HAL
open science

Fast and efficient FPGA implementation of connected operators

Nicolas Ngan, Eva Dokladalova, Mohamed Akil, Fançois Contou-Carrère

To cite this version:

Nicolas Ngan, Eva Dokladalova, Mohamed Akil, Fançois Contou-Carrère. Fast and efficient FPGA implementation of connected operators. Journal of Systems Architecture, 2011, 57 (8), pp.778-789. 10.1016/j.sysarc.2011.06.002 . hal-00682942

HAL Id: hal-00682942

https://hal.science/hal-00682942v1

Submitted on 21 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fast and efficient FPGA implementation of connected operators

N. Ngan [a,*] , E. Dokladalova [b] , M. Akil [b] , F. Contou-Carrère [a]

[a] *Sagem, Massy-Palaiseau, France*

[b] *Université Paris-Est, Unité mixte CNRS UMR-8049, Computer Science department, ESIEE Paris, France*

Abstract

The Connected Component Tree (CCT)-based operators play a central role in the development of new algorithms related to image processing applications such as pattern recognition, video-surveillance or motion extraction. The CCT construction, being a time consuming task (about 80% of the application time), these applications remain far-off mobile embedded systems. This paper presents its efficient FPGA implementation suited for embedded systems. Three main contributions are discussed: an efficient data structure proposal adapted to representing the CCT in embedded systems, a memory organization suitable for FPGA implementation by using on-chip memory and a customizable hardware accelerator architecture for CCT-based applications.

Key words: Image processing, connected component tree, connected filters, graph, hardware implementation, FPGA, embedded architecture.

1 Introduction

Nowadays, speaking about the design of modern architectures for computer vision systems automatically means addressing the performance and the reuse problem. Algorithms in computer vision systems are asked to furnish a high performance and be capable of supporting the entire processing chain: from low-level to high-level processing in various applications. This is a never-ending problem in hardware design. If the performance is achieved by an optimization effort which means high system specialization, it will (by definition) limit its flexibility.

* Corresponding authors: Nicolas Ngan, nicolas.ngan@sagem.com

Obviously, the source of the problem is computational disparity: an application needs a variety of mathematical and algorithmic domains. It involves respecting different computing models. To overcome this problem, many researchers investigate the domain of reconfigurable/adaptable computing in order to find the alternative to the performance/flexibility trade-off [15]. Nevertheless, they remain optimized, in the majority of cases, for low level image processing [29, 14]. Several generalized coarse-grained systems can be adapted to high level processing [16, 24].

In our research, we study if the unified formalism which uses the tools from computational geometry (connectivity, data adjacency and proximity, graph transformation, etc.) allows the algorithm disparity problem to be overcome. We focus on the applications using **graph theory** as they could allow to bridge the gap between low-level [38, 43] and high-level [8, 39] processing implementations. In addition, many useful application domains are organized around graphs: image processing, data and knowledge bases, CAD optimization, digital problems, simulations.

In this paper, we focus on image processing operators using the Connected Components Tree (CCT). CCT plays a central role in the development of new algorithms related to image processing problems [35]. From the practical point of view, the advantage of these methods is that once the CCT is constructed, the processing is performed on the tree by graph transformation(s), and only one data structure is used from low-level to high-level processing (Fig.1). In addition, the graph transformations are applicable to any dimension (1D, 2D, 3D, ...).

Note that these algorithms have been successively used for filtering [20, 38], motion extraction [38] and watershed segmentation [31]. Besides, numerous practical applications exploit CCT data representation: pattern recognition of astronomical images [17, 4], microscopic image analysis [10], video-surveillance applications [34], image registration [23], data visualisation [6].

Fig. 1. Three main phases of connected component tree-based application

On the other hand, time efficient **CCT construction** is a challenging problem. It consumes about 80% of the application execution time[1], which is penalizing for a lot of the mentioned practical applications. This is because the graph structures can be very large and, for each operation, the applications need to access a large, data-dependent portion of the graph. Several

[1] It is a mean estimation obtained by profiling of two CCT construction methods published by P. Salembier [38] and L. Najman and M. Couprie [32]

algorithms have been proposed in order to solve the problem. In the majority of cases, they remain sequential and algorithm improvement relies on fast data structures (fifo-like) [38] or on optimization of computational complexity [22, 6, 8, 5, 42].

The problem of efficient connected component labelling have attracted the attention of hardware designers for many past decades [1, 30, 18, 19, 13]. However, to our knowledge, no efficient hardware implementation has been proposed so far (up to recently in [33]) for connected component tree algorithms due to its complexity. If any parallelization effort has been made on shared-memory computers by [25, 43, 21], there are few studies targeting hardware acceleration of computing CCT. Among them, we should mention Associative nets [27] allowing direct CCT computation or multi-FPGA Step-Based Architecture [11] where the authors present a distributed computing system based on a general graph. Nevertheless, these solutions are not ready yet for mobile embedded applications.

This paper presents an efficient FPGA implementation of CCT based computing suited for embedded systems. The three main contributions of the proposed concept have been validated on an FPGA platform:

- Adaptation of the CCT **data structure** allowing efficient **data mapping** in memory
- Proposal of an **on-chip memory organization** suited for CCT processing
- Proposal of **a hardware accelerator architecture for an embedded system** for CCT based applications

The paper is organized as follows: Section 2 *Connected Component tree algorithms* presents the algorithmical issues of the CCT-based applications. Here, we also discuss the data structures used to represent CCT and the adaptation of CCT allowing efficient data mapping in the FPGA memory. We focus on *Computing tasks dependency* analysis in Section 3 followed by a proposal of an efficient *Hardware implementation* in Section 4. Finally, the *Experimental results* are summarized in Section 5.

2 Connected Component Tree algorithms

CCT operators work on flat zones (connected components) of images, rather than on individual pixels. We call a connected operator an operator merging only flat zones. Hence, it cannot introduce any new contour. It simplifies and preserves the contour information [36, 9]. In the context of segmentation techniques, such as region growing or watershed, they rely on iterative merging strategies [28]. The CCT construction requires a global image analysis in order

to respect the relationship of the adjacent flat zones.

2.1 Basic mathematical notions

Let us consider a 2D greyscale image $f : D \to \mathbb{Z}$, $D = \mathbb{Z}^2$ and suppose that D is equipped with the neighbourhood mapping $N : D \to \mathscr{P}(D)$. $\mathscr{P}$ denotes the power set (the set of subsets). Then, $\forall x, y \in D$, we say that x and y are connected if $x \in N(y)$. Hereafter, let us assume that N is the 4-neighbourhood N_4, i.e. the set of the north, south, east and west neighbours. The associated connectivity is therefore the 4-connectivity.

For some $A \subset D$, we say that A is a connected set if $\forall a, b \in A$ there is a sequence of points $(a = x_1, ..., x_n = b) \subset A$, such that $x_{i+1} \in N_4(x_i)$, $1 < i < n$. For some A, $CC(A) \subset \mathscr{P}(A)$ shall denote the set of connected subsets of A.

For some $k \in \mathbb{Z}$, the set $C^k = \{x \mid f(x) \geq k\}$ is the set of points above the threshold k. The set $\{c_i^k\} = CC(C^k)$ denotes the set of connected components above k. For some $k, l \in \mathbb{Z}$, and $k > l$, it holds that $C^k \subseteq C^l$. The set of all connected components is called $C = \cup_k C^k$.

Let us consider a graph $G = G(V, E)$ where V, such as $V \to C$, is the set of nodes and $E \subset V \times V$ the set of edges. The nodes are tied by the relation child-parent. For some $v_i, v_j \in V$, there is an edge $e_{ij} \in E$ if $c_i \subset c_j$ and there is no c_k that $c_i \subset c_k \subset c_j$. We say that v_j is the parent of v_i and v_i the child of v_j. Notice that the graph is oriented, i.e. $e_{ij} \neq e_{ji}$.

Note that such a graph is an acyclic, oriented graph called **connected component tree**. There are special nodes called *leaves* and *roots*. The leaves do not have any children and the root does not have any parents.

