Dissertation

# A Parallel and Resource-Efficient Single Lookup Connected Components Analysis Architecture for Reconfigurable Hardware

*Michael J. Klaiber*

Author's Edition

February 2017

This is the author's version of the dissertation "A Parallel and Resource-Efficient Single Lookup Connected Components Analysis Architecture for Reconfigurable Hardware". The research work described in this document was completed April $30^{th}$ 2016, submitted to the first examiner for assessment and submitted to the Dean's Office of *Faculty 5: Computer Science, Electrical Engineering and Information Technology* of University of Stuttgart, Germany on July $26^{th}$ 2016. The examination date was December $13^{th}$ 2016. This version contains changes in content and format extending the version submitted to University of Stuttgart.

# Acknowledgements

This dissertation would not have been possible without the help and support of the people around me. There are so many that it is not possible to mention everyone here.

I am very thankful to Prof. Dr. Donald G. Bailey from Massey University in New Zealand for his advice and feedback that greatly contributed to this work. Donald, thanks for teaching me scientific thinking and scientific working. I want to express my thanks to Prof. Lionel Lacassagne from for examining this dissertation. I also acknowledge my supervisor from University of Stuttgart, S. Simon.

My thanks also goes to my colleagues from IPVS, *the Crew*: Silvia Ahmed, Yousef Baroud, Dimitrij Gester, Jajnabalkya Guhathakurta, Jürgen Hillebrand, Steffen Kieß, Wenbin Li, Xuxu Li, Mahdi Najmabadi, Lars Rockstroh, Timo Schweizer, Kaicong Sun, Trung Hieu Tran, Zhe Wang and Marek Wroblewski for all the fun we had during the last five years. It was a great pleasure working with you.

I would also like to thank the University of Stuttgart and the German Research Foundation (DFG) for funding my position as a Research Associate. My acknowledgement also go to the German Academic Exchange Service (DAAD) for their financial support of my research internship in New Zealand.

Many thanks to my parents, Elvira and Kurt Klaiber, and my sister, Susanne, for always supporting me. Most of all I want to thank my loving girlfriend Ute, who always encouraged me and was by my side.

Stuttgart, February 2017

*Michael J. Klaiber*

# Abstract

Connected components analysis (CCA) is an essential step in image processing to extract features such as the area or size of arbitrarily-shaped objects from binary images. In this dissertation two dedicated hardware architectures performing CCA tailored for reconfigurable hardware are presented: the first to process a single pixel per clock cycle, *single lookup CCA (SLCCA)*, and the second to process multiple pixels per clock cycles, *parallel SLCCA (PCCA)*. Both achieve a higher throughput than the state-of-the-art architectures which process a single pixel or multiple pixels per clock cycle. They also require fewer hardware resources, especially on-chip memory resources. The memory reduction is achieved by processing an image stream in a single pass without buffering an entire image in a frame buffer. This is especially important as image sizes and frame rates are steadily increasing, e.g. for new standards such as ultra-high-definition. Stream processing is particularly efficient for processing pixel data generated by image sensors, as a stream of multiple pixels provided in raster scan order is the standard output format of the majority of state-of-the-art image sensors. To achieve a high processing throughput for extracting feature vectors with CCA, reconfigurable hardware is used. This has the advantage over most general-purpose processors that massive parallel processing is possible and a guaranteed processing latency can be achieved.

*SLCCA* is optimised to the properties of the basic building blocks available in reconfigurable hardware, therefore a single lookup per pixel is sufficient where other state-of-the-art algorithms required a series of several lookups per pixel. The *SLCCA* hardware architecture achieves a maximum throughput of 124 Megapixel/s and a guaranteed processing latency of 160 $\mu$s, even for ultra-high-definition images (*UHD8k*). The *PCCA* architecture processes multiple pixels of a binary image simultaneously in the same clock cycle. *PCCA* makes use of spatial and temporal parallelism by first dividing the image into several slices. The results of this first step are feature vectors for each connected component in the image slices. The connected components spanning slice borders consist of multiple feature vectors which are combined by several processing stages arranged in a tree structure. *PCCA*, therefore, achieves a maximum throughput of 6 Gigapixel/s.

# Zusammenfassung

*Connected Components Analysis (CCA)* ist ein grundlegender Schritt in Bildverarbeitungssystemen, der Eigenschaften wie die Fläche oder die Größe von Objekten einer beliebigen Form aus binären Bildern extrahiert. In dieser Arbeit werden zwei CCA Hardware-Architekturen, die auf rekonfigurierbare Hardware angepasst sind, präsentiert: Die erste verarbeitet einen Pixel pro Taktzyklus – *Single Lookup CCA (SLCCA)* – die zweite verarbeitet mehrere Pixel pro Taktzyklus – *Parallel SLCCA (PCCA)*. Beide erreichen einen höheren Durchsatz, als andere *state-of-the-art* Architekturen, die einen bzw. mehrere Pixel pro Taktzyklus verarbeiten. Des Weiteren benötigen beide weniger Hardware Ressourcen, insbesondere *on-chip memory*. Diese Speicherreduzierung wird dadurch erreicht, dass Bilddatenströme in einem einzelnen Durchlauf verarbeitet werden, ohne dass ein komplettes Bild zwischengespeichert werden muss. Dies ist besonders wichtig, da Bildgrößen und Frameraten kontinuierlich wachsen, wie zum Beispiel bei neuen Standards wie *Ultra-High-Definition*. Das direkte Verarbeiten von Bilddatenströmen ist besonders effizient, um Pixel-Daten von Bildsensoren zu verarbeiten, da die Mehrheit der modernen Bildsensoren einen Pixelstrom erzeugt, der aus mehreren Pixeln in Raster-Scan-Reihenfolge besteht. Um einen hohen Durchsatz beim Extrahieren von Feature-Vektoren mit CCA zu erzielen, werden rekonfigurierbare Hardware Bausteine eingesetzt. Im Vergleich zu den meisten *General-Purpose Prozessoren* bietet dies den Vorteil einer hohen Parallelisierbarkeit und einer garantierten Verarbeitungslatenz.

