Max-tree Computation on GPUs

Nicolas Blin, Edwin Carlinet, Florian Lemaitre, Lionel Lacassagne, Thierry Géraud *Member, IEEE*

Abstract—In Mathematical Morphology, the max-tree is a region-based representation that encodes the inclusion relationship of the threshold sets of an image. This tree has been proven useful in numerous image processing applications. For the last decade, works have been led to improve the building time of this structure; mixing algorithmic optimizations, parallel and distributed computing. Nevertheless, there is still no algorithm that takes benefit from the computing power of the massively parallel architectures. In this work, we propose the first GPU algorithm to compute the max-tree. The proposed approach leads to significant speed-ups, and is up to one order of magnitude faster than the current State-of-the-Art parallel CPU algorithms. This work paves the way for a max-tree integration in image processing GPU pipelines and real-time image processing based on Mathematical Morphology. It is also a foundation for porting other image representations from Mathematical Morphology on GPUs.

Index Terms—Mathematical morphology, hierarchical image representation, component-trees, max-tree, graph algorithms.

1 Introduction

Originally from the mathematical morphology field, the component trees are powerful and versatile structure that organizes the level sets of an image as tree. The min- and max-trees were first introduced in [2], motivated by the gain in interest for connected operators. Connected filters are preserving the contours of the objects of an image by only merging its flat zones. These operators are known for quite a long time and date back from the 90s [3], [4], but they are still widely used in today's image processing pipelines for efficient pre- or post-processing steps (e.g., background removal for brain lesion detection [5], noise removal [6]).

Connected filters are directly linked to the min- and max-trees as these structures enable simple and efficient implementations of them [7]. These trees have soared for more advanced forms of filtering: based on attributes [8], [9], with non-trivial filtering rules [10], allowing new generation connectivities [11]. Some uses of the max-tree are illustrated on figure 1.

Beyond filtering and image processing, the component trees are used in computer-vision related tasks. For instance, pattern spectra and attribute profiles, that computes the distribution of sizes and shapes of image regions, have been used with success in classification of satellite and astronomical images [12], [13], [14] and content-based image retrieval [15], [16]. The Maximal Stable Extremal Regions [17] are well-known descriptor to find correspondences between images and fast linear algorithms are based on the max-tree [18]. Region-based analysis using morphological trees have also been used in medical imaging, e.g., for blood vessel segmentation [19] and 3D visualization [20].

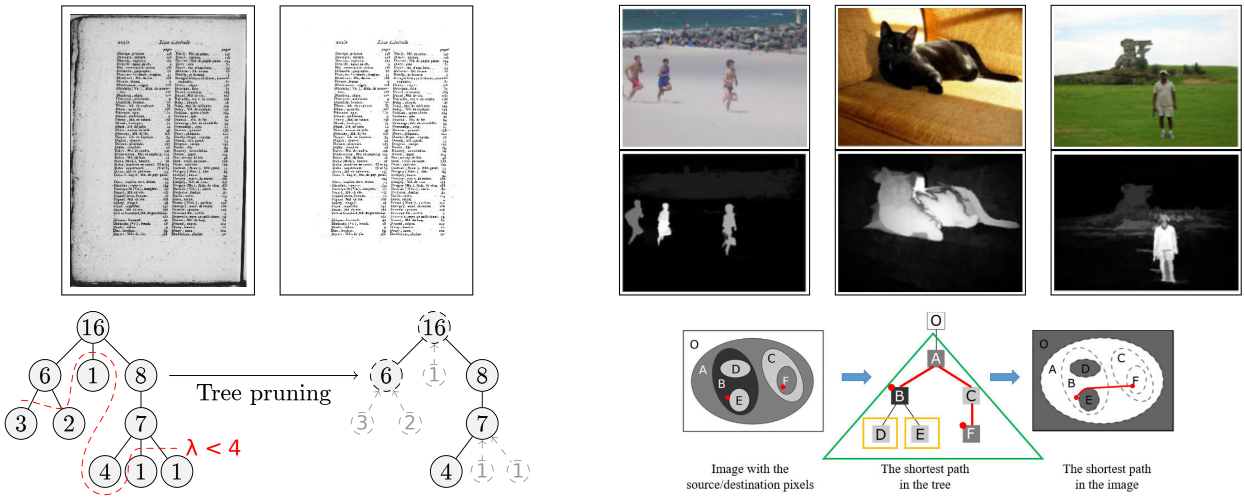
Min- and max-trees are also at the basis of other image representations. In [21], a self-dual hierarchical representation of the image, the tree of shapes, encodes the inclusion of the level lines. It is computed by merging the min- and max-trees. Later, [22] have shown that the max-tree algorithm can compute the tree of shapes (after a first image transformation).

When it comes to deploying image processing methods, fast algorithms are required. Might it be because of some real-time constraints, or because the amount of data to process is increasing steadily, there is a need for fast implementations. A typical pipeline using the max-tree has three steps: construction of the max-tree, attribute computation and filtering, and image restitution. However, more than 90% of the pipeline duration is spent in the construction of the tree [23]. Many algorithms have been developed for speeding-up the max-tree computation. So far, the proposed optimization techniques can *roughly* come under one of these three categories: (a) algorithmic optimizations, i.e., choosing between a top-down or a bottom-up construction with adapted data structures [18], [24], [25]; (b) thread level parallelism, i.e., classical parallelism for multiprocessors with shared memory (SMP) [26], [27], [28]; (c) distributed computing, i.e., joint max-tree computation between distributed memory [29], [30]. To the best of our knowledge, **this is the first time a max-tree algorithm is proposed for massively parallel architectures and fits the SIMT paradigm of GPUs**.

The paper is organized as follows. In section 2, we remind the definition of the max-tree and in section 3, we provide an overview of the sequential, parallel and distributed State-of-the-Art max-tree computation algorithms. We also make some links with the Connected Component Labeling algorithms, in particular those dedicated to GPUs as they will be the base of our proposal. In section 4, we depict our proposed max-tree algorithm with implementation details for hierarchical memory models and in section 5, we propose an optimized version taking advantages of the super-efficient max-tree algorithm for 1D-signals. In section 6, we compare the performance of our algorithms to the State-of-the-Art sequential and parallel ones on three ranges of architectures (desktop stations, mobile devices and servers).

- *Nicolas Blin, Edwin Carlinet, and Thierry Géraud are with EPITA Research and Development Laboratory (LRDE), Paris, France. (Contact:* firstname.lastname@*lrde.epita.fr)*
- *Florian Lemaitre and Lionel Lacassagne are with LIP6, ALSOC team, Sorbonne University, CNRS, Paris, France. (Contact:* firstname.lastname@*lip6.fr)*

Manuscript received April 19, 2005; revised August 26, 2015.



(a) Document image filtering. (b) Salient object detection.

Fig. 1: Image processing with the max-tree. (a) Background extraction for document image analysis with a morphological black top-hat. The min-tree is pruned by removing connected small peak components (< 4 pixels that do no touch the border. The residue (filtered components) forms the clean text and the background is removed. (b) Salient object detection for scene analysis [1]. A morphological tree is used to compute the Dahu distance from image borders. This distance that mixes geodesic distance and gray-level distances can be computed using a tree similar to the max-tree (the tree of shapes).

Section 7 is dedicated to the 8-connectivity case and how it can be managed without loss of performance. Last, we give perspectives in section 8 and conclude in section 9.

2 Forewords

2.1 Mathematical preliminaries

Let $f : \mathbb{Z}^2 \rightarrow \mathbb{N}$ be a 2D regular image having values on a totally ordered set and let $\mathcal{N}$ be a neighborhood on the 2D grid (typically the 4- or 8-connectivity). Given a set of points X, we note $CC(X) \subset \mathcal{P}(\mathbb{Z}^2)$ the set of the connected components of X given the neighborhood $\mathcal{N}$.

Let $\lambda \in \mathbb{N}$ be a gray level, $[f \geq \lambda]$ is the *upper level set* of f at level λ and $CC([f \geq \lambda])$ are the *upper components*. The *upper peak component* of a pixel x at level λ noted P_x^λ is the upper component $X \in CC([f \geq \lambda])$ such that $x \in X$. When λ is omitted as in P_x, λ is implicitely $f(x)$, and P_x is said to be the *upper level component* of x.

The family $\{CC([f \geq \lambda])\}_\lambda$ is increasing, each element of $CC([f \geq \lambda + 1])$ being included in those of $CC([f \geq \lambda])$, this family can be represented with an inclusion tree called the max-tree.