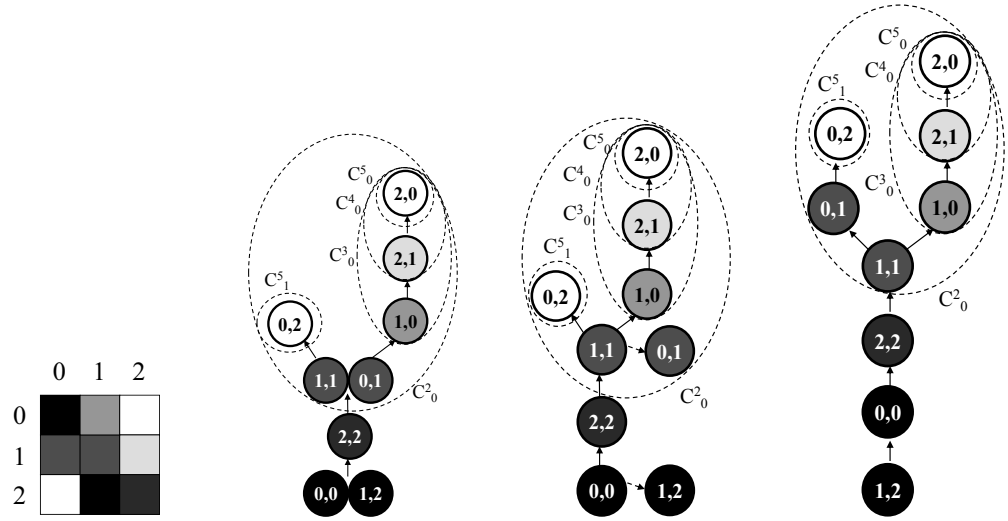
Due to the inclusion relation on the connected components, the graph nodes are organized in a tree structure suitable for filtering or segmentation [40].

2.2 CCT representations

To represent the CCT structure in memory, different data organization leading to different memory requirements, have been proposed : i) **"Native" CCT** (Figure 2(b)) [38] or ii) its "compressed" version **Canonical CCT** (Figure 2(c)) where one node point represents a component and each pixel belonging to a component points toward its representative node [32]; or **Point Tree** (Figure 2(d)), the burst version of the CCT where each node of a point-tree

represents a pixel in the picture [5]. 1-D tree computation is a special case [26], where intermediate results have to be gradually merged as described in [43].

According to the definition of c_i^k, some set of connected components are shown in Figure 2 for different tree structures. In this figure, connected components from level two to five are shown (encircled) as an example picture with six grey levels.



(a) Input image (b) Native CCT (c) Canonical CCT (d) Point Tree

Fig. 2. Illustration of different data structures

The reader can refer to Table 1 for evaluation of memory requirements. Based on the algorithm definitions, the memory requirements are roughly estimated as the sum of memory allocated for input/output and the overall working memory needed for intermediate results. Those estimations have been discussed in [21, 4]. Table 1 also contains the estimation of algorithm complexity. For the sake of clarity, we consider the classical $\mathscr{O}$ notation, where the worst-case scenario is given for a considered algorithm. Data type column means the data type for an input image pixel coding allowed by the algorithm.

Table 1
Comparison of algorithms: n is the total number of pixels in the image, G is number of grey levels of the image, α is a very slow-growing diagonal inverse of Ackermanns function [8].

Tree type	Complexity	Data type	Memory requirements
Native CCT [38]	$\mathscr{O}(nG)$	int	$4n$
Canonical CCT [32]	$\mathscr{O}(n\alpha(n))$	int, float	$6n$
Point tree [5]	$\mathscr{O}(n\log(n))$	int, float	$4n$
1-D [26]	$\mathscr{O}(n)$	int, float	n

2.3 Data structure adaptation: Parent Point-Tree

The data structures proposed in the above mentioned theoretical papers has not been designed for an efficient FPGA implementation. Neither is the number of nodes nor the size of each node, known beforehand and one has to manage complex memory allocations. This difficulty is evident in the case of native CCT structure or canonical CCT since the number and the size of the nodes depend on the image content.

From this point of view, the Point Tree (PT) presented in [5] seems to be an appropriate structure due to a fixed number of nodes. Originally, PT is an oriented graph where the parent node points toward the child node (Figure 2(d)), note that this is a common orientation of the discussed structures (Figure 2). In such a case, the issue is that the number of child nodes for each parent node can only be known at the end of the tree construction. Thus, the number of link memory allocations for each node is not predictable. As shown in Figure 3, we propose to reverse the PT orientation and we call it **Parent Point Tree (PPT)**. Thus, each node has at most, a unique parent node. Consequently, we only need to store the parent address for each child node to create tree connections and the number of nodes is equal to the number of pixels in the image.

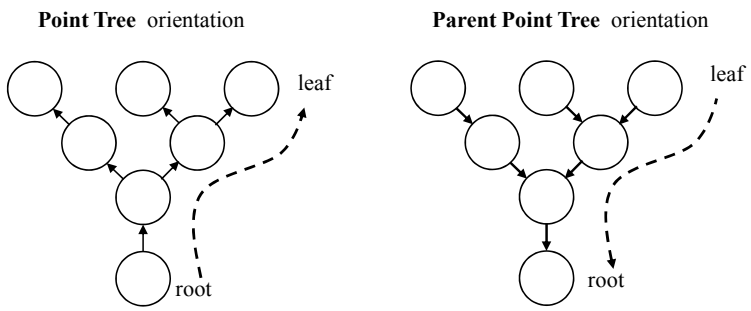


Fig. 3. Parent Point Tree orientation

The next section presents the PPT construction algorithm followed by its FPGA implementation.

2.3.1 Parent Point Tree construction algorithm

The proposed Parent Point Tree construction algorithm (Alg. 1) is a combination of Tarjan's Union-Find [40] as used in [5] with an acceleration technique inspired by the Najman-Couprie algorithm [8].

We define a *branch* as a subset representing a group of nodes that are connected to each other in the complete tree. A *branch* can then be composed of only one node or several nodes. A node which represents the extremity of a

branch (i.e. a node without any child nodes in the tree) is called a *leaf*. As the tree is gradually built with leaves, we call *local root*, the temporary root node for each branch that is not yet connected to the main tree.

We recall that in a *Point Tree* or a *Parent Point Tree*, a node is associated to only one pixel in the picture. Then, let us consider p a pixel of the image f, and q an adjacent pixel of p according to the considered neighbourhood $N_4(p)$ such as $q \in N_4(p)$.

In this algorithm, the tree is gradually built with leaves to the root. Consequently, it starts by building several *branches* that are not necessarily connected to each other. They become a partial tree which then contains several *local roots*.

The tree construction is divided into two parts: the first part involves the sorting of all the pixels in the image in increasing or decreasing order, depending on the type of tree. In order to build a *Max-Tree*, pixels are sorted in decreasing order and the leaves then represent the highest pixel values. The second part is building the tree by processing those pixels sequentially in this order. The algorithm presented in Alg. 1 corresponds to that second part, assuming a correct sorting of pixels p in a *queue* memory. The most costly part of the algorithm is to find the local root in order to link the components. At each neighbour node, it needs to explore the tree until it finds the root.

As described in Alg. 1, each pixel q from the neighbourhood N_4 of a pixel p is processed. The first step is to check its processed status. If it has already been processed from the pixel queue, it means that this pixel already belongs to a branch from partial tree (*tree_links_parent*). The aim is to find the local root of the branch where the neighbour pixel q belongs, to link the current pixel p to it (function *Link*). The following procedure is to jump from node to node in the branch (function *Jump*), starting from pixel q, and check successively if the node is the local root of the branch. As a consequence, this loop is very time consuming and depends on the branch length.

Thus, we propose to accelerate this process by gradually linking the current local root during the construction for each node in a fake and temporary tree so that the next access would point directly to it. We then create a compression of the tree. However, this technique is costly in terms of memory because we need to allocate a "mirror copy" of the original *tree_links_parent* tree. This copy called *tree_links_root* is stored in a memory only dedicated to the tree construction and it can be modified at will for accelerating the construction.

This shortcut technique works mainly in the *tree_links_root* memory by first checking the last registered local root node associated to the pixel q (function *Check*). If this registered node is not a real root node, then we have to jump from node to node starting from this last registered node in the compressed

tree to find the root. During that searching process, encountered nodes are stored in a temporary buffer (function *Store*) to be all updated by linking them to the last root node found (function *Update*). This update then accelerates the searching process for future node access.

When any pixel is linked to its local root, we mark it as *processed*.

