which are nested and can be represented as an inclusion hierarchy (d).

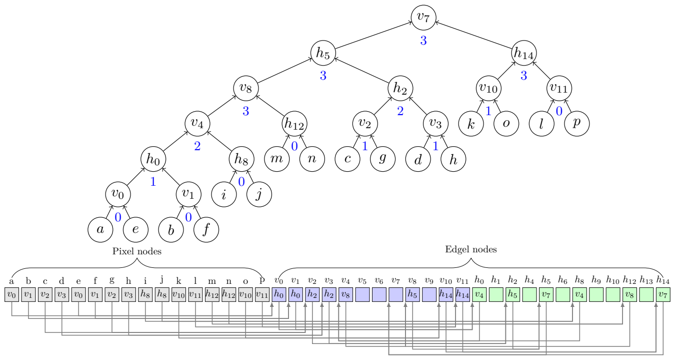


Fig. 4: The Binary Partition Tree representing the hierarchy depicted in figure 3. The pixels are leaves and the edges of the minimum spanning tree are internal nodes. The tree is encoded with an array of size $|V| + |E|$ where each element stores the index of its parent.

algorithmic strategy [23]. In [23], inspired by the max-tree algorithms, the authors propose a flooding strategy (top-down construction) with a better memory footprint. This idea was then extended in [24] for high-dynamic range images. However, recent works [25] have shown that a similar morphological representation, the max-tree, could be computed on GPUs. The contributions of this paper are:

- the introduction of **the first alpha-tree algorithm for massively parallel architectures that fits the SIMT paradigm of GPUs**, inspired by the previous works on the parallel computation of alpha-trees [25, 21, 22]
- a comparison of this algorithm with the State-of-the-Art approaches on many architectures.

The structure of this paper is as follows. In section II, we revisit the definition of the alpha-tree. Next, we review existing alpha-tree building algorithms in section III. In section IV, we detail our proposed alpha-tree algorithm for GPUs which is based on existing methods. In section VI, we benchmark our algorithms against state-of-the-art solutions, discussing their performance on an 8-connected grid and with high dynamic range (HDR) images. Finally, we conclude and provide perspectives in section VII.

II. Alpha-tree definition

A. Mathematical background

While the term *"alpha-tree"* is mostly used in the image processing terminology, this structure is easier to define and generalizable within the graph framework. An image can be represented as an undirected weighted graph $G = (V, E)$ where V is the set of vertices (pixels) and E is the set of edges defined by neighborhood, e.g., 4-neighbors or 8-neighbors in 2D. The weight of an edge $e = (v_i, v_j) \in E$ is defined with a symmetric dissimilarity $w_e = d(v_i, v_j)$ between corresponding pixels (generally a gradient, see figure 3a). Let $u, v \in V$, we can define a path $\Pi(u \to v)$ as a sequence of adjacent vertices that links u and v. We define the α-connected component containing a vertex $v \in V$, or α-CC(v) as a set of vertices (including v), for which there exists a path π from v with weights not greater than α. More formally, $\forall p, q \in \alpha\text{-CC}(v), \exists P = \Pi(p \to q), \forall e \in P, w_e \leq \alpha$. The α-connected components form an ordered sequence when α is growing, such that $\forall \alpha_i < \alpha_j$, $\alpha_i\text{-CC}(v) \subseteq \alpha_j\text{-CC}(v)$.

The alpha-tree is the tree of α-connected components. Therefore, it is the hierarchy where the parenthood relationship represents the inclusion of the α-connected components as illustrated on figure 3.

B. Tree data structure

A common way to represent hierarchies in Mathematical Morphology is to encode the parent relationship between elements in a *parent* image [27] as for component trees. However, the alpha-tree contains more nodes than component trees: the pixels form the leaves and the edge elements (edgels) form the internal nodes of the hierarchy. As we will see in section III, the alpha-tree is closely related to the minimum

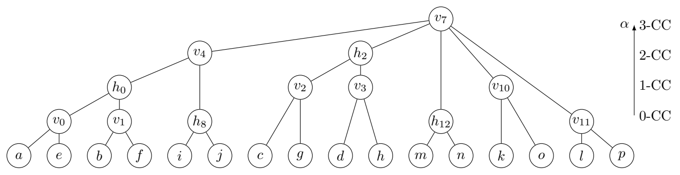


Fig. 5: The canonicalized version of the tree depicted in figure 4. Non-canonical nodes are removed yielding the final α-tree.

spanning tree (MST), so that the maximum number of nodes in the hierarchy is $2.|V| - 1$ (there are $|V| - 1$ edges in the MST). However, the edges that will be part of the MST are not known beforehand. To avoid memory allocation at runtime, our alpha-tree data structure allocates a node for every element (pixels and edgels). It is composed of three arrays in 4-connectivity: one for the pixels and the others for the edgels. Figure 4 shows the alpha-tree representation of the image figure 3a with the *parent* relation encoded in a linear array. The internal nodes are made of edges from a MST (dark edges in figure 3b). The first part of the array represents the pixel nodes (leaves) while the other part contains the nodes used when merging (internal edgel nodes). From a micro-architecture perspective, this organization eases memory management and maximizes the number of coalesced operations when threads have to access contiguous pixels and their edges. The downside remains that some cells will be allocated but unused.

The tree shown in figure 4 features nodes with parents at the same merging level, representing the *non-canonicalized* form of the desired tree (also called Binary Partition Tree by Altitude Ordering in [19]). A *canonicalization* step is required to obtain the final alpha-tree. The alpha-tree is a Binary Partition Tree with has all non-canonical nodes removed. This ensures that 1. $parent(x)$ is canonical for every node x, 2. $level(x) \neq level(parent(x))$ for any non-leaf node x as shown in figure 5.

III. State-of-the-Art

A. Sequential alpha-tree algorithms

1) Immersion-based algorithms: Most alpha-tree algorithms, such as those cited in [19, 20], utilize Tarjan's Union-Find process [28], which adopts a bottom-up approach. This method, which builds the tree from the leaves to the root, was the only one available for building an alpha-tree until recently, leading to the development of several variants.

The algorithm outlined in[19] is illustrated in algorithm 1. This process consists of two phases. First, Kruskal's algorithm[29] computes the minimum spanning tree using a union-find structure, then the tree is canonicalized to remove extra nodes and produce the alpha-tree. The algorithm begins by creating a 'singleton' tree for each pixel, forming a forest. Next, the algorithm processes edges in order of increasing weight. When an edge connects two disjoint trees, the algorithm creates a new node that becomes the new root of the merged trees. When all edges have been processed, there is only a single tree remaining in the forest. The following three functions are used to manage the forest with a union-find structure:

- MAKE-SET(parent, x) creates the singleton set $\{x\}$. It basically creates an empty tree where x is the root: $parent(x) \leftarrow x$
- FIND-ROOT(parent, x) climbs the tree that contains x and returns its root.
- UNION(parent, x, y) merges two partial trees rooted in x and y. It basically creates a new node and set the parents of x and y to the new node. The level of the new node will be the dissimilarity between nodes associated with x and y.

An efficient implementation of the MST in quasi-linear time requires two techniques. *Path-compression* should be performed in FIND-ROOT to shorten the chains to the root, and *union-by-rank* should be used in UNION to balance the tree and keep its height as small as possible when merging. The path-compression requires handling two structures for the trees. The first structure contains the full hierarchy while the other contains a flattened version of it. Therefore, the two techniques require extra-memory as we need to preserve *parent* (that holds the tree), *compressed parent* (that holds the flattened version of the tree), and an extra array to store the ranks. The memory-to-compute trade-off may not be worth it, and only the *path-compression* is usually implemented [20]. In [20], extra memory optimizations are proposed. They start with computing the flat zones (0-CC) to reduce the number of nodes pre-allocated for the leaves. They also propose to use directional hierarchical queues (one queue per edge direction) for sorting the edges and reducing memory usage. For example, a horizontal edge (p, q) would be noted $(p, HORIZ)$ and would be stored in the $HORIZ$ queue with only the value p.