2.2 Max-tree representation

Max-trees can be represented in a compact way using the representation from [24]. An image is used to store the *parent* relationship between nodes. For that purpose, a node is represented by one of its pixel, so-called the *canonical* element. Every other pixel that belongs to the node are *linked* to the canonical element (directly or indirectly by other non-canonical element of the component). A tree is said *canonical* if every path are compressed, i.e., if every pixel is directly linked to a canonical element. *Canonicalization* consists in following the *parent* path from each pixel to the first canonical element and replacing it by a direct edge. This representation is illustrated on figure 3. The *parent* image in

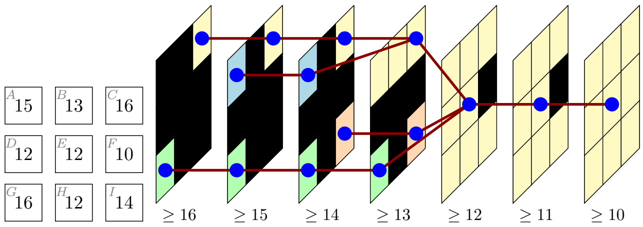


Fig. 2: Level set decomposition of an image and its max-tree. The connected components of $[f \geq \lambda]$ are included in those from $[f \geq \lambda - 1]$ and form an inclusion tree.

(b) is not canonical because some pixels (namely B, D, G, I) do not point to canonical elements. Since the canonical elements are arbitrary chosen, it may have several valid *parent* images. To ensure the uniqueness of the representation, we need a total ordering between pixels, e.g. the scanning order, that designates the bottom-right most pixel as the canonical element of the component. More formally, let $\prec$ be the total order between pixel:

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

A special point *null* is used as the root's parent and verifies $\forall p, null \prec p$ (it is the infimum over $\prec$). The parent image should also meet the following two conditions: 1) $\forall p, parent(p) \prec p$ and 2) $\forall p, parent(p)$ is canonical (the tree is canonicalized).

3 State-of-the-Art

3.1 Sequential Max-tree algorithms

Immersion-based algorithms [24], [25] are based on the Tarjan's Union-Find algorithm and are building the tree from leaves to root. It starts with the sort of the pixels of the image. Disjoint sets are created for each pixel that are

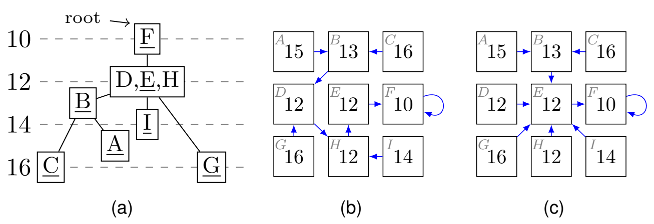


Fig. 3: *parent* image encoding a max-tree. (a) is the original image with pixel ids in gray. (b) is the max-tree of (a), some pixels appear underlined as they are randomly chosen to be the canonical elements. The *parent* relation appears in blue in (a). (a) is not path-compressed as some pixels point to non-canonical elements. (c) is the result of the canonicalization of (a).

Algorithm 1 Scheme of a union-find-based max-tree algorithm.

```
1: procedure MAXTREE(f)
2:     S ← sort (≺) pixels increasing
3:     for p in S backward do
4:         MAKE-SET(parent, p)
5:         for n in N(p) processed do
6:             r ← FIND-ROOT(parent, n)
7:             if r ≠ p then UNION(parent, p, r)

1: procedure FLATTEN(f)
2:     for p in S forward do
3:         q ← parent(p)
4:         if q is not null and f(parent(q)) == f(q) then
5:             parent(p) ← parent(q)
```

merged increasingly according to their gray levels. The process is similar to the Union-find based connected component labeling algorithms where each connected set is encoded as a tree, but it adds a constraint on the merge order. The algorithm is sketched on algorithm 1. It relies on three operations that update the disjoint trees:

- MAKE-SET(parent, x) creates the singleton set $\{x\}$. It basically sets $parent(p) \leftarrow null$
- FIND-ROOT(parent, x) follows the chain of *parent* until the root.
- UNION(parent, x, y) makes the root of x's tree to the root of y's tree.

Union-find based algorithms have a quasi-linear complexity, provided that: (a) the pixels can be sorted in linear time (eg: using radix-sort); (b) *find-root* implements the path-compression technique that updates the parent pointer of all the nodes of the chain (it also implies to have two separate structures, one encoding the compressed tree and another one encoding the real max-tree); (c) *union* uses the union-by-rank technique and chooses the new root so that the tree remains balanced. nce the tree constructed, the FLATTEN procedure in algorithm 1 is responsible to canonicalize the tree. The nodes are traversed from root to leaves to propagate the *canonical* property. Indeed, at line 4, $parent(q)$ is ensured to be a representative node.

Flood-based algorithms [2], [18], [31], [32] proceed in the opposite way and are building the tree from root to leaves. They start from a point (generally the root, i.e. the global minimum) and are flooding the peak component located at this point with a depth-first traversal graph

Algorithm 2 1D-Maxtree algorithm.

```
function UNSTACK(f, r, lvl)
    while !StackEmpty() and lvl ≤ f(StackTop()) do
        parent(r) ← StackPop()
        r ← parent(r)
    return r
procedure 1D-MAXTREE(f, parent)
    r ← start_index
    for all p starting at start_index + 1 do
        if f(r) < f(p) then
            StackPush(r)
            r ← p
        else if f(r) = f(p) then parent(p) ← r
        else
            r ← UNSTACK(f, r, f(p))
            if f(r) > f(p) then
                parent(r) ← p
                r ← p
            else parent(p) ← r
    r ← UNSTACK(f, r, −∞)
    parent(r) ← null
```

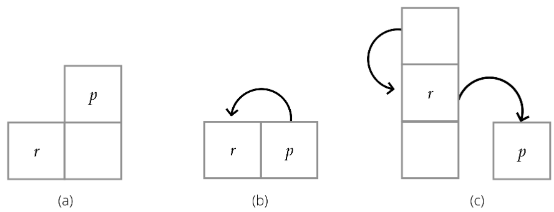


Fig. 4: Three possible cases during the 1D algorithm

pattern. The pixels at the border of this peak component are queued in a priority queue for later processing. Once the peak-component flooded, a local subtree has actually been built and is then attached to the parent node. Then, the process continues in a recursive way with the point at highest priority in the queue. The process ends when the whole image is flooded. Flood-based algorithms are generally faster than their union-find based counterparts, especially for low-dynamic range images [33] where stacks and hierarchical queues are used to remove the recursion and implement efficiently the processing queue.

1D-Maxtree algorithm [34] is a linear algorithm (algorithm 2) that, as the name suggests, is dedicated to the max-tree computation of 1D-signals. It is single pass and very memory efficient as it only requires a constant-size stack (actually $O(min(G, N))$). The algorithm proceeds as follows:

The algorithm iterates over the 1D image starting at $start\_index$ and acts based on the gray level difference between the current and last pixel (p and r respectively). Three cases can arise as depicted in figure 4. If the gray level increases ($f(r) < f(p)$), last pixel is pushed into the stack, creating a new connected component. If the gray level stays the same ($f(r) = f(p)$), the current connected component is extended. The last case, when the gray level decreases, is divided into three scenarios: If the stack is empty, no intermediate component exists between p and r, thus p

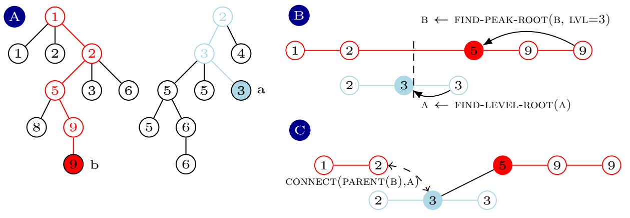


Fig. 5: Merging max-trees. (A) Two disjoints trees to merge by linking a and b. (B) The corresponding branches are followed until the nodes to merge are found. (C) The parent pointer of b is updated, and the connection is made recursively with the old parent.