The following operations are necessary to build the PPT:
Check(x,T) returns the local root of the node associated with pixel x in the tree T,
Jump(x,T) returns the parent node of the node associated with pixel x in the tree T,
Store(x) stores the current parent node associated with pixel x to be updated in buffer,
Link(x,y,T) links the current node associated with pixel x to the local root node for pixel y in the tree T,
Update(x,T) updates the local root node x for all the parent nodes in buffer for the tree T.

Algorithm 1 Parent Point Tree construction

Require: *queue*: pixels ordered in decreasing order
Ensure: Root-Stored PPT

```
while (queue not empty) do
  for all (q ∈ N4(p)) do
    if (q == processed) then
      root ← Check(q, tree_links_root)
      while (root not found) do
        parent_node ← Jump(q, tree_links_root)
        q ← parent_node
        Store(parent_node)
        root ← Check(q, tree_links_root)
      end while
      Link(p, root, tree_links_parent)
      Update(root, tree_links_root)
    end if
  end for
  p <= processed
end while
```

Note that the construction algorithm is sequential and its execution is data-dependent. The more complex in terms of components the image is, the longer the algorithm execution will be. A noisy picture for example, produced directly from an image sensor is represented by a more complex Point Tree in terms of number of leaves because of numerous un-representative high pixel values caused by the noise. Then, one possible solution is to pre-process the picture (low-pass filters such as denoising, smoothing filters) to remove those unuseful

small high values.

2.4 CCT filtering

In the Introduction, we have presented the general scheme of the CCT-based application (Fig. 1). Once a tree construction is completed, the filtering step is performed. In practice, the filtering step analysis each node and it evaluates some criteria [38]. It makes a decision whether the node is preserved or removed.

In this paper, an attribute denotes supplementary information associated with each node allowing to measure a given criterion. We can quote for instance some classical criteria: area, height [12], opening by reconstruction or λ-max operators [41] and other examples can be found in [36].

When the complete CCT is available, with associated node attributes, the filtering can begin. This step is also called CCT pruning [37], the branches of the tree structure are removed or preserved according to the *decision rule*. In this paper, we use the *direct decision rule* as defined in [38].

Let us consider the maxima of the image represented by the leaves of the CCT and some parameter λ. Starting from the leaves, we scan all the sequence of their ancestors going down to the root. The examined node is only preserved when the associated attribute value is higher than λ. Otherwise, the node is removed. When one node is removed, its content is merged with its nearest preserved ancestor.

3 Computing task dependency

As illustrated in Figure 4, we can see that there are four steps to complete an application. It starts with pixel sorting. The tree construction can only begin when all image pixels are sorted. In general, attribute computation can be done parallel (AC1) to the tree construction. Hence, the pixel values accessed for tree construction can be immediately reused for attribute computation. It allows to minimize memory accesses and to parallelize the computing tasks. If it cannot be done in parallel, it has to be executed after the complete tree construction (AC2). Only after that, the tree filtering can be done.

We might want to start the tree filtering process parallel to the tree construction by using attributes (i.e. height) that can be computed along with the construction. It is actually feasible when a pixel node has been identified as a leaf of the tree. The partial branch is read and could be processed step by step

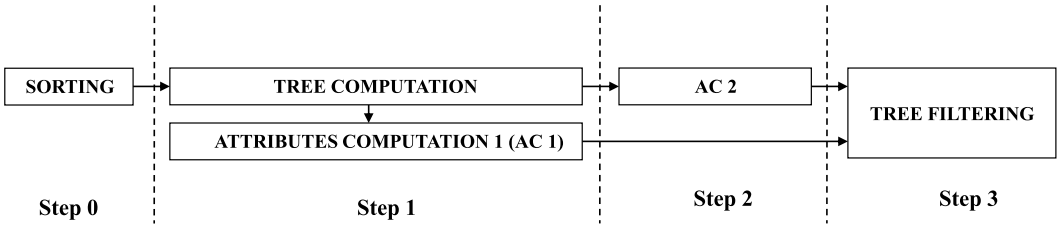


Fig. 4. Execution steps of a complete tree-based application

during its construction. However, it requires switching between branches during the tree construction as they are built gradually starting with the leaves. It obviously results in higher memory requirements to stand by the filtering process because branches are not always complete. As a consequence, it is more efficient to start the filtering process after the complete tree construction and the attributes computation as depicted in Figure 4.

4 Architecture proposal for embedded system

The global architecture (see Figure 5) of the system reflects the four presented execution steps. Hence, it consists of four main computing blocks: pixel sorting block, tree construction block, attribute computing block and filtering block.

For readability reasons, we indicate the address datapath by the symbol @ in Figure 5.

The main controller supervises an application execution and it manages the data flow between the computing blocks and the memory system. Finally, the computing blocks are interconnected by the switch fabric allowing the memory address and data paths to be redirected dynamically.

As shown in Figure 5, the global control is provided by the main controller which is composed of three command units : a block selector (which can activate or deactivate a computing block reporting its status -busy or not), a multi-switch controller (which can dynamically modify the Switch Fabric containing several mutiplexers) and a sequencer (which orders computing block activations and datapath modifications).

Thus, each computing block receives commands from the main controller. It contains an intelligent core: a controller that sequences local computing block operations.

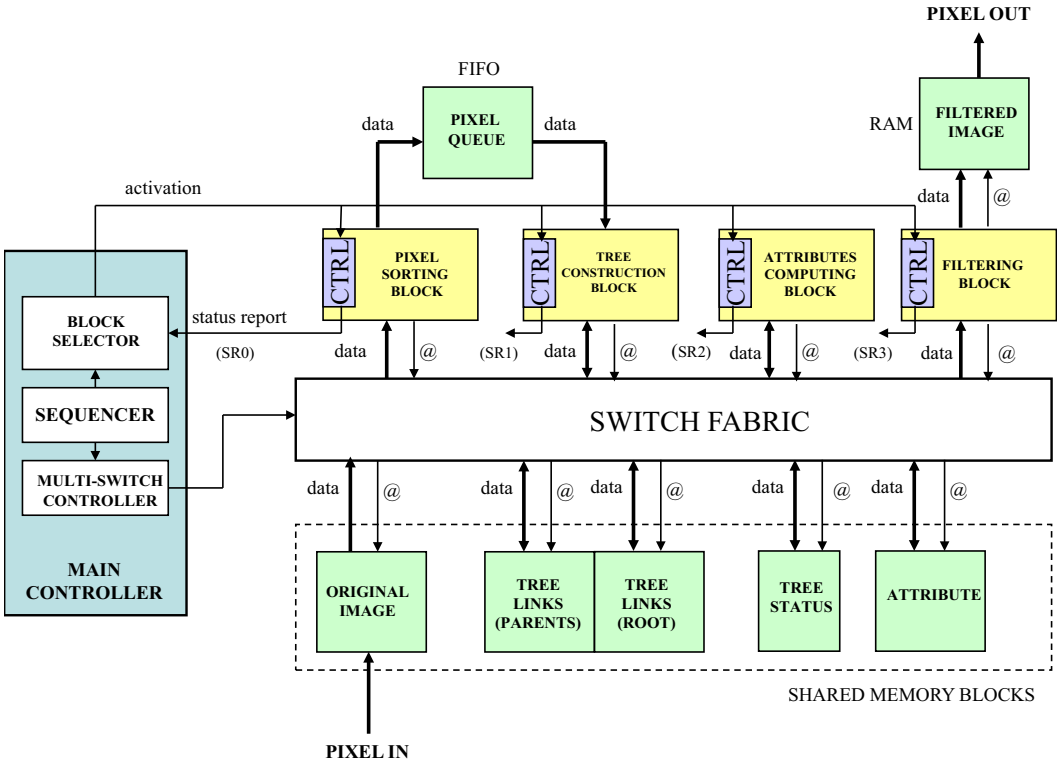


Fig. 5. Global architecture

4.1 Switch Fabric

In order to allow the memory address and data paths to be redirected dynamically, all the blocks share the memory and are plugged into the backbone of this architecture. The switch fabric is composed of selectors like multiplexer and demultiplexer basic components which are controlled by a multi-switch controller inside the main controller. It is therefore configurable before implementation and programmable after synthesis. The Switch Fabric is configured to the correct data path dynamically. Figure 6 shows one configuration example at runtime. The dashed lines represent data paths that are disabled at a specific time in the application.