Once the minimum spanning tree is computed, the $parent$ relation gives a binary partition tree. To obtain an alpha-tree, the algorithm requires a simplification step. As section II-B illustrates, the final alpha-tree is a hierarchy with all non-canonical nodes removed. The canonicalization algorithm (algorithm 1) processes the nodes from root to leaves by processing the sorted edges array backward and propagating the *canonical* property downward. When node x is processed, we know for sure that its parent q points to a canonical element, and can safely be updated.

Later, [22] proposed a different data structure to canonicalize the tree during the construction and minimize the number of node creation. A node is created only if it is needed, that is, only if the edge has a different level from the two roots to merge. Otherwise, it will simply attach one tree to the other without creating a new node. Moreover, edge insertions are interleaved with vertex insertions to reuse as many nodes as possible during the growth of the tree. This variant [22] has better performance than [19] and will be used as a baseline in section VI. However, it does not totally avoid spurious node creation and still needs a canonicalization step.

2) Flooding-based algorithms: Besides bottom-up approaches, top-down algorithms exist to construct other morphological trees like the max-tree [1, 30]. Recently, the same method was applied to the alpha-tree [23]. The alpha-tree flooding algorithm is similar to the min-tree one, with some optimization to handle more nodes, especially for low-dynamic-range images. The main difference lies in that the

Algorithm 1 Union-find-based alpha-tree algorithm.

```
1: procedure FIND-ROOT($parent$, $p$)
2:     while $p \neq parent(p)$ do
3:         $p \leftarrow parent(p)$
4:     return $p$

1: procedure CANONICALIZE($parent, level$)
2:     for edge $x$ in $E$ backward do
3:         $y \leftarrow parent(x)$
4:         if $level(parent(y)) = level(y)$ then
5:             $parent(x) \leftarrow parent(y)$
6:     for pixel $x$ do
7:         $y \leftarrow parent(x)$
8:         if $level(parent(y)) = level(y)$ then
9:             $parent(x) \leftarrow parent(y)$

1: function ALPHATREE($f$)
2:     for pixel $p$ do MAKE-SET($parent, p$)
3:     for edgel $e$ do MAKE-SET($parent, e$)
4:     $E \leftarrow \{$ all edges $\}$
5:     $Sort(E)$ increasing
6:     for $(p, q) = e$ in $E$ do
7:         $r_p \leftarrow$ FIND-ROOT($parent, p$)
8:         $r_q \leftarrow$ FIND-ROOT($parent, q$)
9:         if $r_p \neq r_q$ then UNION($parent, r_p, r_q$)
10:    CANONICALIZE($parent, level$)
11:    return $parent, level$
```

algorithm floods on edges and elements are thus enqueued at the edge weight. It means that a pixel can be enqueued several times at different levels since it can be reached by several edges, but a pixel is processed only once. Therefore, the algorithm has a linear-time complexity. Also, this algorithm is more memory efficient. Indeed, the hierarchy is built by creating level-root nodes only when there are required. No canonicalization step is needed as the flooding algorithm never creates non-canonical nodes, unlike the immersion-based approaches. Moreover, [23] presented a memory optimization called Tree Size Estimation that reduces up to 50% of the used memory. It has then been improved in [24] to handle high-dynamic-range images. The authors noticed that for those images most of edges are non-significant as they connect nodes that were already connected by edges at lower levels. They thus proposed to use a 3-levels hierarchical priority queue that uses the cache and defers the sorting of the strong edges until necessary. This algorithm will be also used in section VI as a comparison baseline.

B. *Parallel Alpha-tree algorithm*

A parallel approach for building an alpha-tree has been proposed in [21, 22]. The method tiles the image and computes the alpha-tree per block in parallel. Then, in order to obtain a global alpha-tree of the whole image, they need to process the remaining edges belonging to the tile's borders. [22] have proposed an algorithm that merges two branches of two

[1]EXCHANGE(p,q) := $old \leftarrow p$; $p \leftarrow q$; return old;

Algorithm 2 Partial alpha-tree merging

```
1: function FIND-LEVEL-ROOT($x, \alpha$)
2:     $q \leftarrow parent(x)$
3:     while $q \neq parent(q)$ and $level(q) \leq \alpha$ do
4:         $x \leftarrow$ EXCHANGE[1]($q, parent(q)$)
5:     return $x, q$

1: procedure CONNECT($a, b$)
2:     while $a \neq b$ do                          ▷ Zipping loop
3:         if $level(a) < level(b)$ then SWAP(a,b)
4:         $a, \_ \leftarrow$ FIND-LEVEL-ROOT($a, level(a)$)
5:         $b, \_ \leftarrow$ FIND-LEVEL-ROOT($b, level(a)$)
6:         if $a = b$ then return
7:         $b \leftarrow$ EXCHANGE($parent(b), a$)

1: procedure MERGE($a, b, \alpha$)
2:     $a, A \leftarrow$ FIND-LEVEL-ROOT($a, \alpha$)
3:     $b, B \leftarrow$ FIND-LEVEL-ROOT($b, \alpha$)
4:     if $a = b$ then return
5:     $parent(a) \leftarrow x$                ▷ Dettach $a$ and $b$, and
6:     $parent(b) \leftarrow x$           ▷ attach to new root node $x$
7:     $level(x) \leftarrow \alpha$
8:     $parent(x) \leftarrow A$ if $level(A) < level(B)$ else $B$
9:     CONNECT($A, B$)
```

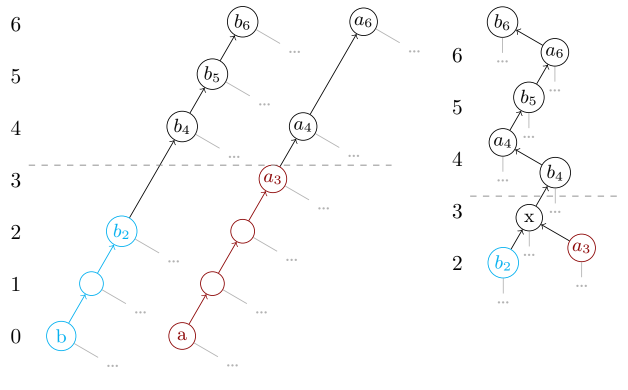


Fig. 6: Connecting two alpha-trees. An edge (a, b) with level $\lambda = 3$ is connecting two branches. (a) Branches are traversed up to the nodes b_2 and a_3 with level not greater than λ. (b) b_2 and a_3 are rooted to a new node x which points to the lowest old parent: b_4. The branches from b_4 and a_4 are merged up to their common ancestor, preserving the ordering of levels.

disjoints hierarchies. This algorithm is thus able to process the border edges that connect two vertices in two disjoint alpha-trees. Processing an edge (a, b) of weight λ requires a merge described in algorithm 2 that operates as follows:

- Climb the branches of a and b to find the roots of the $\lambda - CC$s of a and b. Those are the highest nodes in the tree that have their level not greater than the merge level λ (lines 2,3). The procedure FIND-LEVEL-ROOT follows the ordered linked list of parents up to a node whose level is greater than λ (see. figure 6a)
- Detach the two nodes from the tree and connect them to a new node that will represent the edge node (lines 5-7)
- Connect the new node to one of the lowest detached

parents (lines 8)

- Connect (zip) the two branches up to their common ancestor with the procedure CONNECT. This is barely as simple as merging two ordered linked lists into a single ordered linked list (see figure 6b).