*SLCCA* ist auf die Eigenschaften der elementaren Bauelemente von rekonfigurierbaren Hardware Bausteinen zugeschnitten. Dies ermöglicht eine Verarbeitung mit einem *Lookup* pro Pixel, wo andere Algorithmen eine Abfolge von mehreren *Lookups* benötigen. Die *SLCCA* Hardware-Architektur erreicht einen maximalen Durchsatz von 124 Megapixel/s und eine garantierte Verarbeitungslatenz von 160 $\mu$s, sogar für Bilder mit einer Auflösung in *Ultra-High-Definition (UHD8k)*. Die *PCCA* Architektur verarbeitet mehrere binäre Pixel gleichzeitig im selben Taktzyklus. *PCCA* macht Gebrauch von räumlicher und zeitlicher Parallelverarbeitung, indem das Bild zuerst in mehrere Bildstreifen aufgeteilt wird. Das Ergebnis dieses ersten Schrittes sind *Feature-Vektoren* für jedes der *Connected Components* in den einzelnen Streifen. Die *Connected Components*, die sich über mehrere Bildstreifen erstrecken, bestehen somit auch aus mehreren *Feature-Vektoren*, welche durch die Verarbeitung in mehreren, baumförmig angeordneten Prozessiereinheiten gleichzeitig zusammengefasst werden. *PCCA* erreicht damit einen maximalen Durchsatz von 6 Gigapixel/s.

# Contents

# 1 Introduction

In this chapter, basic ideas and terminology related to *connected components analysis* (CCA) are introduced and state-of-the-art algorithms are presented and discussed. Section 1.2 emphasises the relevance of the research topic by pointing out the variety of fields it can be applied in.

Section 1.1 points out the difference between CCA and CCL and defines the properties of both problems concisely. The conventions for quotations and citations in this dissertation are explained in Section 1.3. The notation and definition of graphs and digraphs used in the following chapters are introduced in Section 1.4. Section 1.5 defines and introduces the set merging problem and state-of-the-art set merging algorithms. In Section 1.6 historical and special cases of CCL algorithms are reviewed. Section 1.7 to 1.8 introduce and discuss sequential and parallel state-of-the-art CCA and CCL algorithms. A CCA algorithm and hardware architecture which process one pixel per clock cycle is presented in Chapter 2 and Chapter 3. In Chapter 4 and Chapter 5 an algorithm and a hardware architecture for further acceleration of CCA by parallelisation is presented. The case study in Chapter 6 applies the previously presented CCA hardware architecture on a problem from mechanical process engineering: the detection of fast droplet collisions in atomisation processes.

The content in this chapter marked by single quotation marks according to the convention from Section 1.3 is mainly from [67].

## 1.1 Definitions of Connected Components Analysis and Connected Components Labelling

The usage of the term *connected components labelling* and its definition is quite consistent in the academic literature, whereas *connected components analysis* varies in terms of both, terminology and problem definition.

Rosenfeld et al. [102] define *connected components labelling* as the "[c]reation of a labeled image in which the positions associated with the same connected component of the binary input image have a unique label." Shapiro et al. [112] define CCL as an operator whose "input is a binary image and [...] output is a symbolic image in which the label assigned to each pixel is an integer uniquely identifying the connected component to which that pixel belongs."

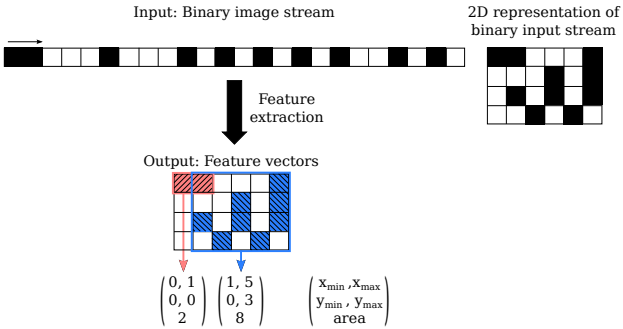


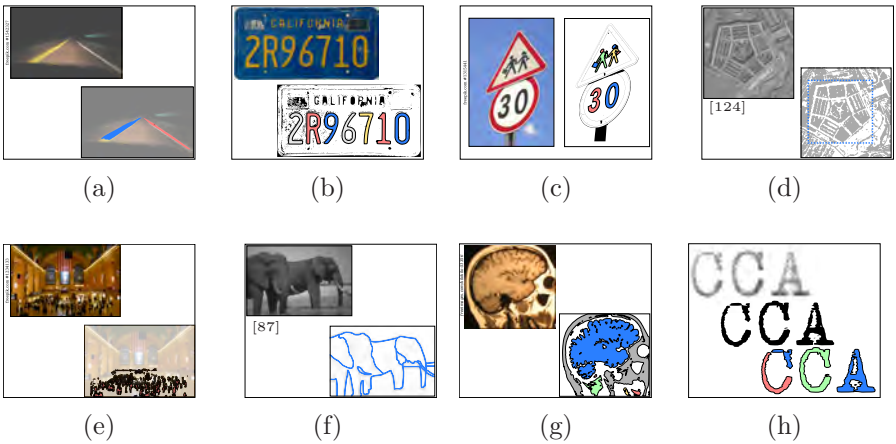

**Figure 1.1:** Definition of input and output data for connected components analysis.

There is no consensus on the definition of *connected components analysis* in the academic literature. It is often used interchangeably with CCL [37, 40]. A more extensive definition is given by Shapiro et al. [112]: "*Connected component analysis* consists of connected component labelling of the black pixels followed by property measurement of the component regions and decision making." The definition for *connected components analysis* used in the following chapters is more general, taking the thoughts expressed in [37, 40, 112] into account.

**Definition 1.** *Connected components analysis* derives one feature vector of each connected component in a binary 2-D input image. A *feature vector* of a connected component is an n-tuple composed of functions of the component's pattern [45].

The derivation of feature vectors does not necessarily require a fully labelled image. Figure 1.1 depicts an example for the input and output data of connected components analysis. The input for both CCL and CCA is a binary 2-D image. CCL assigns the same label to all pixels of one connected component. The output is, therefore, a 2-D array of labels - a labelled image. CCA derives the feature vector for each of the connected components in the input image, such as the bounding box or area of each connected component. Feature vectors can also be extracted from the output of CCL. However, this is an additional step which is not a part of CCL.

In Section 1.2, it is shown that *CCA* is a relevant processing step in many modern image processing systems. An improvement of this method is, therefore, beneficial for all of these systems.




(a) (b) (c) (d)

(e) (f) (g) (h)

**Figure 1.2:** Fields for the application of CCA or CCL. (a) Drive assistance, (b) license plate recognition, (c) traffic sign recognition, (d) aviation, (e) surveillance, (f) image segmentation, (g) medical imaging, and (h) character recognition.