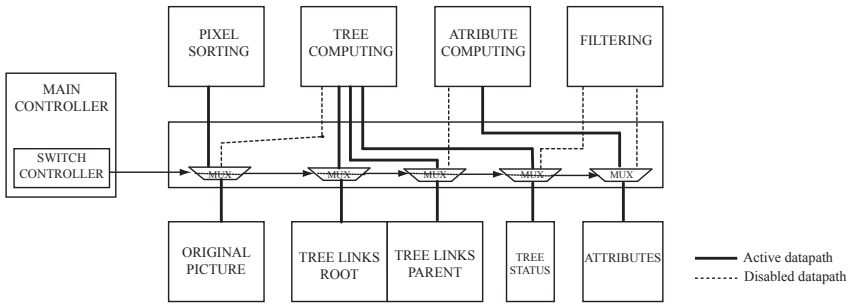


Fig. 6. Switch Fabric

4.2 Memory system

A CCT application works with six main types of data: original and filtered image, pixel queue, CCT, attributes and construction buffers needed during the processing (the tree status for instance). The designed memory system is based on the principle of shared memory constituted by several small independent memory blocks that can be accessed in parallel due to the Switch Fabric. By combinating those memory blocks to the Switch Fabric, we are able to reuse the memory blocks dynamically. However, we have added a dedicated memory for the pixel queue and the output image to buffer pixels more efficiently. We recall that the tree construction time is not deterministic and depends on the image content (Section 2.3.1).

At the beginning of the processing, the filtered image memory section is the exact copy of the original. The filtering is based on the built component tree. Depending on the given application (a segmentation, for instance with pixel extractions or deletions), we only modify the filtered image memory in the desired pixel position. A pixel queue memory (or pixel address queue) is a simple FIFO structure storing the output of the pixel sorting block. In fact, this memory has to store the different pixel positions in increasing or decreasing order.

The input image, CCT, attribute and construction buffers are placed in the shared memory. The original picture memory has to be shared with the other blocks because it contains the greyscale pixel value information contrary to the filtered image memory which is reserved to the filtering block. CCT is constructed in the link memory storing only the child/parent node relationships and the original picture is necessary for navigating between different components in the tree. Then, we have a part of memory dedicated to storing the different attributes (the height, the volume, etc.) as the output of the attribute computing block. During all the processing, some information might need to be buffered like the pixel positions for pruning the tree (the filtering process) by modifying the value or additional buffered addresses to eventually accelerate the processing. Finally, we need to temporarily store the status of each pixel node during the tree construction. We can call it "Tree status memory" and it contains bit flags which are detailed in Section 4.4.

4.2.1 Memory size

Let us consider an image with a width of M and a height of N. The pixel size is fixed to k bits. $ADDR_SIZE$ is defined as equal to $ceil(log2(M) + log2(N))$ ($ceil()$ function rounds to the next largest integer). The memory sizing is presented in Table 2.

Table 2
Memory size (qSIF = 160x120 test image format); H=15 (height attribute); k=8

Storage element	Memory space (bits)	qSIF Application (bits)
Original image	$(2^{ADDR_SIZE}) \times k$	262144
Filtered image	$(2^{ADDR_SIZE}) \times k$	262144
Pixel queue	$ADDR_SIZE \times M \times N$	288000
Tree links 1st part (parents)	$2^{ADDR_SIZE} \times ADDR_SIZE$	491520
Tree links 2nd part (roots)	$2^{ADDR_SIZE} \times ADDR_SIZE$	491520
Tree status	$(2^{ADDR_SIZE}) \times 3$	98304
Attributes	$(2^{ADDR_SIZE}) \times H$	491520

4.3 Pixel sorting block

The pixel sorting block operates on the original picture and fills a pixel queue for the tree construction block. This queue can be considered as a FIFO. The tree construction block computes the different links of the tree and stores them in a memory (tree link memory) which is shared with the other blocks (attribute computing block and filtering block in particular).

Let us take a simple example like a 3x3 picture with five levels of grey. For readability reasons in the following figures, each pixel is designated by a letter instead of its coordinates as shown in Figure 7(a).The corresponding parent point-tree of this example is presented in Figure 7(b).

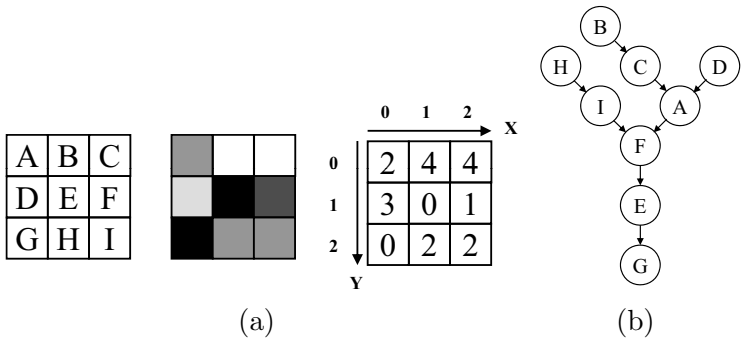
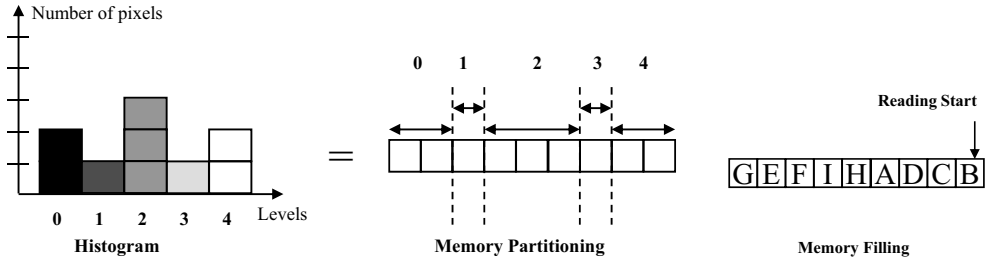


Fig. 7. 3x3 picture example

As mentioned in the previous section, pixel sorting is not the "time consumer" procedure compared to the creation of the connections between nodes (pixels in the parent point-tree). However, it can create high memory requirements. To avoid high consumption of memory blocks, we choose to use a counting sort method [7]. This technique is divided into two passes: the queue partitioning pass and the queue filling pass. In the first pass, we want to partition the queue memory by level sector and each size of the sector can easily be determined

by building a histogram (Figure 8(a)).



(a) First pass (b) Second pass

Fig. 8. Counting sort method

In our example, we can see that the number of pixels in each value is equal to the number of slots reserved in the queue memory. This pass can be done "on the fly" during the picture storage in the image memory. In the following pass, the picture is read from the image memory and the pixels are distributed in the queue memory according to their value.

In each level, the pixels are ordered according to the direction of reading. This order has an impact on the final parent point tree connections between the pixel nodes but it does not change the structure of the corresponding component tree when the pixels are grouped by component (See set c_i^k in Figure 2). In our example (Figure 8(b)), the reading of the queue starts with pixel B when we want to build a Max-Tree (decreasing pixel value order) or with pixel G for a Min-Tree (increasing pixel value order).

4.4 Tree construction block

The tree construction block implements the algorithm 1 presented in Section 2. As illustrated in Figure 9, this computing block contains five main processing units that can be associated to specific functions in the algorithm. The functions *Check(x,T)* and *Jump(x,T)* can be associated to the *Root Finder* unit returning the parent or root nodes. This unit works with the *Bit checking* unit by testing the status bits, corresponding to the if and *while* conditions in the algorithm. The function *Link(x,y,T)* is associated to the *Linker* unit by writing the origin pixel considered as the new local root in the *tree link parent* memory. The *Store(x)* and *Update(x,T)* functions corresponds to the *Root Updater* unit storing nodes in a temporary buffer to update them in the *tree links root* memory. Finally, the *Neighbour Addresses Generator* unit computes the adjacent pixels $q \in N_4(p)$ memory addresses.

The pixel sorting block provides the input data *queue* through the pixel queue containing all the pixel addresses ordered with respect to the decreasing pixel value. As shown in Figure 9, the neighbour address is verified with the Bit

Checking and the Root Finder units to get the status of the pixel node and to eventually get any parent address. If the bit status for the current pixel neighbour is *processed*, the Bit Checking unit notifies the Root Finder unit to loop. The parent address is then loaded to the current neighbour address register in order to get its parent pixel. This parent adress is also stored in the FIFO Buffer of the Root Updater unit (= *Store(x)* function). The checking loop ends when the bit status for the parent presence is *un-processed*. It means that we reach the current root in our tree under construction. Consequently, the Linker unit can update the identified current root by the origin pixel address (= *Link(x,y,T)* function). The origin pixel node becomes the new pixel parent node and its status bits are updated. All buffered pixel addresses that have been met, are updated to be linked to the origin pixel node (= *Update(x,T)* function). Thus, that node becomes the latest local root and the tree construction block can request the following origin address from the pixel queue (*queue*).