The parallelization strategy proposed in [22] employs a classical concurrent divide-and-conquer pattern where "maps" and "reductions" can be performed in parallel with a reduction tree. With this pattern, two threads are always working on disjoint trees and a tree is never updated concurrently. It is thus not adapted to massively parallel architectures where threads have to update the tree simultaneously.

In algorithm 2, the function MERGE can have various implementations. A simple one would be to create a new node as the parent of the two nodes. However, it would create overlong branches in the tree and makes the merging inefficient. A solution proposed by [22] is to create a more complex structure in order to also update all children when merging two nodes. Despite this proposed solution, the merging can be sometimes less efficient than building an alpha-tree sequentially due to these long branches.

C. Links with the MST and the single-linkage hierarchical clustering (SHC)

The α-tree is closely to the single-linkage hierarchical clustering on the graph representation of the image. Also, as noted in [31], *the problem of single-linkage hierarchical clustering is functionally equivalent to constructing the minimum spanning tree (MST)* and then computing the dendrogram of the MST's edges. The parallel computation of the MST has been widely studied [32] and some massively parallel implementation for GPUs have been proposed. In [33], the MST construction is based on iterating rounds of the Borůvka's algorithm, in [34] the authors propose to split the graph in sub-graphs based on edges weights where the Borůvka's algorithm is used to compute the sub-graphs' MST and are merged with the Kruskal's algorithm. The edges of the MST are not enough to get the α-tree, we need to get a dendrogram from them. Hendrix et al. [31] build the dendrogram on sub-graphs and merge them in parallel using Kruskal's algorithm with a procedure similar to CONNECT (algorithm 2). In [32], they turn the MST into a dendrogram by partitioning the MST on *strong* edges and computing recursively the dendrograms on sub-trees with Kruskal. To the best of our knowledge, the parallel algorithms for the dendrogram construction all have a divide-and-conquer strategy which ends with a sequential Kruskal algorithm (to keep merge orders) [35, 36] and makes a GPU implementation difficult [37]. There are actually very few end-to-end (i.e., from the graph to the hierarchy) GPU implementation for the SHC computation (except [38] that uses a sequential agglomerative clustering in $|V|-1$ rounds).

While it exists links between the α-tree and the SHC, it should be noted that the algorithms are designed for different data. The previous algorithms are generally designed for dense graphs that represent distances between points while our proposed algorithm is designed for regular sparse graph (images) where there is at most three times more edges than nodes (in 2D). For dense graphs, it is necessary to build the MST to reduce the number of edges before computing the dendrogram while this is not compulsory for regular sparse graphs. As a consequence, the algorithm proposed in section IV is the first massively parallel algorithm for computing the SHC of a sparse graph and avoids an explicit MST construction step.

IV. UNION-FIND BASED ALPHA-TREE ALGORITHM ON GPU

As stated before, the alpha-tree can be built using immersion-based algorithms that make the tree grow from leaves to root with a union-find. The bottleneck of this approach lies in the need for sorting the edges by weights to obtain a sequential order of processing. Such a sequential order prevents parallelization.

To lift this restriction, Blin et al. [25] have proposed a max-tree algorithm, based on the union-find, that does not require any processing order and allows for massive parallelization. These algorithms build partial trees in several parts of the image and then use an algorithm similar to algorithm 2 to merge the partial trees while conserving the tree properties. The same idea holds for the alpha-tree.

A. Concurrent sort-less alpha-tree algorithm

The algorithm 3 builds an alpha-tree using the immersion-based union-find without a sorting step. It iterates over all edges $e = (p, q)$ of the image and then connects p and q to the edge node. The first call to CONNECT just inserts the edge node e in the branch of p, while the second call does the "real" job of merging two chains. At the end of the construction, the canonicalization of the tree is still required in order to remove useless nodes from the tree: the successive nodes that have the same level.

Algorithm 3 differs from algorithm 1 as it does not require to process the nodes in a particular order where algorithm 1 processes the hierarchy downward. Also, note that this procedure does not shrink the node array to decrease its memory usage by extracting only useful nodes. It only shortens the paths in the tree by modifying the parent relation. It is now obvious that if CONNECT can be called concurrently, algorithm 3 can be massively parallelized on the `for all` loops. That is the purpose of the algorithm 4 explained hereafter.

First, it is worth noticing that the algorithm introduces a new operator between elements (nodes and edges): $\prec$. $\prec$ is a total order that ensures that no cycle can be created if we concurrently update *parent* pointers. Indeed, during the tree construction, there is some non-determinism about how the $parent$ relation is set between elements that have the same level. With the parallel algorithm depicted in section III-B, this is not an issue since when considering a local tree, there are no concurrent updates. To prevent cycles, imposing a total order ensures that the parent relation between any two elements a and b can only be set in one direction (either $a \rightarrow b$ or $a \leftarrow b$ but not both). A possible total order $\prec$, based on levels and the scan order, can be defined as:

$$p \prec q \Leftrightarrow \text{level}(p) < \text{level}(q) \text{ or } (\text{level}(p) = \text{level}(q) \text{ and } p < q)$$

Algorithm 3 Sort-less alpha-tree algorithm.

```
1:  function ALPHATREE($f$)
2:      for all pixel $p$ do
3:          MAKE-SET($parent, p$)
4:          $level(p) \leftarrow 0$
5:      for all edge $e = (p, q)$ do
6:          MAKE-SET($parent, e$)
7:          $level(e) \leftarrow w_e$
8:          CONNECT($p, e$)                      ▷ Insert $e$ in the branch
9:          CONNECT($q, e$)                      ▷ The real merge
10:     return $parent, level$

1:  procedure CANONICALIZE($parent, level$)
2:      for all element $x$ do
3:          $y \leftarrow parent(x)$
4:          $parent(x) \leftarrow$ FIND-LEVEL-ROOT($y, level(y)$)
```

Algorithm 4 Concurrent lock-free CONNECT with a CAS

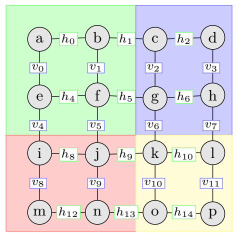
```
1:  procedure CONNECT($a, b$)
2:      while $true$ do
3:          if $level(b) < level(a)$ then SWAP($a, b$)
4:          $a, A \leftarrow$ FIND-LEVEL-ROOT($a, level(b)$)
5:          $b, B \leftarrow$ FIND-LEVEL-ROOT($b, level(b)$)
6:          if $a = b$ then                      ▷ Root reached
7:              return
8:          if $b \prec a$ then                  ▷ Prevent cycle
9:              SWAP($a, b$)
10:             SWAP($A, B$)
11:         $old \leftarrow$ ATOMICCAS($parent(a), A, b$)      ▷ Try
12:         if $old = a$ then          ▷ Early stop: Root reached
13:             return
14:         if $old = A$ then                   ▷ If $false \rightarrow$ retry
15:             $a \leftarrow old$
```

The concurrent procedure CONNECT in algorithm 4 has been proposed in [25]. It enables concurrent merge of nodes by "zipping" their branches up to their common ancestor. The ATOMICCAS avoids data races if the CONNECT procedure is called on the same part of the tree. This read-modify-write operation is used when updating the parent of a node. If two threads want to update the same node simultaneously, one will be granted the operation and continue its climbing while the other will have to retry the operation with the updated node. In the end, this procedure is nothing more than a concurrent merge algorithm of two sorted linked lists that preserves the ordering.