## 1.2 Image Processing Systems Applying Connected Components Analysis

Connected components analysis is a fundamental processing step in many image processing systems. Figure 1.2 shows examples of the wide spectrum of applications making use of CCA. In automotive image processing systems, CCA is used as a basis for *driving assistance* systems [4, 22, 23, 34, 35, 38, 61, 78, 79, 84, 89, 103, 108], for *license plate recognition* [6, 29, 60, 81] and *traffic sign recognition* [32, 135]. In aviation, CCA is applied in *horizon detection* [24], *navigation* [36, 41, 101], *object tracking* [104] and image-based forest *fire detection* [137]. CCA is also a fundamental process for intelligent *surveillance* applications [1, 16, 123]. In *image segmentation*, CCA is used to support the process of distinguishing pixels which belong to different regions of the input image [63]. Examples for the application of CCA in *medical imaging* are the detection of lesions [114] and kidney stones [30], angiography of retina images [94], brain tumour segmentation [59, 92], computer-aided diagnosis [46] and microscopy [121]. In *optical character recognition* (OCR), CCA is used to select the connected components belonging to a character [21], character extraction and for filtering connected components depending on their properties for the character recognition step [25].

## 1.3 Citations and Quotations

This dissertation contains content which has already been published in or submitted to journals and conference proceedings. Content of pre-publications[1] are marked by the qualifiers introduced in the following. Text in single opening quotation marks (‘) followed by a reference, indicates that this text has been pre-published in, or was submitted to a journal or conference given by the reference ( ‘Example of pre-published content.‘[65]). The single opening quotation marks indicate that the cited content is equal to the published or submitted content given as a reference at the end, except for minor changes like citations, cross-references (numeric or verbal), abbreviations or single words which are required to establish consistency with the other content of this dissertation. Single closing quotation marks (’) are used to mark modified pre-published content ( ’Example of modified pre-published content.’[65]). The text between single closing quotation marks consists generally of pre-published content. There might exist changes such as: rearranged sentences, added or removed words or (sub-)sentences. Both qualifiers, *single opening quotation marks* and *single closing quotation marks* can mark pre-published content over several pages. Figure, Tables, etc., which are already pre-published, are marked by an *italic* citation. The manuscripts of pre-publications already published are available online. The language and content of the submitted pre-publications are subject to a review process and their content might alter from the finally published version. The following papers contain pre-published content of this dissertation: [64, 65, 67–70, 72].

## 1.4 Graph and Digraph Notation and Definition

’A graph $G$, consists of a set of vertices $V(G)$ and a set of edges $E(G)$ each of which is a 2-tuple of vertices joined by the edge [19].

$$\begin{aligned} G =& (V, E) \\ V(G) =& \{v_0, \ldots, v_{n-1}\} \\ E(G) =& \{(v_{i_0}, v_{j_0}), \ldots, (v_{i_{m-1}}, v_{j_{m-1}})\} \end{aligned} \tag{1.1}$$

The term graph in the following explicitly refers to an undirected graph.

In a directed graph or digraph, $D$, the edges are directed and are referred to as arcs. For each arc $(v_i, v_j)$ in $E(D)$, $v_i$ is adjacent to and directed to $v_j$.

**Definition 2.** *Path in a graph*[2]*:* For two distinct vertices $u$ and $v$ in a graph $G$, a $u - v$ *path* $P$ in $G$ is a sequence of vertices in $G$, beginning at $u$ and ending at $v$

[1] Referring to the term *Vorveröffentlichung* used in *Promotionsordnung der Universität Stuttgart vom 1. September 2011.*

[2] This definition is a compound of the definitions of the *walk* and *path* from [19, p.31].

such that consecutive vertices in $P$ are adjacent in $G$. In path $P$ a vertex of $G$ only occurs once [19].

A path in a graph $G$ starting at vertex $v_1$, passing $v_2$ and $v_3$ and ending at vertex $v_4$ is denoted as: $v_1 - v_2 - v_3 - v_4$.

**Definition 3.** *Path in a digraph*[3]*:* For two distinct vertices $u$ and $v$ in a digraph $D$, a directed $u \to v$ *path* $P$ in $D$ is a finite sequence $(u = u_0, u_1, u_2, \ldots, u_k = v)$ of vertices, beginning at $u$ and ending at $v$ such that $(u_i, u_{i+1})$ is an arc for $0 \leq i \leq k-1$. In the directed path $P$ a vertex of $D$ only occurs once [19].

A directed path in a digraph $d$ starting at vertex $v_1$, passing $v_2$ and $v_3$ and ending at vertex $v_4$ is denoted as: $v_1 \to v_2 \to v_3 \to v_4$.

**Definition 4.** *Connected (vertices):* Two vertices $v_1$ and $v_2$ in a graph $G$ or digraph $D$ are *connected* if a path in $G$ or $D$ from $v_1$ to $v_2$ exists. This is denoted as $v_1 \mapsto v_2$.

**Definition 5.** *Directed Rooted Tree*: A rooted tree $T$ is a sub-graph of an acyclic digraph with a root vertex $v_r$ with a path to all other vertices in $E(T)$. The arcs of a tree can be either towards $v_r$ [88] (Equation 1.2a) or away from $v_r$ [85,90] (Equation 1.2b).

$$\begin{aligned} V(T) &= \{v_i : v_i \in V(D) \wedge v_i \mapsto v_r\} \qquad (a) \\ V(T) &= \{v_i : v_i \in V(D) \wedge v_r \mapsto v_i\} \qquad (b) \end{aligned} \tag{1.2}$$

To represent a set merging data structures (introduced in 1.5) the first is more useful, since a representative element of several sets has to be identified. If a vertex is not the root, its *parent* is its successor on the path towards the root vertex. A *root* vertex has no parent. *Leaf* vertices are all vertices which are not a parent of another vertex in $T$. The *children* of a vertex are all vertices, the vertex is a parent of.

**Definition 6.** *Directed Forest:* A directed forest structure is a digraph which contains only non-intersecting trees and each vertex belongs to exactly one tree. Since every tree has only one root vertex, each tree is referred to by its root $v_r$.

**Definition 7.** *Connected component*[4]*:* A sub-graph $H$ of a graph $G$ is called a *connected component* if every two vertices of $H$ are connected and $H$ " is not contained in any other sub-graph of $G$ having more vertices or edges than $H$." [187]

**Algorithm 1:** QuickFind *[67]*

```
1  makeSet(vertex e)
2      parent[e] := ∅
3  find(vertex e)
4      if parent[e] = ∅ then
5          return e
6      else
7          return parent[e]
8  union(vertex e, vertex f)
9      root0 := find(e)
10     root1 := find(f)
11     for v in V(F) do
12         if parent[v] = root0 then
13             parent[v] := root1
14     parent[e] := root1
```

## 1.5 Set Merging Algorithms

A *set merging algorithm* solves the problem of "efficiently merging sets according to an initially unknown sequence of instructions, while at the same time being able to determine the set containing a given element rapidly" [53].