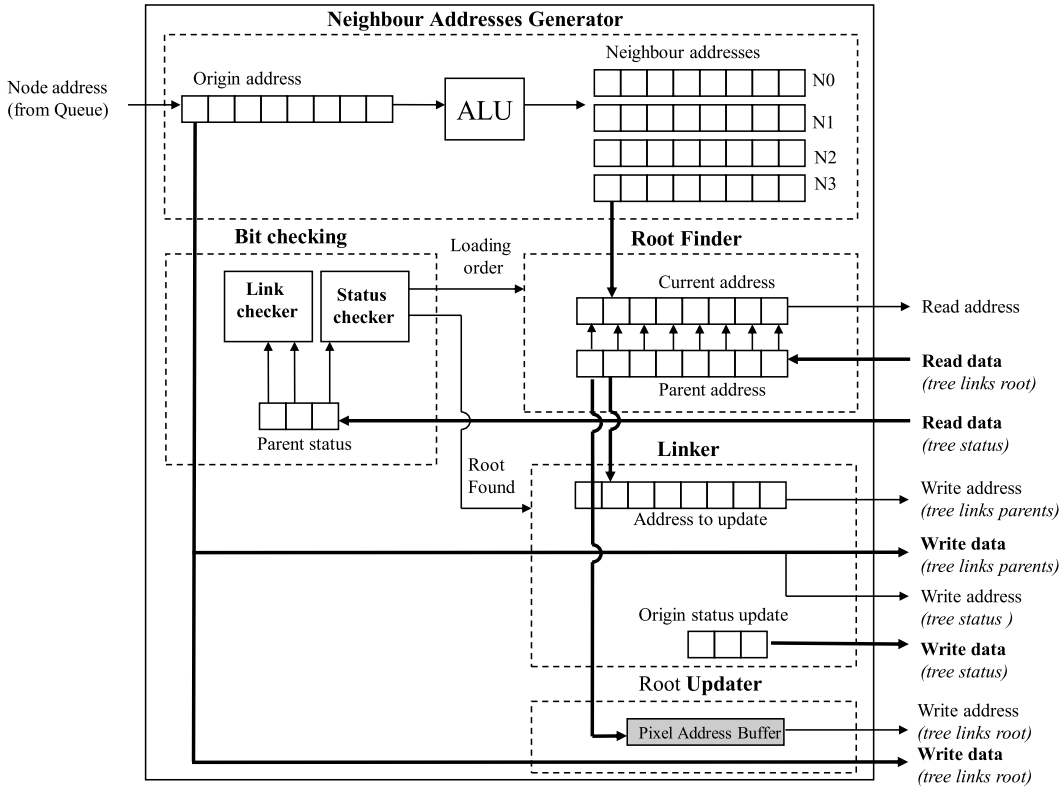


Fig. 9. Tree construction block

Tree construction example: Each pixel node has three bit flags: "Is Processed" (IP) bit to check if the pixel node has been processed, the Parent Checking (PC) bit, meaning that the pixel node has a parent node and the Child Checking (CC) bit, meaning that the pixel node has at least one child node. Let us begin with the pixel B whose coordinates are (1, 0) in Figure 7(a).

This pixel is located on the edge of the picture and its neighbours $N_4(B)$ are C, E and A. According to the pixel coordinates, nonexistent pixel neighbours (represented by XXXX in Figure 10) are just skipped from analysis in the Root Finder knowing the image dimension.

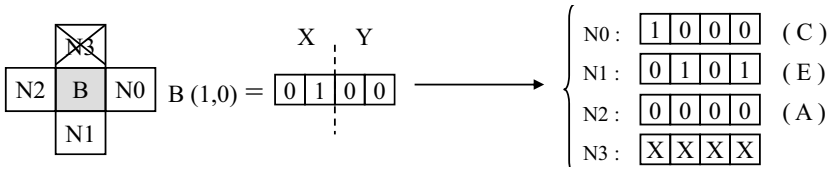


Fig. 10. Pixel B neighbours

At the beginning, none of the pixels are processed and all the "*processed*" - IP bit flags in the tree status table are down. Consequently, all pixel B neighbours are not processed yet. Pixel B becomes a leaf of the tree and is temporarily linked to itself meaning that it is a current root of the tree in construction. Then, the IP bit flag for pixel B is up (Figure 12(a)) and the tree construction block can request the following origin pixel in the queue which is pixel C in our example.

When pixel F is reached (Figure 11), some pixels have already been processed (B, C, D, A, H, and I - See pixel queue in Figure 8(b)) and they might be linked to this pixel. Thus, we are only interested in processing the neighbour N1 (I) and N3 (C) (Figure 11)

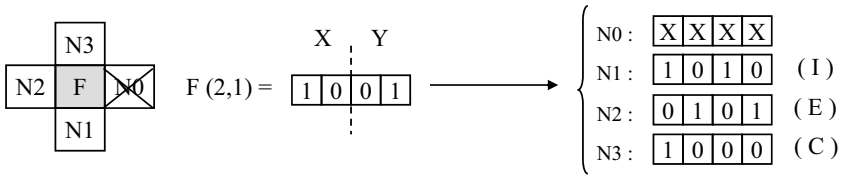
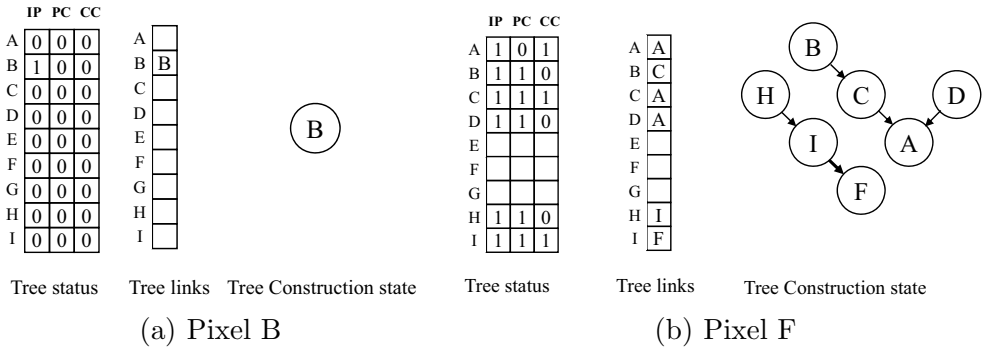


Fig. 11. Pixel F neighbours

Pixel I is already a current root and it is directly linked to pixel F by updating the tree status table (the CC bit to 1) and the tree link table (F address for pixel I).

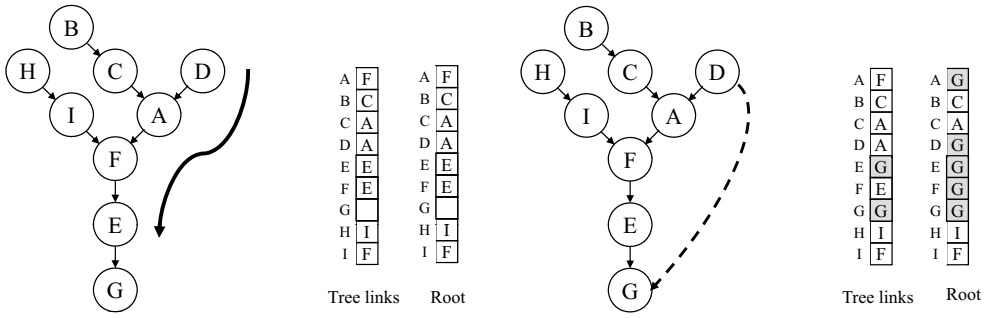
Pixel C is not a current root because its PC bit flag is up. The Root Finder unit in the tree construction block replaces the current address (pixel C) by its direct parent address (pixel A). Pixel A is a current root because its PC bit flag is down (Figure 12(b)). After linking it to pixel F, its tree status bits will be updated from [1 0 1] to [1 1 1].

Shortcut technique example: Let us illustrate the shortcut technique by a simple example. From the Figure 13(a), let us assume that we want to link the G node in the tree beginning with its neighbour D node. The current root E is therefore linked to itself for the time being.