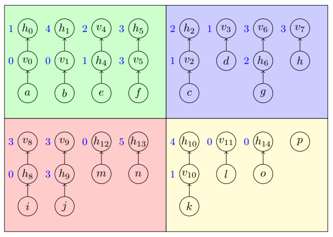
B. GPU Pipeline Implementation

The construction is a two-steps process depicted in figure 7. First the image is divided into blocks (figure 7a) and a partial alpha-tree is built per-block in *shared* memory with as many threads as the number of pixels in the block. Then, the partial alpha-trees are merged to get the final alpha-tree in *global* memory. The algorithm is detailed hereafter.

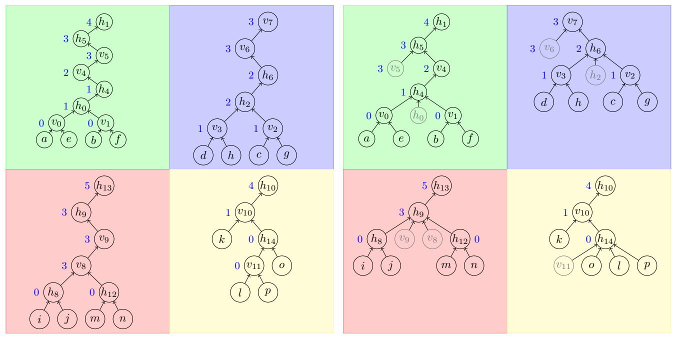
1) Thread-level alpha-chain: A thread of the block works is associated with a pixel in the image. Each of them then has



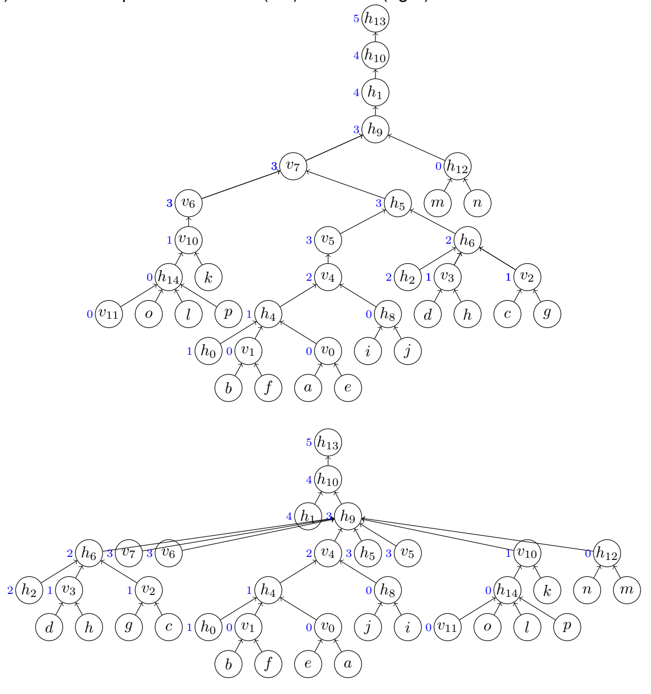
(a) Work split in blocks and threads.



(b) Thread-level alpha-trees.



(c) Block-level alpha-trees before (left) and after (right) block-level canonicalization.



(d) Global alpha-trees computed by horizontal and vertical block merges; before (up) and after (bottom) global canonicalization.

Fig. 7: Construction steps of the alpha-tree on figure 3a. (a) Tiling of the image into 4 blocks with 4 threads per block (b,c) local construction of a partial alpha-tree per block using shared memory. (c) Result of merging along vertical and horizontal tile borders in global memory and canonicalization.

to initialize and load the data from the global memory to the shared memory and computes the weights of the edges.

Then, each thread is responsible for building an alpha-tree for its pixel and its two (in 4-connectivity) or four (in 8-connectivity) connected edges. Each tree (*chain* actually) is built independently and concurrently by a thread. This simply implies to sort 2 or 4 edges by weight and set the parent relation between them. For instance, in figure 7b, the thread T_i is responsible for building the chain of the i-th pixel and edges h_i and v_i. Thread T_0 handles $\{a, h_0, v_0\}$ and sets $h_0 \leftarrow v_0 \leftarrow a$, Thread T_1 handles $\{b, h_1, v_1\}$ and sets $h_1 \leftarrow v_1 \leftarrow b$, and so on.

Then, the threads of the block synchronize to ensure that all the chains are built before the next step.

2) Block-level alpha-tree (figure 7c): Each thread has to merge the associated α-chain with its neighbors. This results in two calls to CONNECT per thread in 4-connectivity and four calls in 8-connectivity. All merges are achieved concurrently. For instance, in the first block, the thread T_0 merges its chain with the chain of T_1 and T_4 by calling CONNECT(h_0, b) and CONNECT(v_0, e). Let denote $a \leftrightsquigarrow b$ the call to CONNECT(a, b). Thus the following merges are performed:

- in Block 1: $h_0 \leftrightsquigarrow b$, $v_0 \leftrightsquigarrow e$, $v_1 \leftrightsquigarrow f$, $h_4 \leftrightsquigarrow f$
- in Block 2: $v_2 \leftrightsquigarrow g$, $h_2 \leftrightsquigarrow d$, $h_6 \leftrightsquigarrow h$, $v_3 \leftrightsquigarrow h$
- in Block 3: $h_8 \leftrightsquigarrow i$, $v_8 \leftrightsquigarrow m$, $v_9 \leftrightsquigarrow n$, $h_{12} \leftrightsquigarrow n$
- in Block 4: $h_{10} \leftrightsquigarrow l$, $h_{14} \leftrightsquigarrow p$, $v_{10} \leftrightsquigarrow o$, $v_{11} \leftrightsquigarrow p$

Once all merges are done, the block has a partial alpha-tree. It is worth noting that at this stage, *all* edges are part of the hierarchy. The next step is to canonicalize the tree in order to shrink non-canonical paths.

In order to reduce the memory footprint in shared memory, we use a 32-bits structure for nodes that contains the level of the node on 8-bits, the level of the parent on 8 bits, and the index of the parent on 16 bits. This structure is enough to handle the local computation and reduces the shared memory usage. Indeed, in CUDA, the maximum number of threads per block is 1024. Each block has 3 nodes per pixel in 4-connectivity, the maximum number of nodes is therefore: $3072 (\ll 2^{16} - 1)$. Reducing the shared memory usage has been of prime importance in order to reach the maximum GPU occupancy and increase the concurrency level.

3) Global alpha-tree merging (figure 7d): Once all the local alpha-trees have been constructed and committed into global memory, the remaining edges on the borders (horizontal and vertical) of the tiles are processed with CONNECT to merge the partial alpha-trees. Considering the example in figure 7d, we perform the following horizontal: $h_1 \leftrightsquigarrow c$, $h_5 \leftrightsquigarrow g$, $h_9 \leftrightsquigarrow k$, $h_{13} \leftrightsquigarrow o$, and the following vertical merges: $v_4 \leftrightsquigarrow i$, $v_5 \leftrightsquigarrow j$, $v_6 \leftrightsquigarrow k$, $v_7 \leftrightsquigarrow l$.

4) Global canonicalization (figure 7d): At the end of the construction, global canonicalization must be applied in order to remove useless nodes that have been created during merges of horizontal and vertical edges. The canonicalization kernel is called in one dimension with a thread associated with a node in the tree and starts from the end of the node array. Indeed, if we have $b \leftarrow a$ with a and b at the same level, then b is after a in the array (because $a \prec b$). Thus when "canonicalizing" a, we have higher chance to have b already canonicalized. This heuristic has been shown to be efficient in practice and reduces the number of links traversed by a thread.