Algorithm 3 Algorithm used to merge two max-trees.

```
function FIND-PEAK-ROOT(parent, x, lvl)
    q ← parent(x)
    while q not null and lvl ≤ f(q) do
        x ← EXCHANGE(q, parent(q))
    return x, q

function FIND-LEVEL-ROOT(parent, x)
    return FIND-PEAK-ROOT(parent, x, f(x))

procedure CONNECT(a,b)
    while b not null do
        if f(b) < f(a) then SWAP(a, b)
        a, _ ← FIND-LEVEL-ROOT(parent, a)
        b, _ ← FIND-PEAK-ROOT(parent, b, f(a))
        if b ≺ a then SWAP(a, b)
        if a = b then return
        b ← EXCHANGE(parent(b), a)                ▷ merge-set
```

directly becomes r parent's. If the stack is not empty, the parent of p might be in the stack. The UNSTACK() function pops the stack while the levels are greater than the current level, linking components in the stack along the way. If a lower gray level is found at the top, p is the intermediate component between the last and current stack top thus r becomes parent of p. Else, the stack has been emptied, there is no intermediate component (and thus $f(r) > f(p)$), p can be set as parent of r. Finally, if there are elements left in the stack we can directly link them together using UNSTACK() and set the last element as the root of the tree.

3.2 Parallel and distributed max-tree algorithms

Merge-based algorithms rely on a divide-and-conquer strategy to compute the max-tree. The image is split into tiles for which local max-trees are computed. Then, the local max-trees are merged hierarchically with a reduction pattern by connecting the pixels on the tile boundaries. The figure 5 illustrates the merge of two max-trees that have to be connected through the edge (a, b). The corresponding algorithm is given in algorithm 3. First, FIND-PEAK-ROOT follows up the chains of a and b to reach the nodes that have to be updated. Supposing $f(a) < f(b)$, it requires to reach the level-root of a (because it may not be the canonical element) and reaching the root of the peak component $P_b^f(a)$, i.e., the highest node with level not greater than $f(a)$. The procedure returns the canonical node and its parent (which is not used for now). If nodes have both the same level, the merge applies on flat-zones, we need to select the "smallest" representative in terms of $\prec$ to be the new root. The parent of b is updated and CONNECT is called recursively on the parent until reaching the root.

This process is well-adapted to the parallel construction of the max-tree because each tile computation is independent. The first parallel algorithms [26], [27], [28] were using this algorithm on shared-memory systems with scalable results (almost linear in the number of threads). As the image were getting bigger, the same strategies were adopted for a distributed computation of the max-tree [29], [35] with the extra burden of minimizing memory exchanges between (cluster) nodes using border max-trees. This idea is even pushed further in [30], [36] with a distributed representation of the max-tree based on border max-trees that avoids storing the final tree in shared-memory and enables distributed tree processing.

3.3 Union-find based Connected Component Labeling on GPU

As Max-Tree computation is close to Connected Component Labeling (CCL), it is interesting to look at the State-of-the-Art for CCL algorithms. The key part of those algorithms is the Union-Find structure to compute the equivalences between pixels.

Concurrent Union-Find is an old problem [37], [38], but until recently, it was not used for CCL. The first CCL implementation on GPU cames in 2015 from Komura [39] using a concurrent Union-Find. The paper also introduces the Komura Equivalence (KE) that modifies the initialization step of the UF to resolve on-the-fly some equivalences and avoids the creation of temporary single-node equivalence trees. In 2018, the Playne algorithme [40] improves upon KE by analyzing pixel patterns in order to avoid redundant merge operations. In 2018, the HA algorithm [41] sped up the Playne algorithm by using small segments (32-pixel wide, the size of a warp) of pixels and CUDA intrinsics. Later, in 2019, the BKE algorithm [42] (Block-based Komura Equivalence) improves the Playne algorithm by exploiting a property of 8-connected components to process pixels in 2×2 blocks. In 2021, the Full-Length Segment Labeling (FLSL) is able to tackle segment longer than the warp size, and advanced algorithmic optimizations reduces the voting bottleneck of Connected Component Analysis algorithms [43].

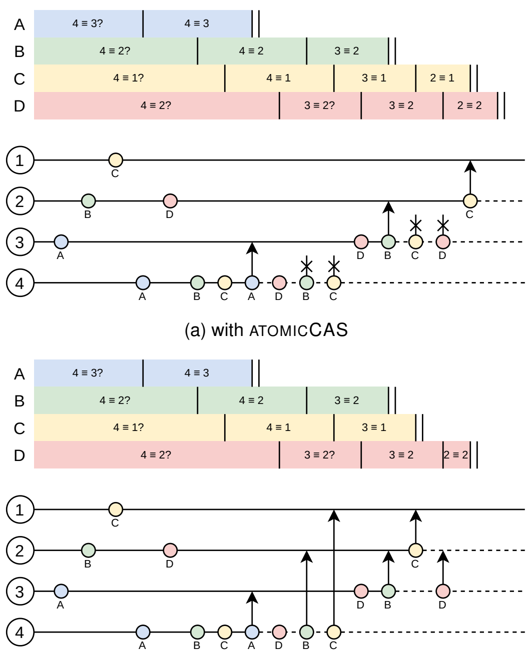
The concurrent Union-Find consists in retrying with a CAS-loop the link between two roots. If the higher root is updated by another thread, the link is retried with the new parent of the formerly higher root. This algorithm is in fact wait-free as the work of a thread is bounded by the height of the resulting tree. It is detailed in algorithm 4. While the original concurrent Union-Find was using a CAS for the retry, [39] used an ATOMICMIN to leverage the natural ordering of labels and reduce the practical number of retries.

Figure 6 is a timeline example that demonstrates how multiple threads can concurrently modify the Union-Find structure. It also shows the difference between the ATOMICCAS-based function and the ATOMICMIN-based one (respectively figure 6a and figure 6b). We can clearly see that the ATOMICMIN version have one less read-modify-

Algorithm 4 Concurrent Union-Find on GPU for Connected Component Labeling.

```
procedure FIND-ROOT(L, a)
    while a ≠ L[a] do
        a ← L[a]
    return a

procedure UNION(L, a, b)
    a ← FIND-ROOT(L, a)
    b ← FIND-ROOT(L, b)
    while a ≠ b do
        if b < a then SWAP(a, b)
        c ← ATOMICMIN(L[b], a)
        if c = b then return
        b ← c
```



(a) with ATOMICCAS

(b) with ATOMICMIN

Fig. 6: Example of a lock-free Union-Find. Four threads (A, B, C and D) process the following unions: $4 \equiv 3$, $4 \equiv 2$, $4 \equiv 1$ and $4 \equiv 2$. The schemas are a timeline of the states of each thread and the operations in memory. A circle corresponds to a read, while an arrow correspond to a write, the end of the arrow being the value written. An arrow with a circle is a read-modify-write atomic, and the cross signifies a failure (only for CAS). The solid lines for labels represent root labels, while dashed ones correspond to labels whose parent have been set. For the sake of demonstration, threads follow a round-robin scheduling.

write atomic, and that the final tree is flatter (label 4 points to label 1 directly).

4 Union-Find based max-tree algorithm on GPU

4.1 Sort-less max-tree algorithm

The first step of algorithm 1 consists in sorting the pixels so that merging the disjoints sub-trees with the union-find occur from the leaves to the root of the max-tree. Using the parallel strategies depicted in section 3.2, we can actually build a max-tree using only tree merges. It consists in calling CONNECT for every edge in the image as shown in algorithm 5. The complexity of this algorithm is $O(V \cdot N)$ and thus highly depends on the number of levels.

Algorithm 5 Sort-less max-tree algorithm.

```
procedure MAXTREE(f)
    for all pair of neighbors (p, q) do
        CONNECT(p,q)

procedure FLATTEN(f)
    for all p do
        q ← parent(p)
        parent(p) ← FIND-LEVEL-ROOT(parent, q)
```

4.2 Concurrent computation of the max-tree

As it stands, the current algorithm cannot be run concurrently as there may be data races when updating the *parent* pointer in algorithm 3 on line 8. Even if read and write operations were atomic, an update might not be seen by the other threads (lost-update problem). The solution lies in the same technique used for the concurrent labeling algorithm exposed in section 3.3. A read-modify-write operation is used when updating the parent pointer. The situation is a bit more complex as we need to select the right node if a conflict occurs. In algorithm 4, ATOMICMIN is used because the representative is chosen to be the lowest label. Choosing the minimum (or the maximum) prevents the creation of cycle that could occur with concurrent updates.

With the max-tree, the same problem arises if there is no total order imposed on pixels. In algorithm 6, line 8 uses ATOMICMAX based on $\prec$ to select the right parent when concurrent updates occur. Suppose that the parent of a node has to be updated concurrently with q_1, q_2 and q_3. The new root will be the "lowest" one (i.e., the one with the highest level). If there is several "lowest" nodes, we then need to select the right representative which is the most bottom-right node. In case of conflict, two cases occur:

- The ATOMICMAX updates the value and makes progress. In this case, the old value of *parent(b)* is held in *b*, we need to merge the old parent node with a.
- The ATOMICMAX does not update the value and fails to make progress. *parent(b)* has been updated by another thread and is stored in *b*. We still have $a \prec b$ (otherwise it would have succeeded), thus it retries to connect the updated parent (*b*) with a.