'Problems which require the manipulation of disjoint sets by carrying out intermixed find and union operations are *union-find problems* [119]. The *makeSet(e)* operation creates a set $S_e$ consisting of a single element $e$, a *union(e,f)* operation replaces the sets $S_e$ and $S_f$ by $S_e \cup S_f$ [53] and a *find(e)* operation returns the representative element of the set containing $e$ [53].

The most common *union-find data structure* to represent disjoint sets is a directed forest. Each directed tree of this forest structure represents a set, and each vertex an element of the set. The root vertex of each tree serves as the representative element for the set. A *union-find algorithm* defines the exact instructions of makeSet, union and find operations on the union-find data structure. In modern CCL and CCA algorithms, union-find is the set merging algorithm which is applied most often.

The set merging algorithms QuickFind (Algorithms 1), QuickUnion (Algorithm 2) and QuickUnion with path compression (Algorithm 3) are discussed in the following.

[3]This definition is a slightly modified compound of the definitions of the *(directed) walk* and *(directed) path* from [19, p.50].
[4]This definition combines the definition of *connected* and *component* from [18, p.42].

**Algorithm 2:** QuickUnion *[67]*

```
1  makeSet(vertex e)
2      parent[e] := ∅
3  find(vertex e)
4      if parent[e] = ∅ then
5          return e
6      else
7          return find(parent[e])
8  union(vertex e, vertex f)
9      root0 := find(e)
10     root1 := find(f)
11     if root0 ≠ root1 then
12         parent[root0] := root1
```

These algorithms operate on a directed forest, $F$, with $n_F$ vertices as a union-find data structure. Adding a vertex to $F$, changing the parent of a vertex or looking up the parent of a vertex in $F$ is in the following referred to as one *uf-instruction* (union-find instruction).

*QuickFind* (Algorithm 1) maintains $F$ so that every leaf is directly connected to the root vertex, i.e. the height of every rooted tree is at most one [109]. A find operation of *QuickFind*, therefore, consists of one uf-instruction. A union operation checks which vertices of $F$ belong to the rooted trees to be combined and changes their parents. A union operation in QuickFind can, therefore, require up to $2 \times n_F$ uf-instructions in the worst case: one to lookup the parent of each vertex and one to change all the parents.

*QuickUnion* (Algorithm 2) joins their root vertices to perform a union operation on two vertices. A find operation carries out repetitive lookups to traverse the path from a vertex to its root vertex. Therefore, QuickUnion requires two find operations for one union operation, which requires in the worst case up to $n_F$ uf-instructions [109]. Both *QuickFind* and *QuickUnion* have quadratic run time in the worst case [109].

*QuickUnion with path compression* (Algorithm 3) joins all vertices which are visited during a find operation directly to the root vertex. Whenever these values are accessed again they will point directly to the root (at the time that the path was compressed). The worst case run time of *QuickUnion with path compression* grows with the inverse of the Ackermann function [2,118] (which is quasi-linear for practical cases) when the tree size of the union-find data structure is balanced with a heuristic such as union-by-rank [118], which is not discussed, being beyond the scope of this dissertation.

**Algorithm 3:** QuickUnion with path compression *[67]*

```
1  makeSet(vertex e)
2      parent[e] := ∅
3  find(vertex e)
4      return findAndCompress(e)
5  findAndCompress(vertex e)
6      if parent[e] = ∅ then
7          return e
8      else
9          r := findAndCompress(parent[e])
10         parent[e] := r
11         return r
12 union(vertex e, vertex f)
13     root0 := find(e)
14     root1 := find(f)
15     parent[root0] := root1
```

For connected components analysis or labelling, the sequence of union and find operations is still unknown (as in the definition of *set merging algorithm* in [53]) but restricted by properties of how two-dimensional images are processed, such as the scan order and scan direction. This is exploited to derive an even more efficient union-find algorithm for the special case of CCA and CCL of two-dimensional images. This algorithm considers the context of the pixels of the input image processed by CCA and CCL, such as the frequency a vertex associated with a pixel is used in find operations. It is, therefore, called *context-based union-find (CB-UF)* and presented in Section 2.2 (Algorithm 4).'[67]

## 1.6 Review on Classical CCL and Special Case CCL Algorithms for Image Processing

The algorithm considered to be the first CCL algorithm was presented by Rosenfeld et al. [102]. This algorithm combined for the first time the concepts of *pixel connectivity*, *forward raster scan*, a *decision tree* and *equivalence classes* which lay the foundations of basically all modern CCA and CCL algorithms. The exact algorithm as described in [102] is hardly used for comparison against modern algorithms, due to the used set merging algorithm. However, most modern publications benchmark against an algorithm very close to the one described by Rosenfeld et al. which use one raster scan to identify equivalence classes, and a second raster scan to assign a distinct label to all pixels associated with a connected component.

The CCL algorithm by *Haralick* [44] introduces multi-pass connected components labelling. This approach scans the image alternating in forward raster scan order and backward raster scan order. New labels are associated with an object pixel if no adjacent pixel is associated with a label. If a label is associated with an adjacent pixel, it is also associated with the current pixel. When there are several different labels associated with adjacent pixels the minimum label is associated with the current pixel. In this way, the minimum label is propagated back and forth in the raster scan until all the pixels of a connected component are associated with a unique connected component. For complex pixel patterns in the input image, many scans can be required, which is one of the main reasons why the multi-pass approach is hardly used in modern CCL algorithms.

*Samet et al.* [105, 106] introduce an efficient CCL algorithm to extract feature vectors from images which are represented as bin-trees or quad-trees. The presented algorithm requires two passes to process an input image consisting of image elements in a bin-tree or quad-tree representation. During the first pass, the algorithm examines each pair of adjacent image elements, which represent one or several object pixels of a binary image, to identify equivalence classes, and stores them in an equivalence table [105]. In the second pass of this two pass algorithm, a label is assigned to each image element. This was one of the first CCL algorithms to identify the well-researched union-find problem [53, 118] as an efficient basis of CCL.

Besides connected components labelling of two-dimensional images there are also a number of CCL algorithms for processing three-dimensional images [54, 56, 91, 116]. In [91] the time is added as a fourth dimension in order to track changes in a series of 3-D images. Even-though these algorithms follow the same basic principles as 2-D CCL algorithms, e.g. applying union-find [56], they are beyond the scope of this dissertation as the focus is on processing 2-D images.