Algorithm 5 Concurrent computation of the alpha-tree preventing redundant and residual edge insertion.

```
1:  procedure CONNECT-WITH($a, b, e$)
2:      $a, A \leftarrow$ FIND-LEVEL-ROOT($a, level(e)$)
3:      $b, B \leftarrow$ FIND-LEVEL-ROOT($b, level(e)$)
4:      if $a = b$ then
5:          return                                   ▷ "residual" edge
6:      elif $level(a) = level(e)$ or $level(b) = level(e)$ then
7:          CONNECT($a, b$)                          ▷ "redundant" edge
8:      else
9:          CONNECT($a, e$);   CONNECT($b, e$)
10: function ALPHATREE($f$)
11:     ...
12:     for all edge $e = (p, q)$ do
13:         CONNECT-WITH($p, q, e$)
14:     ...
```

C. Handling residual and redundant edges (GPU v2)

It is worth noting that the canonicalization does not remove all the "useless" nodes from the hierarchy. For instance, as shown in figure 7d, the node h_6 at level 2 is still present in the hierarchy while it does not connect any new "real" pixels that were already connected by v_3 and v_2. It has been shown in [24] that most edges are redundant (they connect previously connected pixels) or residual (they are above the root). This is especially true in 8-connectivity where the ratio number of edges over the number of pixels is higher.

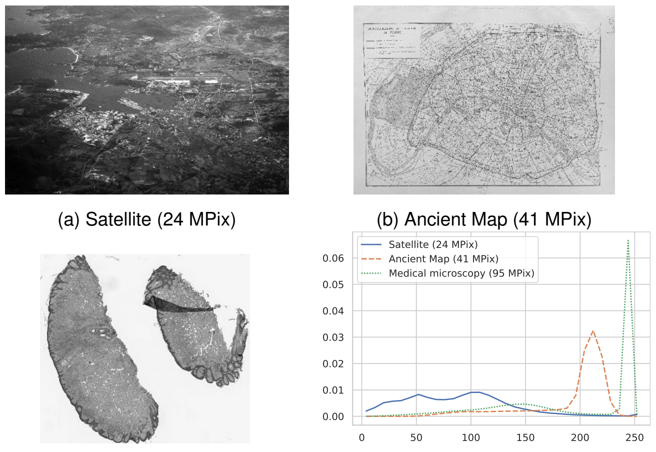
In algorithm 3, all edges are inserted in the hierarchy, even "useless" ones since we cannot know in advance if an edge is redundant or not. However, we can easily prevent the insertion of an edge if it is residual or redundant with an edge *already* in the hierarchy. In algorithm 5, when merging the paths of two pixels a and b connected by the edge e, we first check if a and b are already connected by a node x in the hierarchy such that $level(x) \leq level(e)$; and second, if there is already a node x in the paths of a or b at level $level(e)$. If so, the edge e is not inserted in the hierarchy; these paths are already merged, or we can use x to merge the paths. (Note that we do not build a thread-local alpha-chain anymore as it would result in all edges being inserted in the hierarchy.) This extra-check has no impact on the worst complexity of the algorithm since the calls to FIND-LEVEL-ROOT would have been performed in all cases.

To measure the impact of this optimization, we have run the algorithm on some random generated images and have measured the number of edges inserted in the hierarchy. We remind some interesting results from [24]. With the 4-connectivity, the number of *useful* edges is between 40 and 50% and with the 8-connectivity, it is between 20 and 25% (we have generally more useful edges in 16-bits). This is consistent with the fact that we need at most $|V| - 1$ edges to connect all pixels. Algorithm 5 has a *sequential consistency*, meaning that there exists a sequential order of processing edges that gives

Device	Model
CPU *"Desktop"*	8 × i7-2600 CPU @ 3.4GHz
CPU *"Server"*	40 × Xeon Silver 4210 cores @ 2.2GHz
GPU *"Desktop"*	GTX 1650 4GB - 896 Cores @ 1.5Ghz
GPU *"Desktop"*	GTX 1060 6GB - 1280 Cores @ 1.5Ghz
GPU *"Server"*	RTX 8000 - 4608 cores @ 1.4Ghz

TABLE I: Hardware devices in the benchmark setup.

the same result as the parallel algorithm. In the best case, the edges are treated in increasing order of levels, that gives exactly the same result as the immersion-based algorithms with a minimal number of edges in the hierarchy. In the worst case, the edges are treated in decreasing order of levels. On 1000 random edge sequences, the number of edges inserted in the hierarchy is 20%-25% lower on average with the 4-connectivity and 45-66% lower with the 8-connectivity than the one built with algorithm 3.



(a) Satellite (24 MPix) (b) Ancient Map (41 MPix)

(c) Medical microscopy (95 MPix) (d) Gray levels distribution

Fig. 8: Images used for benchmarking and their gray-level distribution.

V. Complexity analysis

For the complexity analysis of our algorithm, we consider two union-find optimizations that were not adopted in our implementation:

- Random indexes: instead of having a total order between pixels based on the scan-order, the order is based on a randomly generated index image
- Compaction: the union-find trees could be compressed (level-wise) during the FIND-LEVEL-ROOT procedure using *path-halving* or *path-splitting*.

While these optimizations do not change the correctness of our algorithm, it actually tends to slow down the implementation as more *atomic*-writes are performed, and the *randomness* causes more cache-missed memory access. Nevertheless, these procedures give better theoretical asymptotic bounds for the expected total work of the algorithm, and should be considered for those that needs strong worst-case guarantees.

We rely on the result from [39] giving that a problem instance in which there are n elements and m union-find operations can be solved in $\Theta(m.(\alpha(n, \frac{m}{np}) + \log(\frac{np}{m} + 1)))$ with the union-find algorithm (with the two optimizations above enabled) and p concurrent threads.

We remind that $|V|$ is the number pixels (vertices), the number of edges is $|E| = \theta(|V|)$ ($2.|V|$ with the 4-connected grid and $4.|V|$ with the 8-connected grid). Let g be the number of grayscale levels (the edge weights are limited to 256 values for 8-bits depth images).

The *find* operation is known to be $O(\log n)$ with high probability. It follows that the CONNECT(a,b) procedure, that calls FIND-LEVEL-ROOT (*i.e.*, the *find* operation) twice by level, is $O(g. \log |E|)$ with high probability. Our algorithm calls CONNECT for every edge in the image, it follows by substituting $n = |V|$, $m = g.|V|$ that our algorithm has a total work of $\Theta(g.|V|.(\alpha(|V|, \frac{g}{p}) + \log(\frac{p}{g} + 1)))$.