Algorithm 6 uses an ATOMICMAX based on a non-trivial relation that involves comparing gray levels and pixel indexes, but most GPUs support atomic operation on trivial

Algorithm 6 Concurrent lock-free version of algorithm 3

```
1: procedure CONNECT(a,b)
2:     while b not null do
3:         if f(b) < f(a) then SWAP(a, b)
4:         a, _ ← FIND-LEVEL-ROOT(parent, a)
5:         b, _ ← FIND-PEAK-ROOT(parent, b, f(a))
6:         if b ≺ a then SWAP(a, b)
7:         if a = b then return
8:         b ← ATOMICMAX≺(parent(b), a)            ▷ union
```

types only. To overcome this limitation, the parent image is used to store 32-bit records such that comparing the records with $\prec$ is equivalent to comparing the binary representation of the record. Using the LSB 0 bit numbering, *parent(x)* stores in 32 bits:

x's current level	x's parent level	x's parent index
32	24	16 — 0

Supposing the pixel are 1-based indexed, the $null$ value is thus encoded with *0* (0 for all fields). In algorithm 6, line 8 becomes:

8: $newB \leftarrow [f(b), f(a), a]$
9: $[\_, \_, b] \leftarrow$ ATOMICMAX$(parent(b), (\texttt{uint32_t})newB)$

When $parent(x)$ is updated, the *current level* always remains the same, but the *parent level* and the *parent index* might be updated by this new parent if they are greater than the original ones.

This representation works for images with at most $2^{16} - 1$ pixels because the parent index is 1-based and encoded in 16 bits. To handle 32-bit indexes, this representation can be extended to 64 bits at the cost of doubling the shared memory usage. Alternatively, an ATOMICCAS (see algorithm 7) can be used instead of this representation but leads to more *retries*. Indeed, in case of conflicts, only one thread makes progress with a CAS; while using an ATOMICMAX, several threads may update the same parent in the same turn.

Algorithm 7 Concurrent lock-free CONNECT with a CAS

```
procedure CONNECT(a,b)
    while b not null do
        if f(b) < f(a) then SWAP(a, b)
        a, A ← FIND-LEVEL-ROOT(parent, a)
        b, B ← FIND-PEAK-ROOT(parent, b, f(a))
        if b ≺ a then SWAP(a, b) SWAP(A, B)
        if a = b then return
        old ← ATOMICCAS(parent(b), B, a)        ▷ Try
        if old = B then                          ▷ If false → retry
            b ← old
```

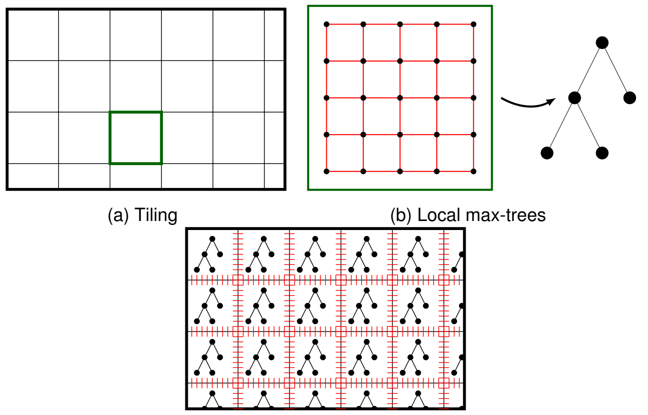
4.3 CUDA Implementation Details

The algorithm has been adapted to fit the memory model hierarchy of CUDA and minimize global memory load and store. It has three main parts depicted in figure 7.

Local max-tree construction. The image is tiled into blocks with as many threads as pixels in the block. (a) All the threads start with loading the input image values into the field *current level* of the *parent* image in shared memory; – the size of the block is small enough for using 16-bit local indexes – (b) CONNECT is called for every pair of neighbors (twice per thread if using the 4-connectivity) as shown in figure 7b; (c) the *parent* image is flattened to make the local max-tree canonical; (d) the *parent* image block is copied from shared memory to global memory, converting local indexes to 32-bit global indexes.

Global max-tree merging (figure 7c). The local max-trees have to be merged along the block boundaries. With as many threads as the number of pixel pairs on the block boundaries, CONNECT is called with the pair of pixels along vertical and horizontal boundaries. The merge acts on global memory with 32-bit indexes (so using an ATOMICCAS in algorithm 6).

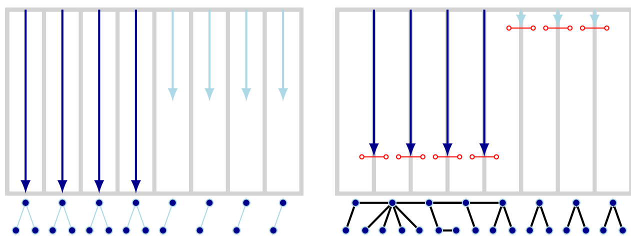
Flattening. The *parent* image is tiled into blocks and each block is flattened.



(a) Tiling (b) Local max-trees

(c) Max-tree merging on boundaries

Fig. 7: Hierarchical computation of the max-tree. (a-b) Local max-trees are first computed on tiles by thread blocks in shared memory and then merged in global memory (c). Each red edge leads to concurrent calls to CONNECT with the corresponding endpoints.



(a) (b)

Fig. 8: 1D-optimized max-tree building. Two thread warps first build the column max-trees (a) then merge concurrently by iteratively calling CONNECT on the both side of the boundaries (b).

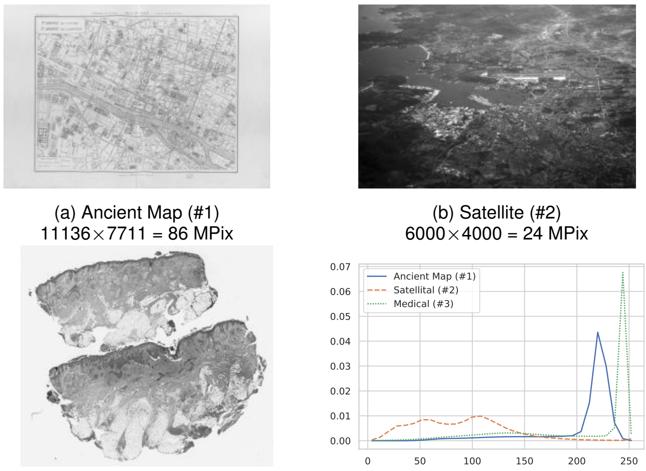
5 OPTIMIZED LOCAL MAX-TREE

As depicted in section 4.1, to build a max-tree on GPU, one could just call the concurrent CONNECT for every edge in the image. This has the drawback of generating a lot of concurrent writes which hurts performance [43] even when issued in shared memory. To reduce the number of calls to CONNECT, local max-trees are first built column-wise using the 1D algorithm depicted in algorithm 2. This algorithm is inherently fast because there are no data dependencies between threads. Inside each tile, each thread inside each thread block starts by fetching the first pixel of its column i.e. the first line of the tile. As each thread unravels its execution path, the thread block keeps on fetching, in a coalesced manner, the lines of the tile. Since each thread is working on its own column max-tree, no atomic nor syncing operations are required. There are also no bank-conflict as each thread works on its own bank.

Once local max-trees are built inside the tile, each thread (minus one) sweeps again top to bottom, calling the concurrent CONNECT on each of the remaining edges. This

Configuration	Device	Model
Embedded	CPU	Jetson TX1 - 4 ARM Cortex-A57 @ 1.9Ghz
Embedded	GPU	Jetson TX1 - 256 Maxwell Cores @ 1.3Ghz
Desktop 1	CPU	2 i7-7567U cores @ 3.50GHz
Desktop 1	GPU	GeForce GTX 1650 - 896 Cores @ 1.5Ghz
Desktop 2	CPU	6 i7-9750H cores @ 2.60GHz
Desktop 2	GPU	GeForce RTX 2060 - 2176 Cores @ 1.4Ghz
Compute 1	CPU	20 Xeon Silver 4210 cores @ 2.20GHz
Compute 1	GPU	Quadro RTX 8000 - 4608 cores @ 1.4Ghz
Compute 2	CPU	16 Xeon Silver 4110 cores @ 2.10GHz
Compute 2	GPU	Tesla V100 - 5120 cores @ 1.3GHz

TABLE 1: Hardware devices in the benchmark setup.