## 1.7 Evaluation and Categorisation of State-of-the-Art Sequential CCA and CCL Algorithms for Image Processing

'Since the introduction of the classic connected components labelling algorithm by Rosenfeld et al. [102], CCL has been continuously improved in many aspects. In the following, mainly modern CCA and CCL algorithms are discussed.'[67] Even-though Rosenfeld's algorithm [102] cannot be considered a modern CCA or CCL algorithm it is used as a reference for comparison. A CCA or CCL algorithm is considered sequential only if one pixel is processed at a time, i.e. neither temporal nor spatial parallelism is applied. 'A summary of several properties of the discussed algorithms is given in Table 1.1 which compares different properties, such as: scan mode and scan order, the number of passes, the worst case run time, the evaluation method for worst case run time and performs a categorisation of set merging algorithm used.

The algorithm in [102] is the very classical CCL algorithm. It is a two-pass algorithm where the first pass uses a binary image as an input and creates a labelled image. If more than one label is assigned to a connected component, these labels are stored in an equivalence table $T$. These equivalence relations detected during the first scan are resolved by iteratively sorting and replacing the entries of table $T$ until $T$ contains one entry for each connected component. After this process, each entry of $T$ contains all labels assigned to its connected component in the first pass sorted in ascending order, starting with the smallest label which serves as a representative element. During the second pass, all the object pixels of the labelled image are replaced by their representative values stored in $T$. This assigns the same label to each pixel of a connected component.

Dillencourt et al. [28] proposed a general two-pass CCL algorithm for different image representation such as 2-D arrays and quad-trees. This algorithm uses QuickUnion with path compression extended by a heuristic called *age-balancing* embedded into the union operation. Age-balancing ensures that the label associated with the first pixel of a connected component encountered during a scan is always the root vertex. Using this property, it is formally proven that the algorithm used in [28] scales linearly with the number of pixels.

In [115], a multi-pass CCL algorithm is proposed using an equivalence table (called connection table) $M$ to store the relations between provisional labels. This algorithm is, therefore, referred to as a *scan plus connection table* (SCT) CCL algorithm. Previous multi-pass algorithms propagated labels by neighbourhood operations [45]. The algorithm in [115] creates a forest structure stored in the equivalence table $M$ during the first scan with one tree structure for each connected component consisting of provisional labels as vertices. Every scan over the image decreases the height of the tree structure in $M$ by one.'[67] This effectively distributes the steps of a find operation over the multiple scans, since only the parents of neighbourhood

| Algorithm | # Passes | Scan mode | Scan order | Connectivity |
|---|---|---|---|---|
| Classical [102] | 2 | pixel | raster scan | 8 |
| SCT [115] | multi | pixel | raster scan | 8 |
| SAUF [127] | 2 | pixel | raster scan | 8 |
| GCCL [28] | 2 | pixel | raster scan | 4 |
| RTS [49] | 2 | run | raster scan | 8 |
| HCS [48] | 1.5 | run | raster scan | 8 |
| LSL [76] | 3 | run | raster scan | 8 |
| SEL [27] | 2 | pixel | raster scan | 4 |
| CT [17] | 1.5 | pixel | contour tracing | 8 |
| Bailey2007 [7] | 1 | pixel | raster scan | 8 |

| Algorithm | RT complexity | RT evaluation method | Set merging algorithm |
|---|---|---|---|
| Classical [102] | N/A | N/A | Rosenfeld [102] |
| SCT [115] | Linear | Experimental | None |
| SAUF [127] | Linear | Formal proof | QuickUnion + pc |
| GCCL [28] | Linear | Formal proof | QuickUnion + pc |
| RTS [49] | N/A | N/A | QuickFind + opt |
| HCS [48] | Linear | Experimental | QuickFind + opt |
| LSL [76] | N/A | N/A | QuickUnion |
| SEL [27] | N/A | N/A | QuickFind |
| CT [17] | Linear | Formal Proof | None |
| Bailey2007 [7] | Linear | Proof | [7] |

**Table 1.1:** This table shows a comparison of properties such as run time (RT) complexity and the method of evaluating the run time (RT eval meth). Additionally, the set merging algorithms according to the definitions from Section 1.5 are identified. Some of them are an optimised variant (opt) of the algorithm from 1.5, while some use path compression (pc). *[67]*

pixels are looked up in $M$. 'The algorithm associates each pixel with its connected component, however, it cannot be categorised as a union-find algorithm such as those in Section 1.5. The run time is stated to be linear which is determined by experimental evaluation. Most of the images used for evaluation require four or fewer passes for final labelling [115].

In the two-pass CCL algorithm in [125–127] the union-find data structure is represented by an array, therefore, it is referred to as a *scan plus array-based union-find* (SAUF). QuickUnion with path compression is used to maintain this array-based union-find data structure. To accelerate the label selection process for each pixel, a decision tree is proposed reducing the number of labels of the neighbourhood to be accessed. A formal proof for the linear run time of the algorithm is given.

The CCL algorithm in [49] is a two-pass algorithm which carries out run-length encoding of the binary image during the first pass and processes these runs in the second pass. The algorithm uses a union-find data structure stored in an array which is updated by an optimised variant of QuickFind. To avoid updating all entries of $M$ for a union operation, an additional linked list is maintained for each tree structure in $M$ containing all the vertices of the tree structure. A union operation on two vertices links the two lists and updates the equivalence table entries of these vertices to the root vertex. In [50], an optimisation of [49] is proposed which processes only runs of object pixels in the second pass. In [48], the Euler number is also computed and the rest of the algorithm is equivalent to [50]. Since, this review focuses on the feature extraction, only the part of [48] involved in CCL is considered. In the following, [48] is referred to as *HCS*.

In [27], *simple and efficient connected components labelling (SEL)* is presented. It requires two passes to label all pixels using an equivalence table as union-find data structure carrying out the QuickFind algorithm. The algorithm is improved for the worst case image. The image pattern becoming the new worst case with the proposed improvement still, however, requires a quadratic number of uf-instructions.

For *Light Speed Labeling (LSL)* [76] memory accesses and conditional statements were identified to be the key issue slowing down CCL on state-of-the-art processors with a RISC (reduced operation set computer) architecture. The algorithm is consequently following this optimisation principle by distributing the labelling process to three passes replacing conditional operations. Both QuickUnion and Selkow's algorithm [111] are applied as a set merging algorithm. Selkow's algorithm [111] is a variation of the idea behind QuickFind. In [14], *LSL* was identified to require the fewest amount of processing cycles per pixel when carried out on a general-purpose processor (GPP).