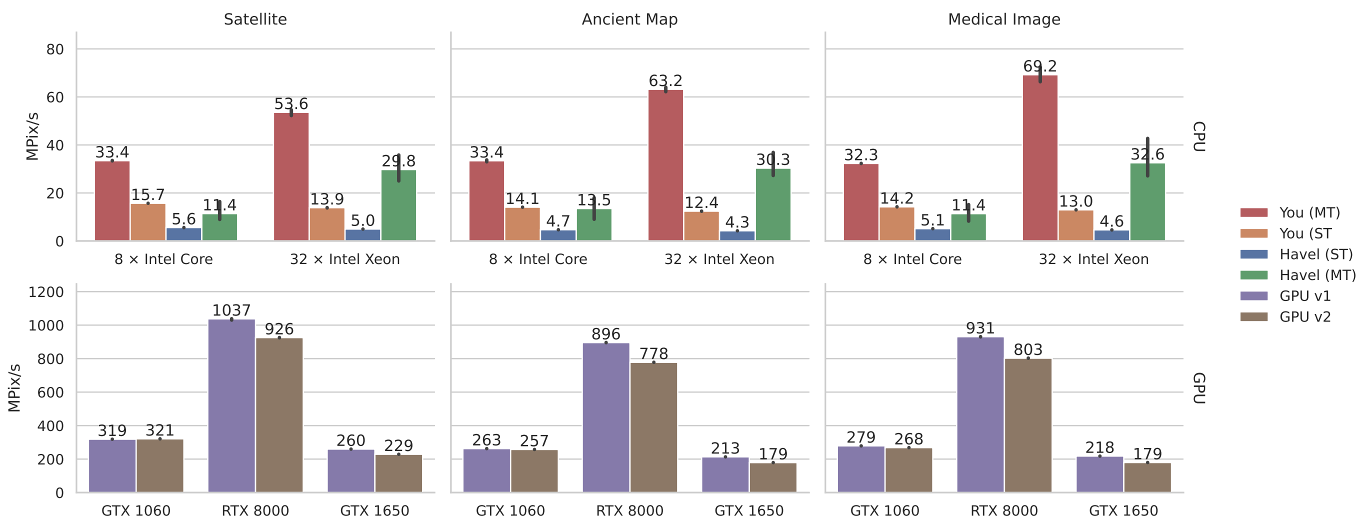
VI. Performance evaluation

We compare the performance between our GPU implementations and three State-of-the-Art CPU algorithms. Two of them are based on an immersion process that constructs the alpha-tree with a bottom-up approach (Union-Find). One of the best public implementations was from [22] that proposed a sequential and a parallel version. They will be designated respectively as *Havel (ST)* (single-thread) and *Havel (MT)* (multi-threads) in the benchmarks. The sequential and parallel flooding-based algorithm proposed by [23] have also a public implementation provided by [24]. These implementations will be denoted *You (ST)* and *You (MT)*.

Those CPU algorithms are compared to our GPU algorithms presented in section IV. The timings do not include the memory transfers between the host and the device. In a GPU pipeline, when processing images in batch, the latency due to the memory transfers between host and device can be hidden by the computational work (*latency hiding*). Therefore, the processing time will be much closer to the time without memory transfers. The same general configuration for all algorithms was used: grayscale one channel images, L1 norm dissimilarity function, and 4-connectivity. For the CPU parallel variants (*You (MT) and Havel (MT)*), the number of threads is actually a hyperparameter that is architecture and image size dependent. Therefore, it was set to t times the number of CPU cores available on the target architecture. Several t has been benchmarked to get the best trade-off between workload balancing and parallelization overhead, and we have retained the best t for each case.

Benchmarks were conducted on setups detailed in table I ("Desktop" and "Server") such that the compared devices belong to the same range. The benchmark was run on different kinds of images (shown in figure 8) that represent possible applications of the alpha-tree: remote sensing, document analysis, and medical imaging. We have also tested the performance of the algorithms on 1256 pictures including 639 grayscale aerial pictures (with sizes ranging from 2430×3500 to 3289×3500) from the database of the Netherlands Institute of Military History and 617 color images including maps, arts, and natural scenes. This is the dataset used in [23].

Figure 9 shows a comparison between all algorithms with the different images. Globally, the proposed GPU algorithm from 8 to 20 times faster than the best State-of-the-Art CPU algorithms on both *desktop* and *server* configurations.



(a) Performance comparison on the satellite, ancient map, and medical microscopy images.

	Desktop	Server
GPU v1	284.8 MP/s (± 5%)	919.3 MP/s (± 3%)
GPU v2	281.4 MP/s (± 6%)	825.6 MP/s (± 4%)
Havel (MT)	12.9 MP/s (± 7%)	14.4 MP/s (± 5%)
Havel (ST)	5.5 MP/s (± 7%)	5.0 MP/s (± 6%)
You (MT)	33.7 MP/s (± 2%)	40.1 MP/s (± 3%)
You (ST)	15.4 MP/s (± 6%)	13.8 MP/s (± 6%)

(b) Performance comparison on the grayscale image dataset from the Netherlands Institute of Military History.

Fig. 9: Performance comparison of several alpha-tree State-of-the-Art algorithms on several hardware configurations and image types.

Image	Local Trees	Global Merges	Canonicalization	Total
Map	96 ms (61%)	39 ms (25%)	23 ms (14%)	158 ms
Medical	206 ms (61%)	88 ms (26%)	47 ms (14%)	340 ms
Satellite	51 ms (68%)	12 ms (16%)	12 ms (16%)	75 ms

TABLE II: Processing time the GPU-v1 (in milliseconds) of each algorithmic steps on the GTX 1060.

Table II details the processing time taken by each CUDA kernels of the *GPU v1*. Note that similar distributions are observed on the other GPUs and with the *GPU v2* algorithm (distributions vary by $\pm 5\%$ at most by step). We can see that most of the time is spent on the computation of local trees. On the one hand, the computation of local alpha-trees being done in shared memory, it is better to spend the overall time there. On the other hand, it could also mean that too many operations are done. We can observe similar kernel profiles for the max-tree computation on GPUs [25]. However, they achieved to go under 50% of the time spent on the computation of the local trees by considering a 1D optimization to compute a max-tree.

A. *Performance on the 8-connected grid*

The extension of the algorithms to 8-connectivity is straightforward. When merging locally and then globally, diagonal connections are added. During the local tree construction, the computational work and the memory usage double. Indeed, we have twice as many edges thus we double the number of nodes. The number of merges to perform is also doubled. It explains why the performance is halved on all GPUs as

		GPU v1			GPU v2		
		4-C	8-C	↘	4-C	8-C	↘
GTX 1060	Map	**263 MP/s**	103 MP/s	-61%	257 MP/s	**131 MP/s**	**-49%**
	Medical	**279 MP/s**	111 MP/s	-60%	268 MP/s	**134 MP/s**	**-50%**
	Satellite	319 MP/s	147 MP/s	-54%	**321 MP/s**	**179 MP/s**	**-44%**
GTX 1650	Map	**213 MP/s**	**75 MP/s**	-65%	179 MP/s	73 MP/s	**-59%**
	Medical	**218 MP/s**	78 MP/s	-64%	180 MP/s	**79 MP/s**	**-56%**
	Satellite	**260 MP/s**	**128 MP/s**	-51%	228 MP/s	108 MP/s	-53%
RTX 8000	Map	**896 MP/s**	316 MP/s	-65%	778 MP/s	**319 MP/s**	**-59%**
	Medical	**930 MP/s**	339 MP/s	-64%	803 MP/s	**346 MP/s**	**-57%**
	Satellite	**1033 MP/s**	**494 MP/s**	-52%	925 MP/s	435 MP/s	-53%

TABLE III: Performance of the proposed algorithms with 8-connectivity (8-C) versus the 4-connectivity (4-C).

Image	Local Trees	Global Merges	Canonicalization	Total
Map	220 ms (55%)	138 ms (34%)	44 ms (11%)	401 ms
Medical	478 ms (56%)	294 ms (34%)	88 ms (10%)	859 ms
Satellite	101 ms (62%)	41 ms (25%)	22 ms (13%)	164 ms

TABLE IV: Processing time the GPU-v1 (in milliseconds) of each algorithmic steps on the GTX 1060 in 8-connectivity.

shown in table III. However, the performance drop is not the same between the algorithm versions. The *GPU v1* algorithm is more impacted by the 8-connectivity ($\tilde{6}0\%$ slower) than the *GPU v2* algorithm ($\tilde{5}0\%$ slower). This is due to the fact that the *GPU v2* algorithm tries to prevent the insertion of redundant and residual edges. This yields a smaller hierarchy and thus, less work to do when merging trees. It is confirmed by the time distribution shown in table IV. We can observe that the *global merges* part is more impacted and takes now up to 35%

		GPU v1			GPU v2		
		8-bits	16-bits	↘	8-bits	16-bits	↘
GTX 1060	Map	263 MP/s	31 MP/s	-88.3%	257 MP/s	42 MP/s	-83.9%
	Medical	279 MP/s	23 MP/s	-91.8%	268 MP/s	27 MP/s	-89.8%
	Satellite	319 MP/s	55 MP/s	-82.7%	321 MP/s	56 MP/s	-82.6%
GTX 1650	Map	213 MP/s	15 MP/s	-92.7%	179 MP/s	18 MP/s	-89.7%
	Medical	218 MP/s	12 MP/s	-94.7%	179 MP/s	12 MP/s	-93.1%
	Satellite	260 MP/s	30 MP/s	-88.3%	229 MP/s	27 MP/s	-88.0%
RTX 8000	Map	896 MP/s	160 MP/s	-82.1%	778 MP/s	145 MP/s	-81.4%
	Medical	931 MP/s	118 MP/s	-87.3%	803 MP/s	95 MP/s	-88.1%
	Satellite	1037 MP/s	224 MP/s	-78.4%	926 MP/s	171 MP/s	-81.5%

TABLE V: Performance of the proposed algorithms with 16-bits images versus 8-bits images.

of the total time (instead of 25% in 4-connectivity). It can take up to 60% of the total time on the RTX 8000 and becomes the bottleneck of the algorithm while it remains below 50% with the *GPU v2* algorithm.