(a) Ancient Map (#1)
11136×7711 = 86 MPix

(b) Satellite (#2)
6000×4000 = 24 MPix

(c) Medical brightfield microscopy (#3)
27744×24352 = 675 MPix

(d) Gray levels distribution

Fig. 9: The dataset used for benchmarking and the gray-level distribution of these images.

effectively cuts by half the number of CONNECT needed and significantly improves performance. Those two steps are shown in figure 8.

6 Performance Evaluation

We compare the performance of our GPU implementation to the State-of-the-Art CPU ones. For the CPU part, the sequential Salembier's algorithm (*Salembier ST*) implemented without recursion and pre-allocated priority queues is used. The parallel version (*Salembier MT*) uses a divide-and-conquer strategy. It runs Salembier's algorithm on tiles and merges the trees hierarchically as described in [33]. The two GPU versions are the algorithm described in section 4 (*Base*) and its 1D-optimization from section 5 (*Optim 1D*). They are benchmarked with and without the memory transfer from and to the host memory. We have benchmarked on three profiles: embedded systems, desktop computers, and compute servers. Their descriptions are shown in table 1. The benchmark also includes several image types (satellite images, medical images, and documents) to consider the variability of real image contents. These images are shown on figure 9.

As one can see on figure 10, our GPUs versions outperform the sequential and the parallel version running on CPUs on comparable devices by at least a factor 5 on a single image. However, when processing a stream of images, the transfer latency from the host memory can be hidden and only the GPU kernel time has to be taken. Then, performance of *1D Optimized* is one order of magnitude higher than those on CPU.

#	Algorithm	Local trees	Merge Global	Flatten	Total time
1	Base	88.7 (49%)	71.7 (40%)	19.8 (11%)	180.2
1	Optim 1D	45.3 (33%)	– (52%)	– (15%)	136.9
2	Base	23.0 (61%)	9.4 (25%)	5.3 (14%)	37.8
2	Optim 1D	11.7 (44%)	– (36%)	– (20%)	26.5
3	Base	646.9 (50%)	521.9 (40%)	120.4 (9%)	1289.2
3	Optim 1D	358.5 (36%)	– (52%)	– (12%)	1000.3

TABLE 2: Processing time (in milliseconds) of each kernel with the *Desktop 1* configuration on test images.

When comparing the GPU algorithms, the 1D-optimization improves the performance by about 30% on average. As depicted in table 2, the local max-trees computation time that represents more than half the GPU compute time is reduced by 50%. Several factors may explain this performance gain. First, the work per thread is much higher because a single thread processes a full column (16× higher because the columns are 16 pixels high). While it reduces the theoretical occupancy of the Streaming Multiprocessor (SM) and the number of active warps per SM, it actually leads to less contention between threads because it reduces the number of concurrent *atomic writes* trying to update the same parent. From our experiment, the stall of individual warps (mostly due to atomics) and the thread divergence (mostly due to the FIND-PEAK-ROOT loop) are reduced by 25%.

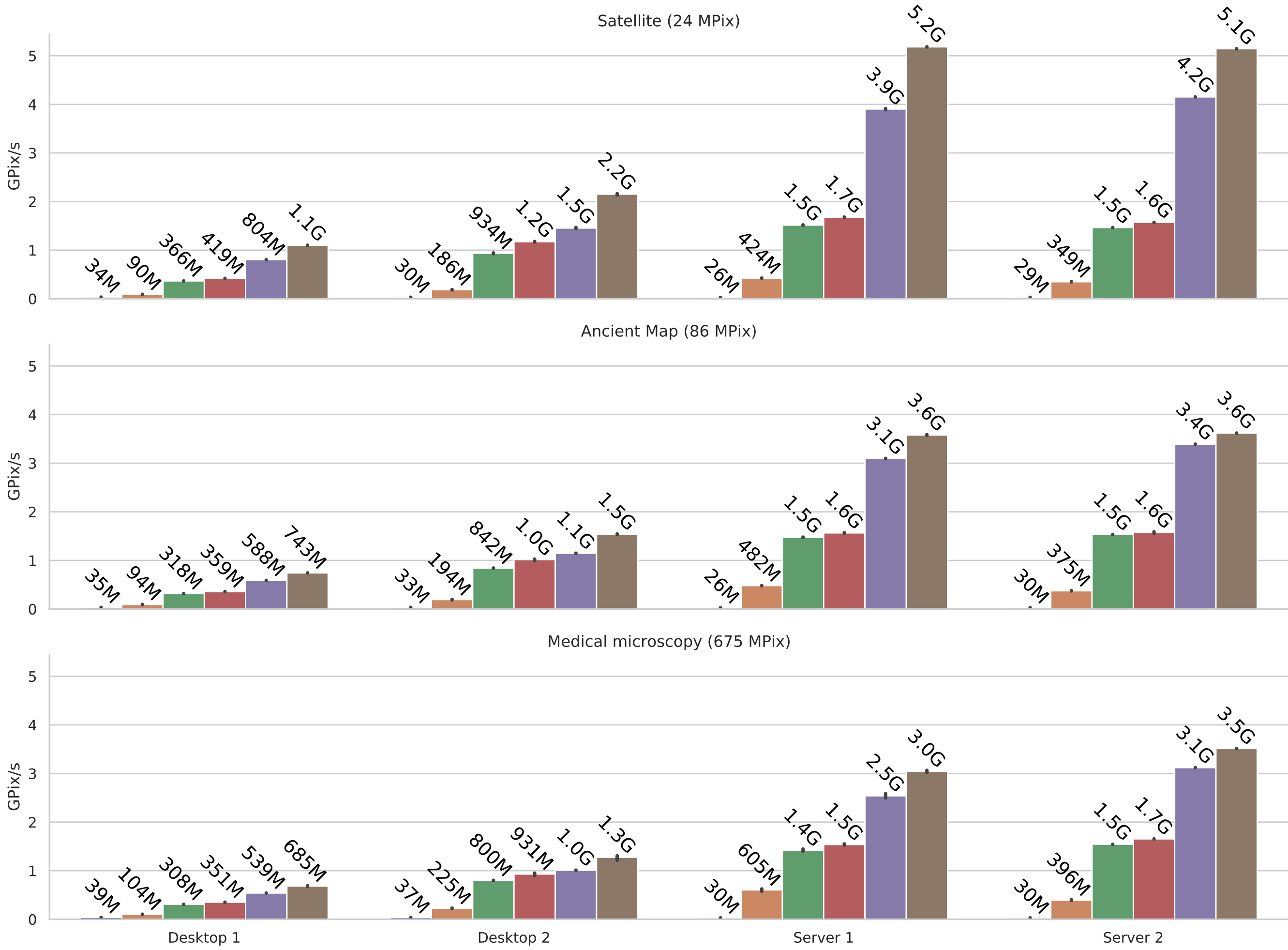
It is worth mentioning that to speed up column merging and flattening of our *1D-optimized*, another work organization was tried. It was running one thread per pixel, only using one thread per column during the 1D-construction part (yielding many idle threads) and then, using all the threads for column merging and flattening (more parallel work). However, this approach reduced the performance and showed that augmenting the work-per-thread to decrease the contention when merging was better.

7 Extension to the 8-connectivity

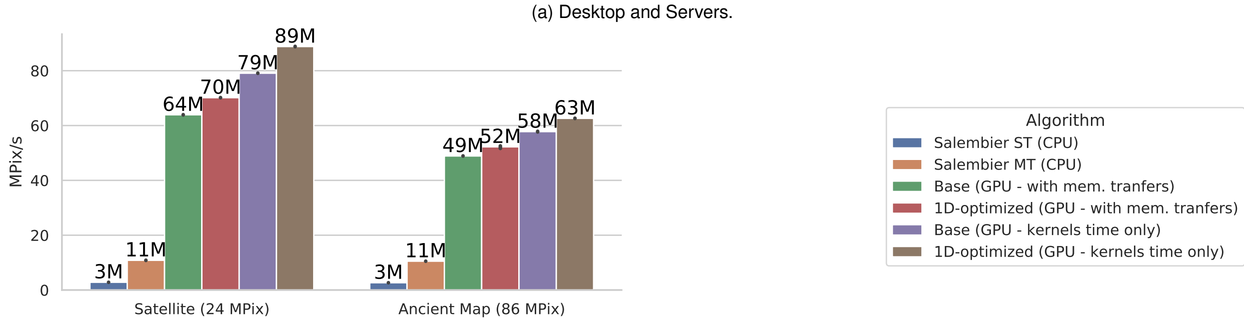
7.1 Grid simplification

The extension of the algorithms to 8-connectivity is straightforward. When merging columns or tiles, one just need to call CONNECT in the diagonal directions (see figure 11a). It follows that the number of CONNECT calls doubles during the local construction of the *Base* algorithm and triples when merging columns or at tile boundaries. This is experimentally confirmed by figure 12, showing that the processing time of the *base* algorithm increases by 120% on average, while the time of the *1D-optimized* algorithm is multiplied by a factor 2.5. Again, it confirms that CONNECT is an expensive operation, and we should minimize its use.