The algorithm in [17] follows a completely different approach. Instead of scanning the image in raster scan order, connected components are identified by contour tracing which requires random access to the image data. It is, therefore, referred to as *CT* in the following. All control information is stored in the labelled image,

therefore no set merging algorithm is required. This algorithm is considered to be a single-pass algorithm. However, random access to the input image, as well as the generation of control information is required. Actually more than one pass is required; in Table 1.1 it is, therefore, denoted as a 1.5 pass algorithm. The required random access makes this algorithm less practical for recent GPPs and dedicated hardware architectures.

Some of the CCL algorithms compared in this section have formally proven linear run time [17, 28, 127]. All of the two-pass CCL algorithms use a set merge algorithm which requires either a minimum of two instructions for a find operation, or have a union operation which scales quadratically with the number of labels. The major contribution of the *SLCCA* algorithm presented in Chapter 2 is to introduce the *one lookup per pixel* paradigm, which can be reached by a novel context-based union-find algorithm.'[67]

The algorithm by *Bailey et al.* [7] is a single-pass CCA algorithm which aims to accelerate CCA on *FPGA* devices. It builds the groundwork for the *SLCCA* algorithm presented in Chapter 2 and the *PCCA* algorithm presented 4 and can, therefore, be considered the algorithm with the most influence to this dissertation. It was the first algorithm to introduce a stack to accelerate union-find operations, which builds the basis for the *context-based union-find* algorithm presented in Section 2.2. The FPGA hardware architecture of this CCA algorithm is presented in [8, 58]. An optimisation to reduce the amount of memory required compared to [7] is presented in [83]. The technique introduced in [83] is referred to as *aggressive relabelling* in the following. A linear run time in the number of pixels of the algorithm is demonstrated by analysis of the worst case pattern.

## 1.8 Evaluation and Categorisation of State-of-the-Art Parallel CCA and CCL Algorithms

Even in early CCA algorithms, the prospect of accelerating the execution by parallel processing was identified and explored [86]. Since parallel processing devices such as multi-core processor systems and FPGAs are emerging, parallelism is an important issue to accelerate image processing systems applying CCA. Here a CCA or CCL algorithm is considered a parallel CCA or CCL algorithm if it processes several pixels simultaneously and makes use of *spatial parallelism* or *temporal parallelism*.

Table 1.2 compares several parallel state-of-the-art CCA and CCL algorithms with respect to the same criteria as in Section 1.7: scan mode, scan order, connectivity and the used set merging algorithms. The scalability with the number of processing instances or threads is a major issue for parallel CCA and CCL algorithms, therefore, the algorithms discussed in the following are compared with respect to:

- Types of parallelism used,

- Maximum number of parallel processing threads/instances $TR$,
- Speed-up $S_{max}$: from $TR$ compared to a single thread/instance.

In Tables 1.2 and 1.3, CCA and CCL algorithms designed and optimised for different processing devices from general-purpose processors (GPP) to graphics processing units (GPUs) are shown.

Since the presented algorithms have different design goals, and the reported benchmark results are from the execution on different processing devices, a direct quantitative comparison is not useful. However, since almost all of the presented algorithms are based on the same basic algorithmic principles (set merging algorithm, run-length encoding, etc.), an overview of the scalability of these principles on different processing devices is given.

The algorithm in [42] at first divides the input image into several slices (called chunks). These slices are processed in parallel using the decision tree proposed in [127] and the linked list approach from [50]. As set merging algorithms *QuickFind* and a parallel version of *RemSP* [100] are used; *QuickFind* for processing the image slices and *RemSP* to merge the labels of slice components in parallel. The algorithm is executed and benchmarked on a *Cray XE6* multi-processor system, storing image data and control information in off-chip memory and processor caches. The scalability is analysed for $2, \ldots, 24$ image slices, each processed by one thread. For processing natural images from the SIPI database [124], a speed-up of $S_{max} \approx 5$ is reached when executing $TR = 24$ threads.

The *SPCCA* algorithm [98] is a single pass CCA algorithm extracting feature vectors from a binary input image. It makes use of temporal parallelism by dividing the images into vertical slices which are processed in parallel. The merging of the border labels is sequential. The reported results are from executing *SPCCA* on $1, \ldots, 4$ cores of a multi-core *XMOS* processor. The presented speed-up grows sub-linearly with

| Algorithm | #Passes | Scan mode | Scan order | Connectivity | Set merging algorithm |
|---|---|---|---|---|---|
| PaRemSP [42] | 2 | pixel | raster scan | 8 | QuickFind/ RemSP [100] |
| SPCCA [98] | 1 | pixel | raster scan | 8 | N/A |
| Chen2014 [20] | 2 | pixel | raster scan | 8 | QuickFind + opt [50] |
| Stava2011 [113] | 2 | pixel | raster scan | 8 | QuickUnion |
| ParLSL [12] | 2/3 | run | raster scan | 8 | QuickFind |

**Table 1.2:** Comparison of the scan mode, scan order, connectivity and set merging algorithms of parallel CCA and CCL algorithms used.

the number of threads. For executing *SPCCA* on four processor cores, a speedup of $S_{max} = 2$ is reached.

The CCL algorithm proposed by Chen et al. [20] processes several pixels simultaneously by extending the merging method introduced in [50] . The algorithm makes use of spatial parallelism by dividing the image into vertical image slices, each executed on a separate processing instance. This first step uses QuickFind in addition with the linked list optimisation from [50] as a set merging algorithm. In a second processing step, temporal parallelism is exploited by associating the labels assigned to neighbouring slice components with their connected components. This step uses a parallel version of the *merge* function from [50], merging the labels of neighbouring slices step by step in a tree-like manner. The CCL algorithm proposed in [20] is executed and benchmarked on the many-core architecture TILE64 using external off-chip memory to store image data and control information. The scalability is analysed by evaluation of the speed-up of $2, \ldots, 48$ image slices (executed on $2, \ldots, 48$ processor cores) compared to a single one. For an increasing number of processor cores, the speed-up seems saturated. For 48 cores the speed-up is 11.38. The synchronisation between the processing cores and the data dependencies in the second pass are identified as bottleneck for achieving better performance.