(a) Pixel B (b) Pixel F

Fig. 12. Tree tables



(a) Before linking (b) After linking

Fig. 13. Shortcut technique example

The tree is explored starting with the D node and the nodes A, F and E are met before linking. When the "finding root process" is triggered, all the nodes that have been met during the exploration, are linked to the pixel point to be attached. In other words, the origin pixel will be the current root of the tree branch. If we consider that this root becomes the parent of all the pixel nodes that have been met, the following accesses would be faster by making large node jumps. In our example, we can store the nodes A, F and E in a temporary memory and update their "virtual" parent node G in the root memory as shown in Figure 13(b). Note that the exploring accesses are consequently made on the root memory.

4.5 Attribute computing block

As mentioned in section 2.4, attributes are additional node characteristics that can be used for filtering an image depending on user criteria (conditions on height, area, depth or volume of a component for example). Some of them can be computed during the tree construction and others have to be computed when the tree is complete. The depth of a node can be computed, for instance, on-the-fly with a counter and additional logics (min-max) during the tree construction.

Depending on the complexity of required attributes, the computing block might be time consuming. The study of attributes is beyond the scope of this paper and one can find some attribute computations in [25, 32].

Note that an attribute computing block is not mandatory to complete an application. In order to save memory and time, the simplest implementation can be done without attributes but the criteria have to be chosen so that they are merely based on components and pixel values. Some basic criteria could amount to conditions on the contrast between two components by subtracting consecutive component levels. Nevertheless, computing attributes allows to apply more complex criteria on the tree for better results in the filtering process.

4.6 Filtering block

The component tree can be used as a new working base for filtering the associated picture. Thus, pixels or components of the picture can be manipulated by changing the value or the node positions in the tree. The component tree is particularly interesting compared to the classic 2-D matrix picture representation because it gives the hierarchical layer disposition, in addition to the spatial pixel position. Traditionally, a greyscale picture can be viewed as a set of hills (highest values) and valleys (lowest values) with a layer organisation. Flattening hills means removing the highest layers and leaving the last implied underlayer apparent.

In a tree-based application, a picture is usually simplified by pruning the corresponding component tree. Back in Figure 2, pruning the tree at C_0^3 is removing the upper nodes (2,0) and (2,1) to leave the underlayer value apparent (level 3). Thus, pruning the component tree can be understood as flattening hills. Note that a tree pruning application can be found in [4] and [5] for an astronomical context where the aim is to clean the sky from stars in greyscale pictures.

In our context, we propose an original tree-based filtering application based on the component tree pruning. Instead of merging components (i.e. flattening hills) with a traditional tree pruning process, we propose to give an arbitrary value for all the identified nodes to be removed from the tree. Thus, it results in an original local thresholding by using the component tree. It can be efficient for simplifying pictures from an infrared camera because the most interesting parts are the highest local values in the image. Those local parts can have different levels of grey and using an arbitrary global level threshold would be obviously inefficient.

Consequently, a specific tree pruning block has been designed for this purpose.

Flattening the hills of a greyscale picture can be done technically by replacing the removed pixel value nodes with the pixel value of their parent nodes. In our case, we replace any removed pixel value nodes with an arbitrary value.

A tree pruning process consists in exploring the tree from each leaf to the root in order to cut down branches when proposed criteria are not respected as explained in section 2.4. Those criteria are based on previously computed attributes (height, volume, depth, etc.) and by combining them, we can obtain original results. One can find some examples in [25].

The leaves of the tree are the starting points for the filtering process. They can be identified with the *Child Checking* (CC) bit from the tree status table. We recall that a CC bit flag down means that the pixel node does not have a child node. Notice that the reading direction of the tree, from its tops to its root, is well suited considering the proposed tree link structure (a parent-tree where child nodes point at its parent node) and the targeted application (local thresholding). Since we know that we need the leaves, they can be stored during the tree construction in a buffer (i.e. a FIFO for instance) to feed the tree pruning block. In particular, we can note that, during the tree construction, a current processed origin pixel node, which is not linked to any other nodes, becomes a leaf node.

The tree pruning block architecture is presented in Figure 14 with two criteria in input. It contains three main processing units: a *Tree explorer* unit, an *Attribute collector* unit and an *Image updater* unit.

Notice that, in addition to the *tree links parent* table, the block has to refer to the original image to identify the component values.

As illustrated in Figure 14, each leaf node address is loaded in the *Tree explorer* unit. This unit works parallel to the *Attribute collector* unit which reads successively the associated attributes for each processed node. The *Tree explorer* unit explores each tree branch by several reading loops in the *tree links parent* table until one of the criteria is not valid according the comparison units.

When the criteria are respected, the pixel node is stored in a temporary "Zone FIFO" buffer located in the *Image Updater* unit, until the end of a tree branch exploration. When one of the criteria is not respected, the branch is modified by overwriting the pixel value of the nodes stored in "Zone FIFO" with a specific value. Note that the use of this FIFO is optional in our case because we impose a value. Thus, the overwriting can be done gradually. The FIFO is only necessary for a tree pruning block that requires to merge components.

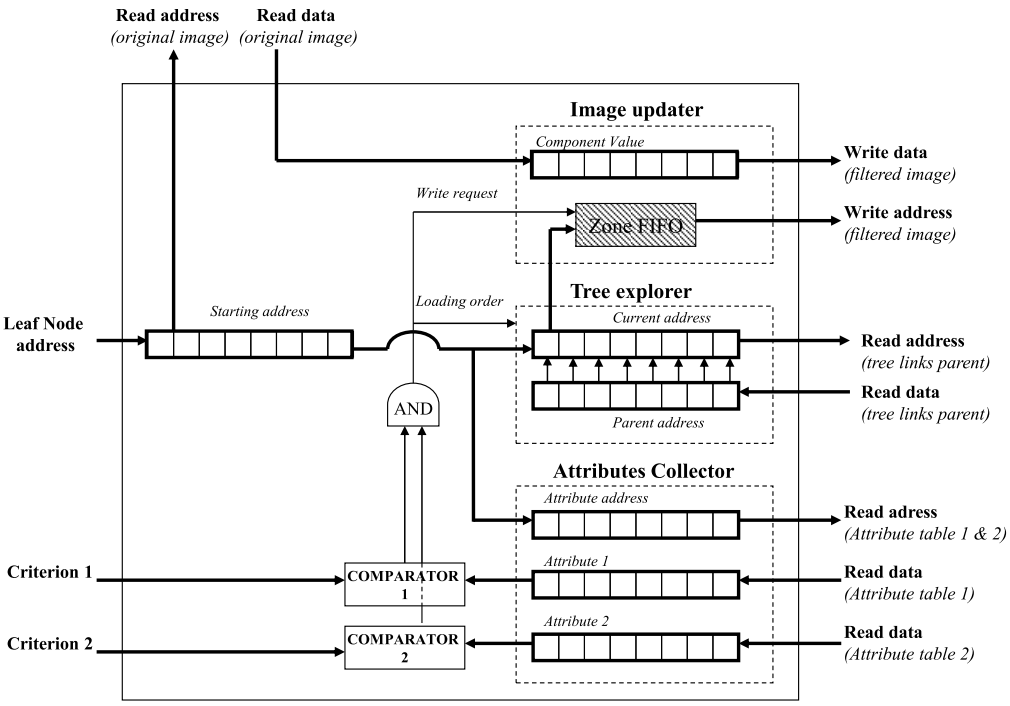


Fig. 14. Tree pruning block

5 Experimental results

The proposed generic system architecture has been implemented and validated on ALTERA Stratix II 2S60 Development board [2]. To the best of our knowledge, there is no other FPGA implementation of a complete application based on tree construction. For our hardware implementation, we propose a simple infrared (IR) hot spot image filtering application based on pruning the tree. To maximize the memory bandwidth, we choose to use the on-chip memory directly available on the FPGA chip. These RAM Memory blocks are very limited resources and in our case, we had a budget of 2,544,192 On-chip RAM bits for the 2S60 version. Moreover, Stratix II devices features a particular memory system called TriMatrix based on three sizes of embedded RAM blocks (M512, M4K and M-RAM) which can be configured in different addressing structures. Further information can be found in [3].