B. Performance on HDR images

The current algorithm can be used as it is on high-quantized data. However, as described in [40], merging nodes is not efficient when quantization is high: performance depends on the length of the branch (that can drastically increase with the number of bits). To benchmark the performance penalty, RGB-24 images are transformed into 16-bit images following the protocol from [41]. As seen in table V, our algorithm on 16-bit images suffered a drastic slow-down (around 5 to 10 times slower). This slow-down is mostly caused by global merges that take 80% to 95% of the total compute time (for both *GPU v1* and *GPU v2* algorithms). Such results were expected: with 16-bit images the number of (canonical) nodes increases drastically and thus, the chains are longer. Accesses in global memory are even more sparse, data locality is reduced, and cache miss rate is increased. Longer chains also induce bad workload balancing among threads. While one thread may climb a very long chain, other threads inside the block are stalling. In [42], the authors suggest duplicating the tile boundaries (called *halo*), so that merges in global memory are done on two nodes with the same levels. This technique was later used in [25] to optimize the max-tree construction on HDR images. The past results have shown that on GPUs, the *halo* improved very slightly the performance. Therefore, we did not pursue this lead any further.

Totally removing "useless" edges from the local alpha-trees before the global merges to reduce the hierarchy size might improve the performance to some extent, but it requires extra-computation and extra-memory to detect them. This is a trade-off that has to be considered in future works.

C. Comparison with related GPU algorithms

As stated in section III, the α-tree is closely related to the single-linkage clustering whose typical implementation uses a two-steps process: (a) building the MST, (b) organizing the edges in a tree. The current State-of-the-Art GPU implementation from [33] has been adapted for the 4-connected grid and optimized with the algorithm from [43, 44] to track disjoint connected components with a concurrent union-find approach.

Image	*GPU v1*	MST (GPU)	Sequential dendrogram
Map	213 MPix/s	226 MPix/s	5.7 Mpix/s
Medical	218 MPix/s	231 MPix/s	6.9 MPix/s
Satellite	260 MPix/s	212 MPix/s	8.9 MPix/s

TABLE VI: Performance of the MST and Sequential dendrogram constructions on 8-bits images.

Our version implements the concurrent Borůvka's algorithm to compute the MST as follows. First, each pixel x is set to its own component ($L[x] = x$). Then, the following procedure is repeated until there remains a single connected component:

- each pixel x concurrently selects the weakest edge (x, y) in its neighborhood that links two disjoint components ($L[x] \neq L[y]$). It then updates $WE[L[x]]$ that stores the weakest outgoing edge of the component rooted in $L[x]$.
- each pixel x concurrently looks at the edge (u, v) in $WE[x]$ (if it exits) and calls the union-find operation $union(L, u, v)$ that merges the components of u and v. The edge (u, v) is added to the MST edge list.
- each pixel *path*-compresses the label L image so that $L[x]$ points to the *root* label, and in the same time, it increments the connected components counter if x is a root label.

This MST algorithm has been benchmarked with the GTX 1650 (see table VI) and already shows to be comparable or slower than our α-tree algorithm while it only computes the first step of the whole process. In the public implementation of [32] and [45], only the MST computation and edge sorting are parallelized. The dendrogram construction is done sequentially on the CPU. In our tests, the sequential dendrogram construction reaches from 6 to 9 Mpix/s which drastically slows down the whole process. As a consequence, the two-steps track to build the α-tree has not been investigated any further.

VII. Conclusion & perspectives

In this paper, the first GPU algorithms for alpha-tree computation have been proposed. These algorithms were benchmarked against the State-of-the-Art solutions and achieves a significant speed-up: they are about **one order of magnitude faster** than the best CPU algorithms. The performance of our algorithm makes possible real-time applications based on hierarchical segmentations. It could also speed up very large images processing by distributing the work on many GPUs and merging the trees with parallel strategies. As a matter of reproducible research, our code is publically available on GitLab at https://gitlab.lrde.epita.fr/qkaci/gpu_alpha_tree.

This paper has also highlighted some shortcomings of this algorithm. In particular, its performance drops with HDR images which are common in remote sensing. Also, we have not yet tested the performance on 3D images, but the experiments with the 8-connectivity suggest that managing the 26-connectivity with so many edges will be challenging, and we expect a drop of performance proportional to the number of edges. High-dimensionality and high-dynamics are thus the current limitations, and we hope to address them in future works. Hybrid strategies [41] and recent advances in the management of high-dynamic-range edges [24] could be used to improve the performance on these kinds of images.

Acknowledgments

The authors would like to thank Theo Lepage and Ilan Guenet from EPITA for interesting discussions and ideas about the parallel computation of the alpha-tree. Credits:

Aerial view of Olbia https://upload.wikimedia.org/wikipedia/commons/8/82/Aerial_view_of_Olbia.jpg

Ancient Map Atlases of Paris - Atlas municipal des vingt arrondissements de Paris. 1925. Bibliothèque de l'Hôtel de Ville. City of Paris. France. https://bibliotheques-specialisees.paris.fr/ark:/73873/pf0000935524.locale=fr

Brightfield Microscopy (APERIO CMU-1.SVS) https://openslide.cs.cmu.edu/

Havel, Array / Parallel (CPU) implementation https://github.com/jirihavel/libcct

You, Flooding (CPU) implementation https://github.com/jwRyu/AlphaTree

References

[1] 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, 1998.

[2] R. Jones, "Connected filtering and segmentation using component trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, 1999.

[3] P. Salembier and L. Garrido, "Binary partition tree as an efficient representation for filtering, segmentation and information retrieval," in *Proceedings 1998 International Conference on Image Processing. ICIP98 (Cat. No.98CB36269)*, vol. 2, 1998, pp. 252–256 vol.2.

[4] S. Beucher, "Watershed, hierarchical segmentation and waterfall algorithm," in *Mathematical Morphology and its Applications to Image Processing*. Springer, 1994, pp. 69–76.

[5] F. Meyer, "The dynamics of minima and contours," in *Mathematical Morphology and its Applications to Image and Signal Processing*. Springer, 1996, pp. 329–336.

[6] P. Soille, "Constrained connectivity for hierarchical image partitioning and simplification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 7, pp. 1132–1145, 2008.

[7] P. Soille and L. Najman, "On morphological hierarchical representations for image processing and spatial data clustering," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*. Springer, 2010, pp. 43–67.

[8] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik, "Contour detection and hierarchical image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 5, pp. 898–916, 2010.

[9] J. Cousty, L. Najman, Y. Kenmochi, and S. Guimarães, "Hierarchical segmentations with graphs: quasi-flat zones, minimum spanning trees, and saliency maps," *Journal of Mathematical Imaging and Vision*, vol. 60, no. 4, pp. 479–502, 2018.