Let an image f with two disjoints rectangular regions that connect over the boundary pixels $U = \{u_1, u_2, \cdots u_n\}$ and $V = \{v_1, v_2, \cdots v_k\}$. Let E the set of edges that connects U to V. Considering the 4-connected grid G_4, $E = \bigcup_{1 \leq k \leq n} E_k$ where $E_k = \{(u_k, v_k)\}$, i.e, there are n calls to CONNECT. With the 8-connected grid G_8, $E_k = \{(u_k, v_k), (u_{k-1}, v_k), (u_k, v_{k-1})\}$ (for $k > 1$), so there



(a) Desktop and Servers.



(b) Embedded system. (Performances on the microscopy image are missing due to the memory limit.)

Fig. 10: Performance comparison of the max-tree computation with CPUs and GPUs on many hardware configurations and various image types.

are three calls to CONNECT for each k. To reduce the number of calls to CONNECT, we rely on the following propositions.

Proposition 1. *Considering the 8-connectivity, the max-tree of f with edges E is equivalent to the max-tree of f with the set of edges $E' = \cup E'_k$ where $E'_1 = E_1$ and for $k > 1$:*

$$E'_k = \begin{cases} \{(u_k, v_k)\} & \text{if } f(u_k) > f(u_{k-1}) \wedge f(v_k) > f(v_{k-1}) \\ \{(u_{k-1}, v_k)\} & \text{if } f(u_k) \leq f(u_{k-1}) \wedge f(v_k) > f(v_{k-1}) \\ \{(u_k, v_{k-1})\} & \text{if } f(u_k) > f(u_{k-1}) \wedge f(v_k) \leq f(v_{k-1}) \\ \emptyset & \text{otherwise} \end{cases}$$

Proof. It is worth mentioning that an edge (u, v) is only involved in the construction of P_u^α and P_v^α with $\alpha \leq \min(f(u), f(v))$. Let $k \in \mathbb{N}, 1 \leq k \leq n$ and

$$(a, A) = (u_k, u_{k-1}) \text{ if } f(u_k) < f(u_{k-1}) \text{ else } (u_{k-1}, u_k)$$
$$(b, B) = (v_k, v_{k-1}) \text{ if } f(v_k) < f(v_{k-1}) \text{ else } (v_{k-1}, v_k)$$

Then (a, b), (a, B) and (A, b) are useless edges in the max-tree construction as they do no change the peak components.

Let $\alpha = \min(f(a), f(b))$. a and b connects through the path $a \leftrightarrow A \leftrightarrow B \leftrightarrow b$, with $\min(f(A), f(B), f(a), f(b)) \geq \alpha$, so $P_a^\alpha = P_b^\alpha$ with or without (a, b).

Let $\alpha = \min(f(a), f(B))$. a and B connects through the path $a \leftrightarrow A \leftrightarrow B$ with $\min(f(a), f(A), f(B)) \geq \alpha$, so $P_a^\alpha = P_B^\alpha$ with

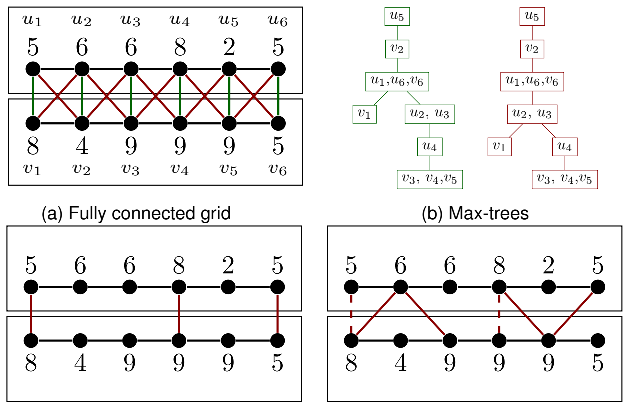


Fig. 11: Connecting columns and tile boundaries. The 4-connected and 8-connected neighborhood (a) and the corresponding max-trees in 4-connectivity (b-left) and 8-connectivity (b-right). The simplified connectivity grids for 4-connectivity (c-left) and 8-connecivity (c-right) give the same max-trees.

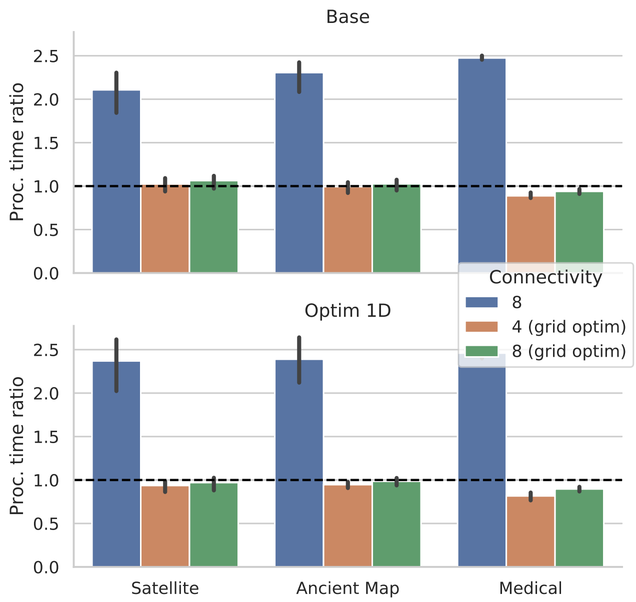


Fig. 12: Impact of the 8-connected neighborhood and the optimized connectivity grid. The processing time of the 4-connectivity kernels is used as a baseline. The processing time of each variant is expressed as a factor of the baseline time and averaged over the hardware configurations from table 1 (lower is better).

or without (a, B).

Let $\alpha = \min(A, b)$. A and b connects through the path $A \leftrightarrow B \leftrightarrow b$ with $\min(f(A), f(B), f(b)) \geq \alpha$, so $P_A^\alpha = P_b^\alpha$ with or without (A, b).

By recursively removing the useless edges for all k (in whichever order), we hold proposition 1. Note that the edge (u_{k-1}, v_{k-1}) might be missing for some k. Nevertheless the equivalence holds as there still exists an equivalent path. □

Proposition 2. *Considering the 8-connectivity, the max-tree of f with edges E' is equivalent to the max-tree of f with the set of*

Image	Base (8b)	Base (16b)	Halo (16b)
#1	804 MPix/s	29.7 MPix/s	47.9 MPix/s
#2	588 MPix/s	18.4 MPix/s	29.6 MPix/s
#3	539 MPix/s	15.7 MPix/s	26.7 MPix/s

TABLE 3: Desktop 1 performance on 16-bit HDR test images compared to 8-bit images.

edges $E'' = \cup E_k''$ where $E_n'' = E_n'$ and for $k < n$:

$$E_k'' = \begin{cases} E_k' \setminus \{(u_k, v_k)\} & \text{if } f(u_k) < f(u_{k+1}) \vee f(v_k) < f(v_{k+1}) \\ E_k' & \text{otherwise} \end{cases}$$

Proof. Similar to the previous proof. There exists an alternative path that do not pass by (u_k, v_k). □

From proposition 1 and proposition 2, it follows that $|E_k''| \leq 1$. In the context of the merging tile boundaries, it follows that **there is at most one CONNECT per thread**. The grid simplification can also be applied with the 4-connected grid using proposition 3 which also enables to remove some unnecessary calls to CONNECT.

Proposition 3. *Considering the 4-connectivity, the max-tree of f with edges E is equivalent to the max-tree of f with the set of edges $E''' = \cup E_k'''$ where:*

$$E_k''' = \begin{cases} \emptyset & \text{if } \min(f(u_{k-1}, f(v_{k-1})) < \min(f(u_k), f(v_k)) \\ \emptyset & \text{if } \min(f(u_{k+1}, f(v_{k+1})) < \min(f(u_k), f(v_k)) \\ E_k & \text{otherwise} \end{cases}$$

In figure 11c, we illustrate the grid simplification of the fully connected grid depicted in figure 11a and show that they lead to the exact same max-trees. The dashed edges for 8-connected simplified grid represent the edges removed from E' to E''.