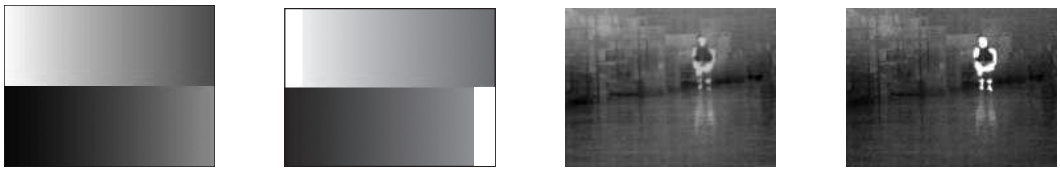
In particular, we used M4K (4 Kbits) and M-RAM (512 Kbits) memory blocks to fit our memory system and we used an input image resolution of 160x120 coded with 8 bits pixels. Thus, the different tables (Tree Links and Tree Roots) must have 15-bit-wide data words and the Tree Status table has 3-bit-wide data words (IP, PC and CC bits). Additional memory is needed for the pixel address queue, the two images (one for keeping the original picture and the other for storing the filtered picture) and several buffers (Root, Zone and Leaf

FIFOs).

The FIFOs (Root, Leaf and Zone) size is defined according to the free memory space left. Smaller FIFOs decrease the overall performance. Note that it is possible to share the same FIFO for the Root FIFO and the Zone FIFO because they are not used at the same time. It is also possible to decrease the number of level counters by only taking the four most significant bits of the input pixel value.

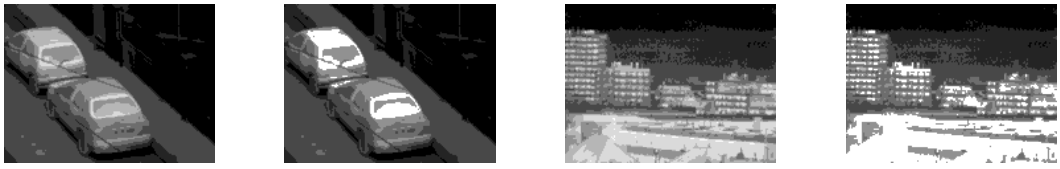
One can notice that the Attribute Table is not really necessary for our first implementation. The pruning can simply be tested on the basis of pixel values and specified by an arbitrary threshold. Our focus is on the tree processing time and we want to validate our architecture conception.

Four types of pictures have been selected for the test (Fig. 15 and 16): a gradient picture with smooth grey level transitions (Fig. 15(a)) and three infrared (IR) image camera outputs.



(a) Gradient test (b) Gradient test result (c) Man IR test (d) Man IR test result

Fig. 15. Test images



(a) Cars IR test (b) Cars IR test result (c) Rooftop IR test (d) Rooftop IR test result

Fig. 16. Test images 2

Those IR pictures represent real scenes (man, cars, rooftop) with the different heat levels detected by the sensor. The man IR picture (Fig. 15(c)) represents the direct output from the image sensor and is a very noisy picture. The two others (Fig. 16(a) and 16(c)) represent pre-processed pictures. Those last two pictures, showing cars and rooftops, have been denoised by a median filter and the contrast has been enhanced by a histogram equalization. As a consequence, those conditions have a direct impact on the number of connected components to process and the tree structure complexity as shown in Table 3. In this table, the man IR picture is the most complex scene to process because of the number of connected components resulting from the high noise in the image sensor.

The results (Fig. 15(b), 15(d), 16(b) and 16(d)) represent the detection of the highest local components in the test pictures by combining attributes. In this case, we use an arbitrary max pixel value threshold λ applied to two attributes : contrast (component intensity difference) and height (number of pixel value component layers). It then creates a satisfying local thresholding for our prototype.

Table 3
Characteristics of test images

Image	Size	Number of connected components
Man	160×120	3068
Gradient	160×120	353
Cars	160×120	84
Rooftop	160×120	362

As shown in the pie (Figure 17), the architecture uses 78 % of memory space available and only 30 % represents the tree information (Original image and Tree Links). The remaining memory is dedicated to compute the tree, to accelerate the construction and to buffer intermediate data. Table 4 contains the results in terms of resource utilization obtained after the synthesis on the Stratix II FPGA. The logic utilization is very small as all the work is based on memory management, extremely important for the FPGA implementation. Note that our implementation has a maximum frequency of about 100 MHz for an Altera Stratix II FPGA EP2S60F484C4 target [2]. That maximum frequency is limited by the logic elements required to make the Switch Fabric. For the FPGA prototype, the system runs at 50 MHz as shown in Table 5 for measuring timing performance.

The 160×120 images are low resolution and come from a 320×240 IR image input downscaled by 2. That resolution is acceptable as it aims at low resolution display screens embedded in portable systems. The global architecture in Figure 5 shows that the memory, required for computing a picture, scales linearly according to its resolution. Thus, by keeping the same algorithm and the same architecture, there are only two options for a higher image resolution implementation. Firstly, to keep the memory performance, the simplest way is to upgrade the FPGA target for a higher on-chip memory capacity (such as Altera Stratix III or IV). Secondly, in order to keep the same FPGA target (for area constraint), the memory tables have to be stored in external memories. That second option greatly impact the timing performance because of the external memory access time and it also needs additional logics such as memory controllers. However, it remains relevant for applications without any critical real-time constraint such as photo applications which require image quality and higher resolution.

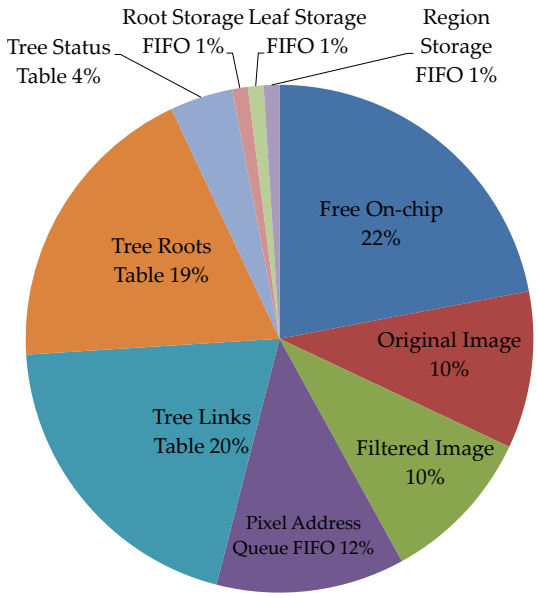


Fig. 17. On-chip memory block occupation in Stratix II 2s60

Table 4
Synthesis report STRATIX II 2S60

Resources	Quantity	Occupation 2S60
Combinational ALUTs	2680	6%
Logic Registers	3564	7%
Total RAM bits	1984470	78%

Table 5
Measured execution time (clock: 50 MHz)

Image	Size (pixels)	Tree construction (ms)	Filtering (ms)
Man	160×120	8.25	0.12
Gradient	160×120	7.41	0.22
Cars	160×120	7.75	0.55
Rooftop	160×120	7.93	6.55

Since the tree construction depends on the image content, the execution time is not known beforehand. This complexity is not only based on the number of grey levels and the connected components but also on their mutual relations (adjacency). The proposed algorithm is inspired from by already discussed quasi-linear algorithm [32], hence the execution times do not vary significantly even if the number of connected components is very different; see the Table 5 for the execution time for the tree construction. Note that in the test pictures (Fig. 15), even if the number of connected components for the Man test picture is nine times higher than the others, the tree construction

time overhead is just about 10 %. That performance results from the shortcut technique which allows to reduce the construction time dependency from the picture complexity. Note that even if the Gradient test picture has the lowest tree construction time, the shortcut technique is less efficient because the pixels are already spatially ordered and the Tree Root table is therefore less requested. Table 5 gives the execution time for the filtering. Note that the filtering time is longer for the Rooftop picture because more pixels are detected. Those timings depends on the tree complexity (number of leaves), the attributes used (contrast and height for this case) and the number of modified pixels (thresholding). Considering those variable parameters in practice, the complete application can run from 50 to 120 frames per second.

Finally, the proposed implementation allows to obtain an acceleration up to 3 on the Stratix II at 50 MHz for the tree construction compared to a standard desktop PC with Intel Pentium IV HT (3GHz) processor running under Linux. We obtain, for instance, respectively 16 ms and 21 ms for the Gradient and Man IR test images with Najman-Couprie algorithm on Pentium IV. Note that we use the C-implementation of the Salembier [38] and Najman-Couprie [32] algorithms for this comparison.