[10] J. Cousty, L. Najman, and B. Perret, "Constructive links between some morphological hierarchies on edge-weighted graphs," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*. Springer, 2013, pp. 86–97.

[11] G. K. Ouzounis, "Segmentation strategies for the alpha-tree data structure," *Pattern Recognition Letters*, vol. 129, pp. 232–239, 2020.

[12] F. Merciol and S. Lefèvre, "Fast image and video segmentation based on alpha-tree multiscale representation," in *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*, 2012, pp. 336–342.

[13] W. Tabone, M. H. F. Wilkinson, A. E. J. V. Gaalen, J. Georgiadis, and G. Azzopardi, "Alpha-tree segmentation of human anatomical photographic imagery," in *Proceedings of the 2nd International Conference on Applications of Intelligent Systems*. New York, NY, USA: Association for Computing Machinery, 2019.

[14] P.-E. Forssén, "Maximally stable colour regions for recognition and matching," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2007, pp. 1–8.

[15] M. Donoser and H. Bischof, "Efficient maximally stable extremal region (mser) tracking," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1, 2006, pp. 553–560.

[16] D. Ehrlich, T. Kemper, X. Blaes, and P. Soille, "Extracting building stock information from optical satellite imagery for mapping earthquake exposure and its vulnerability," *Natural Hazards*, vol. 68, p. 79?95, 2013.

[17] G. Ouzounis, "Automatic extraction of built-up footprints from high resolution overhead imagery through manipulation of alpha-tree data structures," Mar. 2014, uS Patent 8,682,079.

[18] M.-T. Pham, E. Aptoula, and S. Lefèvre, "Classification of remote sensing images using attribute profiles and feature profiles from different trees: a comparative study," in *Proc. of the IEEE Intl. Geoscience and Remote Sensing Symposium (IGARSS)*, 2018, pp. 4511–4514.

[19] L. Najman, J. Cousty, and B. Perret, "Playing with kruskal: algorithms for morphological trees in edge-weighted graphs," in *Proc. of the Intl. Symp. on Mathematical Morphology (ISMM)*, 2013, pp. 135–146.

[20] G. K. Ouzounis and P. Soille, "The alpha-tree algorithm," *JRC Scientific and Policy Report*, 2012.

[21] J. Havel, F. Merciol, and S. Lefèvre, "Efficient schemes for computing α-tree representations," in *Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2013, pp. 111–122.

[22] J. Havel, F. Merciol, and S. Lefèvre, "Efficient tree construction for multiscale image representation and processing," *Journal of Real-Time Image Processing*, vol. 16, p. 1129–1146, 08 2019.

[23] J. You, S. C. Trager, and M. H. F. Wilkinson, "A fast, memory-efficient alpha-tree algorithm using flooding and tree size estimation," in *Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2019, pp. 256–267.

[24] J. Ryu, S. C. Trager, and M. H. F. Wilkinson, "A fast alpha-tree algorithm for extreme dynamic range pixel dissimilarities," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, pp. 3199–3212, 2024.

[25] N. Blin, E. Carlinet, F. Lemaitre, L. Lacassagne, and T. Géraud, "Max-tree computation on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3520–3531, 2022.

[26] E. Carlinet, "A tree of shapes for multivariate images," Ph.D. dissertation, Université Paris Est, Paris, France, Nov. 2015.

[27] C. Berger, T. Géraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective component tree computation with application to pattern recognition in astronomical imaging," in *IEEE International Conference on Image Processing (ICIP)*, vol. 4. IEEE, 2007, pp. IV–41.

[28] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, p. 215–225, apr 1975.

[29] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[30] M. H. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proc. of the Intl. Conference of Image Processing (ICIP)*, 2011, pp. 1021–1024.

[31] W. Hendrix, M. M. Ali Patwary, A. Agrawal, W.-k. Liao, and A. Choudhary, "Parallel hierarchical clustering on shared memory platforms," in *Proc. of the Intl. Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2012, pp. 1–9.

[32] Y. Wang, S. Yu, Y. Gu, and J. Shun, "Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1982–1995.

[33] V. Vineet, P. Harish, S. Patidar, and P. J. Narayan, "Fast minimum spanning tree for large graphs on the GPU," in *Proc. of the ACM Conference on High Performance Graphics (HPG)*, 2009, p. 167.

[34] S. Rostrup, S. Srivastava, and K. Singhal, "Fast and memory efficient minimum spanning tree on the gpu," in *Proceedings of the 2nd Intl. Workshop on GPUs and Scientific Applications (GPUScA, 2011)*, 2011, pp. 3–13.

[35] C. Jin, M. M. A. Patwary, A. Agrawal, W. Hendrix, W.-k. Liao, and A. Choudhary, "Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce," in *Proc. of the Intl. SC Workshop on Data Intensive Computing in the Clouds (DataCloud)*, vol. 23. Citeseer, 2013, p. 27.

[36] C. Jin, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary, "Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce," in *Proc. of the Symposium on High Performance Computing (HPC)*. Society for Computer Simulation International, 2015, p. 83–92.

[37] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," in *Modeling and Simulation for Defense Systems and Applications VIII*, vol. 8752. SPIE, 2013, pp. 97–104.

[38] D.-J. Chang, M. M. Kantardzic, and M. Ouyang, "Hierarchical clustering with cuda/gpu." in *Parallel and Distributed Computing and Communication Systems*. Citeseer, 2009, pp. 7–12.

[39] S. V. Jayanti and R. E. Tarjan, "A randomized concurrent algorithm for disjoint set union," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, 2016, pp. 75–82.

[40] E. Carlinet and T. Géraud, "A comparative review of component tree computation algorithms," *IEEE Transactions on Image Processing*, vol. 23, no. 9, pp. 3885–3895, Sep. 2014.

[41] U. Moschini, A. Meijster, and M. H. Wilkinson, "A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 3, pp. 513–526, 2017.

[42] M. Gotz, G. Cavallaro, T. Geraud, M. Book, and M. Riedel, "Parallel computation of component trees on distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2582–2598, Nov. 2018.

[43] Y. Komura, "GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the swendsen–wang multi-cluster spin flip algorithm," *Computer Physics Communications*, vol. 194, pp. 54–58, 2015.

[44] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, "Optimizing gpu-based connected components labeling algorithms," in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*. IEEE, 2018, pp. 175–180.

[45] P. Sao, A. Prokopenko, and D. Lebrun-Grandie, "Pandora: A parallel dendrogram construction algorithm for single linkage clustering on gpu," in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 908–918. [Online]. Available: https://doi.org/10.1145/3673038.3673148

Edwin Carlinet received the Ing. degree from EPITA (Paris, France) in 2011, the M.Sc degree in Mathematics, Vision, and Machine Learning from ENS Cachan in 2012 and the Ph.D. degree in computer science from University Paris-Est in 2015. He is now an associate professor at EPITA. His research interests include HPC, in particular, the optimization of algorithms dedicated to Mathematical Morphology and Image processing. He is the maintainer of PYLENE https://gitlab.lre.epita.fr/olena/pylene.

Nicolas Blin received the Ing. degree in image-processing at EPITA, Paris, France in 2022 and was also working as a research assistant at EPITA Research Laboratory (LRE). His research interests involve algorithms parallelization on GPU, mathematical morphology, and metaheuristic algorithms. He is now Senior DevTech Engineer at NVidia.

Quentin Kaci received the Ing. degree in image-processing at EPITA, Paris, France in 2022 and was also working as a research assistant at EPITA Research Laboratory (LRE). His research interests involve the optimization of Computer Vision & Image Processing algorithms. He is now Software Engineer at Seoul Robotics.