7.2 Performance

Figure 12 shows the impact of the grid simplification on performance. In particular, it shows that the grid simplification for the 8-connectivity is always beneficial, and we get back to the same running times as with the 4-connectivity baselines. With a 4-connected neighborhood, the grid simplification benefits are less obvious. This exhibits an interesting trade-off to have between "*more work, more contention but better work balancing and better parallelism*" and "*less work but unbalanced work and less parallelism opportunity*". Indeed, in the extreme case where all boundary edges are replaced by a single link, it leads to a single thread that merges the whole branch while the other threads are idle. The level of work done in parallel eventually drops while the branch could have been merged by several threads concurrently.

8 PERSPECTIVE

High dynamic range images. As it stands, nothing prevents the current algorithm from running on high-quantized data. However, as described in [33], CONNECT is not efficient for those data as its performance depends on the length of the branch (that drastically increases with the number of bits of the values). To figure out the performance penalty of the

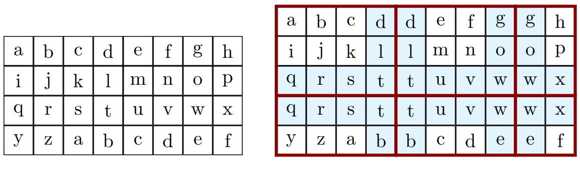


Fig. 13: Tile border duplication with tiles of size 4×3. (a) Original image (b) Image with halo.

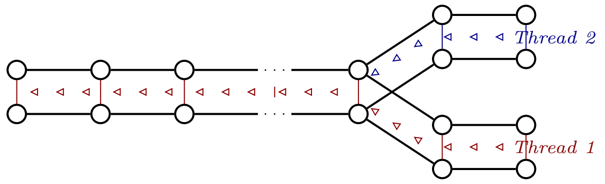


Fig. 14: Unbalanced merge work between threads. If two threads of the same warp merge two trees but have a low common ancestor, one thread is stalled and waits for the other to finish.

quantization on our algorithm, we have transformed the original 24-bit RGB test images into 16-bit images. Following the protocol in [44], the luminance of RGB values is computed by $0.2126R + 0.7152G + 0.0722B$, and the value is quantized on 16 bits. In table 3, the second column shows the performance of our *Base* algorithm on 16-bit images and highlights a drastic slow-down (about 30 times slower). Actually, this slow-down is mostly due to the merge of max-trees in global memory that takes 90% of the total compute time. Indeed, with 16-bit images, there are more (canonical) nodes and the chain are longer. It follows that we hit the global memory at more random location (the L2 cache hit rate is less than 40%) and the memory latency is the first cause of the thread stalls. Longer chains also induce less workload balancing between threads. Indeed, suppose that threads have to merge branches from two disjoint trees, but the branches meet soon in the hierarchy. One thread is going to be elected to merge the whole chain, while the others are being stalled waiting for the elected thread to finish. This problem is illustrated on figure 14. This thread divergence causes a low number of active threads per warp (less than 3 actives thread/warp on some images).

In [29], the authors suggest duplicating the tile boundaries (called *halo*, see figure 13) so that CONNECT is called on global memory on two nodes with the same levels. The cost of the first FIND-PEAK-ROOT in global memory is thus lower and compensates the extra-work induced to process the halo as shown in the third column of table 3 (adding and removing the halo counts only for 2% of the total time). Even if it improves slightly the performance, processing 16-bit images is still much slower than 8-bit images (up to 20 times slower).

Some interesting approaches have been proposed in [44], where the max-tree construction is "stratified" by buckets. An extension of our algorithm for high-dynamic images could probably benefit from these ideas, running the max-tree construction at different low-quantized bucket and eventually, merging them.

3D images. The presented algorithm can easily be extended to 3D images. The max-tree algorithm depicted in section 5 could build the max-trees of the 2D slices. Then, the max-trees of the slices would be merged depth-wise just like we did for merging the 2D tiles vertically and horizontally. However, adding those z-connections would drastically reduce performance. Indeed, the maximal 26-connectivity would lead to a large amount of CONNECT issued in global memory during global max-tree merging. Even if the grid simplification trick could probably be extended in 3D, managing such a high amount of connection in global memory stays challenging.

Tree of Shapes (ToS) and Alpha-tree. As depicted in [22] the ToS can be computed using a max-tree algorithm. As stated, this approach benefits from efficient component-tree implementation. The newly presented max-tree GPU algorithm could serve as a foundation for efficient ToS computation. However, the method from [22] requires first to transform the input images with 3 steps, namely *interpolation*, *immersion* and *propagation*. The first two steps can be trivially parallelized on GPU. Nevertheless, porting the *propagation* (that as the name suggests, uses a propagation flow) remains challenging on GPU.

Our algorithm seems to be also adapted to compute the α-Tree (*a.k.a.* the quasi-flat zone hierarchy). In this representation, the flat-zones are the leaves of the tree while the internal nodes are the edges of minimum spanning tree (MST) ordered by altitude. The most common α-tree algorithm [45], [46] is based on Kruskal's MST algorithm and relies on the Union-Find. In [47], it has been observed that the α-tree is closely related to computing the min-tree on the *edge graph* of an image, and as a consequence, the adaptation looks straightforward.

9 Conclusion

In this work, the first massively parallel GPU algorithm for the computation of the max-tree has been presented. By taking advantages of the non-ending growth of the GPU computing performance, our algorithm is at least 5 times faster than the current State-of-the-Art CPU parallel algorithm and one order of magnitude faster when the memory transfer latency can be hidden. Moreover, we have proposed algorithmic variants that handle the 8-connectivity with no overhead and no added complexity. This work will especially benefit the recent researches dedicated to a distributed max-tree computation for terabyte images as it will lead to a significant speed-up in each cluster node. Not only does this new algorithm allow the integration of the max-tree computation in GPU pipelines, but it also paves the way for the portage on GPUs of many max-tree based structures as the Tree of Shapes.

Acknowledgments

The authors would like to thank Arthur Hennequin from LIP6 and LPNHE for having run our code on a V100 GPU. We would also like to thank Ilan Guenet from EPITA for helping on the performance tuning, as well as Alexandre Bourquelot and Reda Dehak from EPITA for the TX1 technical support. Image 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-2.SVS) - https://openslide.cs.cmu.edu/