6 Conclusion

This paper presents an efficient FPGA implementation of CCT algorithms suited for embedded systems. The main contributions are: the adaptation of the CCT data structure allowing efficient data mapping in the memory, the proposal of an on-chip memory organization suited for CCT processing and a proposal of an overall embedded system for CCT-based application. After the introduction of state-of-the-art CCT algorithms and implementation, the design issues in terms of data structure and memory are discussed. The proposed architecture is then presented in detail. Assuming that this study is motivated by the search for a new approach to the flexibility of embedded systems, it demonstrates the FPGA implementation feasibility of relatively complex algorithms. In future, we will concentrate on the tree merging methods and implementation of the data parallelization by merging several 1D trees. Concerning the design challenges, the main objective is to explore and improve the interconnections between computing and memory resources.

References

[1] H.M. Alnuweiri and V.K. Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE Transactions on Pattern Analysis*

and Machine Intelligence, 14(10):1014–1034, 1992.

[2] ALTERA. Stratix ii ep2s60 dsp development board data sheet. 2006.

[3] ALTERA. Trimatrix memory in stratix ii devices. 2006.

[4] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. In *ICIP07*, volume IV, pages 41–44, 2007.

[5] C. Berger and N. Widynski. Using connected operators to manipulate. Technical report, LRDE Seminar, July 2005.

[6] Yi-Jen Chiang, Tobias Lenz, Xiang Lu, and Günter Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Comput. Geom. Theory Appl.*, 30(2):165–195, 2005.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] M. Couprie, L. Najman, and G. Bertrand. Quasi-linear algorithms for the topological watershed. *J. Math. Imaging Vis.*, 22(2-3):231–249, 2005.

[9] José Crespo, Jean Serra, and Ronald W. Schafer. Theoretical aspects of morphological filters by reconstruction. *Signal Process.*, 47(2):201–225, 1995.

[10] O. Cuisenaire and E. Romero. Automatic segmentation and measurement of axones in microscopic images. In *SPIE Medical Imaging*, volume 3661 of *Lecture Notes in Computer Science*, pages 920–929. IEEE, 1999.

[11] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr. Knight, and Andre DeHon. Graphstep: A system architecture for sparse-graph algorithms. In *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Jonathan Fabrizio and Beatriz Marcotegui. Fast implementation of the ultimate opening. In *Proceedings of the 9th International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*, ISMM '09, pages 272–281, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Holger Flatt, Steffen Blume, Sebastian Hesselbarth, Torsten Schunemann, and Peter Pirsch. A parallel hardware architecture for connected component labeling based on fast label merging. In *ASAP '08: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 144–149, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, R. Reed Taylor, and R. Reed. Piperench: A reconfigurable architecture and compiler. *Computer*, 33:70–77, 2000.

[15] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Munich, Germany, 2001. IEEE

Press.

[16] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Mapping applications onto reconfigurable kress arrays. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 385–390, London, UK, 1999. Springer-Verlag.

[17] A.C. Jalba, M.H.F. Wilkinson, and J.B.T.M. Roerdink. Morphological hat-transform scale spaces and their use in pattern classification. *Pattern Recognition*, 37(5):901–915, May 2004.

[18] Shuenn-Der Jean, Chi-Min Liu, Chih-Chi Chang, and Zen Chen. A new algorithm and its vlsi architecture design for connected component labelling. In *International Symposium on Circuits and Systems (ISCAS)*, pages 565–568, 1994.

[19] Christopher T. Johnston and Donald G. Bailey. Fpga implementation of a single pass connected components algorithm. *Electronic Design, Test and Applications, IEEE International Workshop on*, pages 228–231, 2008.

[20] R. Jones. Component trees for image filtering and segmentation. In E. Coyle, editor, *Proceedings of the 1997 IEEE Workshop on Nonlinear Signal and Image Processing*, Mackinac Island, September 1997.

[21] P. Matas, Eva Dokladalova, Mohamed Akil, Thierry Grandpierre, L. Najman, M. Poupa, and V. Georgiev. Parallel algorithm for concurrent computation of connected component tree. In Jacques Blanc-Talon, Salah Bourennane, Wilfried Philips, Dan C. Popescu, and Paul Scheunders, editors, *ACIVS*, volume 5259 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2008.

[22] Julian Mattes and Jacques Demongeot. Efficient algorithms to implement the confinement tree. In *DGCI '00: Proceedings of the 9th International Conference on Discrete Geometry for Computer Imagery*, pages 392–405, London, UK, 2000. Springer-Verlag.

[23] Julian Mattes, Mathieu Richard, and Jacques Demongeot. Tree representation for image matching and object recognition. In *DCGI '99: Proceedings of the 8th International Conference on Discrete Geometry for Computer Imagery*, pages 298–312, London, UK, 1999. Springer-Verlag.

[24] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In G. A. Constantinides P. Y. K. Cheung and J. T. de Sousa, editors, *Field-Programmable Logic and Applications*, pages 61–70. Springer, 2003.

[25] A. Meijster. *Efficient Sequential and Parallel Algorithms for Morphological Image Processing*. PhD thesis, University of Groningen, 2004.

[26] D. Menotti-Gomes, L. Najman, and A. de Albuquerque Araujo. 1d component tree in linear time and space and its application to gray-level image multithresholding. In Gerald Jean Francis Banon, Junior Barrera, Ulisses de Mendonça Braga-Neto, and Nina Sumiko Tomita Hirata, editors, *International Symposium on Mathematical Morphology*, volume 1,

pages 437–448. INPE, 2007.

[27] Alain Mérigot. Associative nets: A graph-based parallel computing model. *IEEE Trans. Comput.*, 46(5):558–571, 1997.

[28] F. Meyer and S. Beucher. Morphological segmentation. *Journal of Visual Communication and Image Representation*, 1(1):21–46, september 1990.

[29] Takashi Miyamori and Kunle Olukotun. Remarc: Reconfigurable multimedia array coprocessor. In *IEICE Transactions on Information and Systems E82-D*, pages 389–397, 1998.

[30] Alina N. Moga and Moncef Gabbouj. Parallel image component labeling with watershed transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):441–450, 1997.

[31] L. Najman and M. Couprie. Watershed algorithms and contrast preservation. In *DGCI'03*, volume 2886, pages 62–71. Lecture Notes in Computer Sciences. Springer Verlag, 2003.

[32] L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11):3531–3539, 2006.

[33] Nicolas Ngan, F. Contou-Carrère, B. Marcon, S. Guerin, Eva Dokládalova, and Mohamed Akil. Efficient Hardware Implementation of Connected Component Tree Algorithm. In *Workshop on Design and Architectures For Signal and Image Processing*, 2007.

[34] Patrick Piscaglia, Andrea Cavallaro, Michel Bonnet, and Damien Douxchamps. High level description of video surveillance sequences. In *EC-MAST '99: Proceedings of the 4th European Conference on Multimedia Applications, Services and Techniques*, pages 316–331, London, UK, 1999. Springer-Verlag.

[35] P. Salembier and L. Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *IEEE Transactions on Image Processing*, 9(4):561–576, April 2000.

[36] P. Salembier and J. Serra. Flat zones filtering, connected operators, and filters by reconstruction. *IEEE Transactions on Image Processing*, 4(8):1153–1160, August 1995.

[37] Philippe Salembier and Luis Garrido. Connected operators based on region-tree pruning. In Max Viergever, John Goutsias, Luc Vincent, and Dan S. Bloomberg, editors, *Mathematical Morphology and its Applications to Image and Signal Processing*, volume 18 of *Computational Imaging and Vision*, pages 169–178. Springer US, 2002.

[38] Philippe Salembier, A. Oliveras, and Luis Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, April 1998.

[39] Pierre Soille. Constrained connectivity for hierarchical image decomposition and simplification. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(7):1132–1145, 2008.

[40] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[41] Luc Vincent. Morphological area openings and closings for grey-scale

images. In *Proc. NATO Shape in Picture Workshop*, pages 197–208. Springer, 1992.

[42] M. Wilkinson and J. Roerdink. Fast morphological attribute operations using tarjan 's union-find algorithm. In *Mathematical Morphology and its Applications to Image and Signal Processing, Kluwer*, pages 311–320, 2000.

[43] Michael H. F. Wilkinson, Hui Gao, Wim H. Hesselink, Jan-Eppo Jonker, and Arnold Meijster. Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(10):1800–1813, 2008.