In [113], a CCL algorithm optimised for GPUs (graphics processing unit) is presented. The input image stored in the global memory of the GPU is divided horizontally and vertically into image tiles which are processed in parallel. In this first processing step, each pixel of the input image is passed twice. This is followed by a recursive merging step carried out on several tiles in parallel. As set merging algorithm QuickUnion is used. The scalability of the presented algorithm with the number of threads is not presented in [113].

| Algorithm | Parallelism | $TR$ | $S_{max}$ | Processing device | Memory type |
|---|---|---|---|---|---|
| PaRemSP [42] | *Spat*+*Temp* | 24 | 5 | Cray XE6 | OFF+ON |
| SPCCA [98] | *Spat* | 4 | 2 | XMOS | OFF+ON |
| Chen2014 [20] | *Spat*+*Temp* | 48 | 11 | TILE64 | OFF+ON |
| Stava2011 [113] | *Spat* | N/A | N/A | NVIDIA GTX480 | OFF+ON |
| ParLSL [12] | *Spat*+*Temp* | 24 | 18.8-22.2 | Intel Xeon E5-2695v2 | OFF+ON |

**Table 1.3:** Comparison of parallel CCA and CCL algorithms: Maximum number of threads $TR$. Speed-up $S_{max}$ for $TR$ threads/instances compared to a single one. Parallelism type: spatial (*Spat*) or temporal (*Temp*). Memory type: on-chip memory (*ON*) or off-chip memory (*OFF*).

Parallel *Light Speed Labeling (ParLSL)* [12] is the parallelisation of *LSL* [76]. The image at first is divided into $p$ horizontal image slices which are labelled in parallel. Afterwards, the border labels are merged. Due to the proposed *pyramid merging* method, the border labels of several image slices can be merged simultaneously. The benchmarks from [12], carried out on a Intel Ivy-Bridge Xeon E5-2695v2 multi-processor system with 24 processing cores, show that *ParLSL* scales almost linearly with the number of image slices, where each slice is processed by one of the processor cores. A speed-up between 18.8 and 22.2 compared to processing on a single core is achieved when executing *ParLSL* on all 24 processor cores in parallel. Due to the run-length encoding and other optimisations introduced in *LSL*, *ParLSL* is very memory-efficient. These optimisations reduce the amount of memory required for control information to fit in on-chip caches of state-of-the-art processors which reduces the access to slower off-chip memory. In [13] *ParLSL* was further improved for multi-processor systems and benchmarked with a 60-core quad socket Intel Xeon system.

# 2 *SLCCA* - The Single-Lookup Connected Components Analysis Algorithm

In this chapter the single-lookup connected components analysis algorithm, *SLCCA*, is introduced. The majority of content in single quotation marks is based on [67].

'Some CCA and CCL algorithms are optimised to the instruction sets or memory architectures of the hardware device they are used on [76]. The *SLCCA* algorithm is motivated by the idea of creating a CCA algorithm for a customised high-performance architecture. *SLCCA*, therefore, only contains operations and data structures which can be mapped to basic processing and basic storage elements such as adder, lookup table (LUTs), flip-flops or on-chip random-access memory (RAM). To realise such a high-performance architecture, the properties explained in the following are deemed most important:

- *Single-pass processing*
- *Linear processing time*
- *Single lookup per pixel to determine the representative label*

Two-pass or multiple pass algorithms maintain a data structure for a labelled image [50, 115] and optimise the access to this labelled image to accelerate the processing of the input data. For a *single-pass* CCA algorithm, a labelled image is not required, hence does not need to be optimised in a way.

Acquiring the input data is linear since the binary input image is either read from a memory or received as a pixel stream. The *SLCCA* algorithm processes the input data in linear time as pixels are read or received. This will be shown in Section 2.5.

A pixel can only be processed if a label has been assigned to the preceding pixel in the scan through the image. This sequential data dependency requires several lookup instructions in many CCL and CCA algorithms [28, 49, 50, 115, 127] due to the union-find data structures the equivalence relations are represented by. The *SLCCA* algorithm introduces a data structure and union-find algorithm which reduces the number of lookups to determine the representative label of a pixel to a *single lookup* per pixel. The proposed novel optimisations of the union-find algorithm embodied in *SLCCA* are context-based. These optimisations consider the restricted order in which union and find operations are induced when processing pixel patterns of 2-dimensional images in raster scan to reduce the number of

| Abbreviation | Name |
|---|---|
| $DT$ | Data table |
| $AT$ | Active tags |
| $F$ | Label graph for $L$ |
| $FV$ | Feature vector |
| $G_P$ | Pixel graph |
| $H$ | Image height |
| $I$ | Source image |
| $L$ | Labelled image |
| $LS$ | Label stack |
| $M$ | Merger table |
| $R$ | Reuse FIFO |
| $S$ | Stack |
| $VF$ | Valid flags |
| $W$ | Image width |

**Table 2.1:** Nomenclature used in this chapter. *[67]*

operations necessary. The union-find algorithms used in previous CCL and CCA algorithms [28, 49, 50, 115, 127] require several lookups per pixel to identify which connected component a pixel is associated with. The single lookup property was identified as being especially useful for dedicated hardware architectures and is further discussed in Chapter 3 which introduces an architecture using the *SLCCA* algorithm. The *SLCCA* algorithm enables this hardware architecture realised on an FPGA to process a higher throughput than any other CCA architecture (processing one pixel per clock cycle) in the academic literature [65]. In Section 2.3, the proposed algorithm, *SLCCA*, is presented. In Section 2.5, SLCCA is compared to other CCA and CCL algorithms.'[67]

## 2.1 General Definitions

'The abbreviations and names of data structures used in the following are summarised in Table 2.1. The input image $I$ is a binary image of width $W$ and height $H$. An image position is a two-dimensional coordinate $(a, b)$, where $a$ and $b$ are non-negative integers, hence $a \in \mathbb{N}_0$, $b \in \mathbb{N}_0$, $0 \leq a < W$ and $0 \leq b < H$. The set $imagePos$ consists of all image positions in $I$.

$$imagePos = \{(a, b) : 0 \leq a < W,\ 0 \leq b < H,\ a \in \mathbb{N}_0,\ b \in \mathbb{N}_0\}. \tag{2.1}$$

At each image position of $I$ there is either an *object pixel* or a *background pixel*. In the following, an *object pixel* is represented by the Boolean value *True* in equations

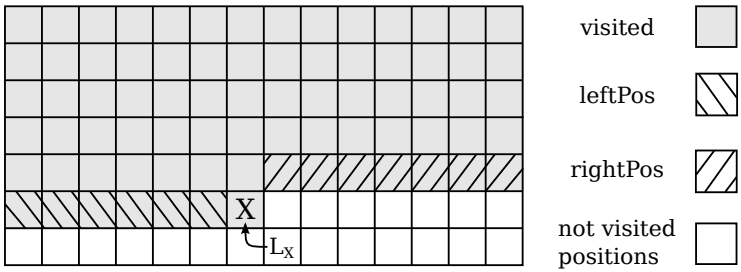



**Figure 2.1:** Visualisation of the positions in the sets *visited*, $rightPos$, $leftPos$ in the image. *[67]*

and by a black pixel in images. A *background pixel* is represented by the Boolean value *False* in equations by a white pixel in images. The pixels of $I$ are visited in forward *raster scan order*. The image is scanned in each row from left to right, starting the scan at the top left position of the image, $(0, 0)$. After the end of each image row, at the horizontal position $W - 1$, the next row is processed starting at horizontal position 0. This process is continued until the position at the right bottom of the image, $(W - 1, H - 1)$, is reached. This is defined by the $W \times H$-tuple $rasterScan$ containing $W \times H$ image positions.