REFERENCES

[1] M. Ôn Vũ Ngọc, N. Boutry, J. Fabrizio, and T. Géraud, "A new minimum barrier distance for multivariate images with applications to salient object detection, shortest path finding, and segmentation," *Computer Vision and Image Understanding*, vol. 197–198, Aug. 2020.
[2] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Trans. Image Process.*, vol. 7, no. 4, pp. 555–570, 1998.
[3] L. Vincent, "Morphological grayscale reconstruction in image analysis: applications and efficient algorithms," *IEEE Trans. Image Process.*, vol. 2, no. 2, pp. 176–201, 1993.
[4] P. Salembier and J. Serra, "Flat zones filtering, connected operators and filters by reconstruction," *IEEE Trans. Image Process.*, vol. 3, no. 8, pp. 1153–1160, 1995.
[5] Y. Xu, T. Géraud, É. Puybareau, I. Bloch, and J. Chazalon, "White matter hyperintensities segmentation in a few seconds using fully convolutional network and transfer learning," in *International MICCAI Brainlesion Workshop*. Springer, 2017, pp. 501–514.
[6] E. Puybareau and T. Géraud, "Real-time document detection in smartphone videos," in *Proceedings of the 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2018, pp. 1498–1502.
[7] R. Jones, "Component trees for image filtering and segmentation," in *IEEE Workshop on Nonlinear Signal and Image Processing*. Mackinac Island: E. Coyle, Ed., Sep. 1997.
[8] ——, "Connected filtering and segmentation using component trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, 1999.
[9] J. Hernández and B. Marcotegui, "Shape ultimate attribute opening," *Image and Vision Computing*, vol. 29, no. 8, pp. 533–545, 2011.
[10] E. R. Urbach and M. H. Wilkinson, "Shape-only granulometries and grey-scale shape filters," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*, vol. 2002, 2002, pp. 305–314.
[11] G. K. Ouzounis and M. H. Wilkinson, "Mask-based second-generation connectivity and attribute filters," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 990–1004, 2007.
[12] M. Dalla Mura, J. A. Benediktsson, B. Waske, and L. Bruzzone, "Morphological attribute profiles for the analysis of very high resolution images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 48, no. 10, pp. 3747–3762, 2010.
[13] M. Dalla Mura, J. Atli Benediktsson, B. Waske, and L. Bruzzone, "Extended profiles with morphological attribute filters for the analysis of hyperspectral data," *International Journal of Remote Sensing*, vol. 31, no. 22, pp. 5975–5991, 2010.
[14] G. K. Ouzounis, M. Pesaresi, and P. Soille, "Differential area profiles: Decomposition properties and efficient computation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 8, pp. 1533–1548, 2011.
[15] P. Bosilj, M. H. Wilkinson, E. Kijak, and S. Lefèvre, "Local 2d pattern spectra as connected region descriptors," *Mathematical Morphology Theory and Applications*, vol. 1, no. 1, 2016.
[16] B. Mirmahboub, J. Moré, D. Youssefi, A. Giros, F. Merciol, and S. Lefèvre, "Fast pattern spectra using tree representation of the image for patch retrieval," in *International Conference on Discrete Geometry and Mathematical Morphology*. Springer, 2021, pp. 107–119.
[17] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," *Image and Vision Computing*, vol. 22, no. 10, pp. 761–767, 2004.
[18] D. Nistér and H. Stewénius, "Linear time maximally stable extremal regions," in *Proceedings of the 10th European Conference on Computer Vision (ECCV)*, 2008, pp. 183–196.
[19] Y. Xu, T. Géraud, and L. Najman, "Connected filtering on tree-based shape-spaces," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 6, pp. 1126–1140, 2015.
[20] M. A. Westenberg, J. B. T. M. Roerdink, and M. H. F. Wilkinson, "Volumetric attribute filtering and interactive visualization using the max-tree representation," *IEEE Trans. Image Process.*, vol. 16, no. 12, pp. 2943–2952, 2007.
[21] P. Monasse and F. Guichard, "Fast computation of a contrast-invariant image representation," *IEEE Trans. Image Process.*, vol. 9, no. 5, pp. 860–872, 2000.
[22] E. Carlinet, S. Crozet, and T. Géraud, "The tree of shapes turned into a max-tree: a simple and efficient linear algorithm," in *Proceedings of the 25th International Conference on Image Processing (ICIP)*. IEEE, 2018, pp. 1488–1492.
[23] R. Souza, L. Tavares, L. Rittner, and R. Lotufo, "An overview of max-tree principles, algorithms and applications," in *29th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*. IEEE, 2016, pp. 15–23.
[24] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Trans. Image Process.*, vol. 15, no. 11, pp. 3531–3539, 2006.
[25] 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 *Proceedings of the 18th International Conference of Image Processing (ICIP)*, vol. 4, San Antonio, TX, USA, Sep. 2007, pp. 41–44.
[26] G. K. Ouzounis and M. H. Wilkinson, "A parallel implementation of the dual-input max-tree algorithm for attribute filtering," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*, vol. 1. Rio de Janeiro, Brazil: Instituto Nacional de Pesquisas Espaciais (INPE), Oct. 2007, pp. 449–460.
[27] M. Wilkinson, Hui Gao, W. Hesselink, J.-E. Jonker, and A. Meijster, "Concurrent computation of attribute filters on shared memory parallel machines," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 10, pp. 1800–1813, Oct. 2008.
[28] P. Matas, E. Dokládalová, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev, "Parallel algorithm for concurrent computation of connected component tree," in *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, S. Bourennane, W. Philips, D. Popescu, and P. Scheunders, Eds., vol. 5259. Juan-les-Pins, France: Springer Berlin Heidelberg, Oct. 2008, pp. 230–241.
[29] M. Gotz, G. Cavallaro, T. Geraud, M. Book, and M. Riedel, "Parallel computation of component trees on distributed memory machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2582–2598, Nov. 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8360392/
[30] S. Gazagnes and M. H. Wilkinson, "Distributed connected component filtering and analysis in 2d and 3d tera-scale data sets," *IEEE Trans. Image Process.*, vol. 30, pp. 3664–3675, 2021.
[31] W. H. Hesselink, "Salembier's min-tree algorithm turned into breadth first search," *Information Processing Letters*, vol. 88, no. 5, pp. 225–229, 2003.
[32] M. H. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proceedings of the 18th International Conference of Image Processing (ICIP)*, 2011, pp. 1021–1024.
[33] E. Carlinet and T. Géraud, "A comparative review of component tree computation algorithms," *IEEE Trans. Image Process.*, vol. 23, no. 9, pp. 3885–3895, Sep. 2014.
[34] D. Menotti, L. Najman, and A. de Albuquerque Araújo, "1D component tree in linear time and space and its application to gray-level image multithresholding," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*. Rio de Janeiro, Brazil: Instituto Nacional de Pesquisas Espaciais (INPE), Oct. 2007, pp. 437–448.
[35] J. J. Kazemier, G. K. Ouzounis, and M. H. F. Wilkinson, "Connected morphological attribute filters on distributed memory parallel machines," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*, J. Angulo, S. Velasco-Forero, and F. Meyer, Eds., vol. 10225. Cham: Springer International Publishing, 2017, pp. 357–368, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-57240-6_29
[36] S. Gazagnes and M. H. F. Wilkinson, "Distributed component forests in 2-D: Hierarchical image representations suitable for tera-scale images," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 33, no. 11, p. FIXME, Oct. 2019. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218001419400123

[37] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, 1991, pp. 370–380.
[38] 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.
[39] 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.
[40] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1217–1230, 2018.
[41] A. Hennequin, L. Lacassagne, L. Cabaret, and Q. Meunier, "A new direct connected component labeling and analysis algorithms for GPUs," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2018, pp. 76–81.
[42] S. Allegretti, F. Bolelli, and C. Grana, "Optimized block-based algorithms to label connected components on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 423–438, 2019.
[43] F. Lemaitre, A. Hennequin, and L. Lacassagne, "Taming voting algorithms on GPUs for an efficient connected component analysis algorithm," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 7903–7907.
[44] U. Moschini, A. Meijster, and M. H. Wilkinson, "A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 3, pp. 513–526, 2017.
[45] G. K. Ouzounis and P. Soille, "The alpha-tree algorithm," *JRC Scientific and Policy Report*, 2012.
[46] L. Najman, J. Cousty, and B. Perret, "Playing with kruskal: algorithms for morphological trees in edge-weighted graphs," in *Proceedings of the International Symposium on Mathematical Morphology (ISMM)*. Springer, 2013, pp. 135–146.
[47] 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.

Nicolas Blin is currently pursuing an Ing. degree in image-processing at EPITA, Paris, France, and will graduate in 2022. He is also working as a research assistant at EPITA Research and Development Laboratory (LRDE). His research interests involve algorithms parallelization on GPU, mathematical morphology, and metaheuristic algorithms.

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.lrde.epita.fr/olena/pylene.

Florian Lemaitre Florian Lemaitre is a computer science postdoctoral researcher at Sorbonne Université. He works on the design and the optimization of new algorithms for micro-controllers, CPUs and GPUs. His expertise covers architectural code optimization, SIMD, multi-core, many-core, and GPU programming. He got an engineering degree in Computer Science at Ecole Supérieure d'Electricité (Supélec) in 2015 and a master degree in Computer Architecture at Université Paris-Sud. He received his doctorate in computer science from Sorbonne Université in 2019.

Lionel Lacassagne Lionel Lacassagne is a professor of computer science at Sorbonne University, formerly University Pierre et Marie Curie - Paris 6. His research focuses on the design of new algorithms for multi-core, many-core processors, and GPUs. He's working on high performance computing and image processing applied to embedded systems. He is an expert in code optimization for SIMD multi-core processors, GPUs, Cell processors, and VLIW DSP. Before his current post, he was an assistant professor at Paris-Sud University. He received his doctorate in robotics from Paris 6 University in 2000 and earned his engineering degree in computer science in 1995 at EPITA.

Thierry Géraud received the PhD degree in signal and image processing from Télécom Paris-Tech, in 1997, and the Habilitation à Diriger les Recherches from the Université Paris-Est, in 2012. He is one of the main authors of the Olena platform, dedicated to image processing and available as free software under the GPL licence. His research interests include image processing, pattern recognition, software engineering, and object-oriented scientific computing. He is currently working with EPITA Research and Development Laboratory (LRDE), Paris, France. He is a member of the IEEE.