$$rasterScan = ((0,0), ..., (W-1,0), (0,1), ...(W-1, H-1)). \tag{2.2}$$

A position $p_1$ preceding another position $p_2$ in $rasterScan$ is denoted as $p_1 \prec p_2$. If position $p_2$ succeeds $p_1$, this is denoted by $p_2 \succ p_1$.
The sets of positions $leftPos$, $rightPos$ and *visited* are defined for raster scan order as depicted in Figure 2.1. The set $leftPos(a, b)$ contains the positions of the pixels of the current row to the left of position $(a, b)$.

$$leftPos(a,b) = \{(i,b) : 0 \leq i < a,\ i \in \mathbb{N}_0\}. \tag{2.3}$$

The set $rightPos(a, b)$ contains the positions of the pixels of the previous row to the right of position (a,b).

$$rightPos(a,b) = \{(i,b-1) : a < i < W,\ i \in \mathbb{N}_0\}. \tag{2.4}$$

The set $visited(a, b)$ contains all positions of all pixels which have already been visited including position $(a, b)$.

$$visited(a,b) = \{(i,j) : (i,j) \prec (a,b),\ i \in \mathbb{N}_0, j \in \mathbb{N}_0\} \cup (a,b). \tag{2.5}$$

For the current position $(x, y)$, $leftPos(x, y)$, $rightPos(x, y)$ and $visited(x, y)$ provide the corresponding sets of image positions.

**Definition 8.** *Pixel graph:* Let $G_P$ be the pixel graph with one vertex $v_{a,b}$ for each object pixel in $I$ at position $(a, b)$.

$$V(G_P) = \{v_{a,b} : I[(a, b)] = 1 \wedge (a, b) \in imagPos\}. \tag{2.6}$$

Two vertices of $G_P$, $v_{a1,b1}$ and $v_{a2,b2}$, are *adjacent* if

$$\|(a1, b1) - (a2, b2)\| = 1. \tag{2.7}$$

Here, 8-connectivity is assumed (i.e. using uniform norm $\| \ \|_{L\infty}$), although the same can be applied for 4-connectivity (using Taxicab geometry $\| \ \|_{L1}$). Adjacent pixels in $G_P$ are also connected. The set of edges $E(G_P)$ contains a 2-tuple for each pair of adjacent vertices $(v_{a1,b1}, v_{a2,b2})$.

$$\begin{aligned} E(G_P) =& \{(v_{a1,b1}, v_{a2,b2}) : v_{a1,b1}, v_{a2,b2} \in V(G_P) \\ & \wedge \|(a1, b1) - (a2, b2)\| = 1\}. \end{aligned} \tag{2.8}$$

The object pixels in $I$ and their vertices in $G_P$ have a one-to-one relation. Therefore, the properties defined for the pixel graph $G_P$ do hold true for pixels in $I$, too.
$G_P(x, y)$ denotes the state of $G_P$ immediately after processing the pixel at $(x, y)$ in raster scan order.

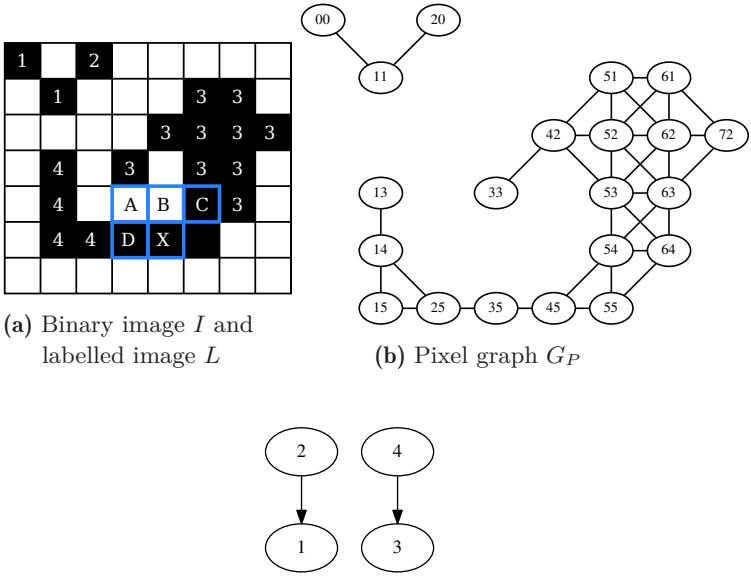
$$\begin{aligned} V(G_P(x, y)) =& \{v_{a,b} : v_{a,b} \in V(G_P) \wedge \ (a, b) \in visited(x, y)\}. \\ E(G_P(x, y)) =& \{(v_i, v_j) : v_i, v_j \in V(G_P(x, y)) \wedge (v_i, v_j) \in E(G_P)\}). \end{aligned} \tag{2.9}$$

**Definition 9.** *Connectedness*: Two vertices $v_{p1}$ and $v_{p2}$ in $G_P$ are *connected* if a path exists consisting of vertices from $V(G_P)$ and arcs from $E(G_P)$. This is denoted as $v_{p1} \mapsto v_{p2}$.
Two pixels in $I$ at position $p1$ and $p2$, $I[p_1]$ and $I[p_2]$, belong to the same connected component if their vertices $v_{p1}$ and $v_{p2}$ are connected.

**Definition 10.** *Component segment:* A subset of connected object pixels of a connected component of $I$ is called a *component segment*.

Connected components labelling assigns a label $L[(x, y)]$ to each pixel at position $(x, y)$, with the goal of eventually assigning the same label to all pixels belonging to a single connected component. Label 0 is reserved for background pixels. The label graph $F$ is a directed forest structure where each vertex, $v_L \in V(F)$, corresponds with exactly one label assigned to $L$.

The labelling process assigns labels by associating every vertex of $V(G_P)$ with exactly one vertex of $V(F)$. Let $v_{a,b} \in V(G_P)$ and $v_{L_i} \in V(F)$ be the vertex corresponding to $L_i$, then associating $v_{a,b}$ with $v_{L_i}$ is equivalent to assigning $L[(a, b)] \coloneqq L_i$.




**(a)** Binary image $I$ and labelled image $L$

**(b)** Pixel graph $G_P$

**(c)** Label graph $F$

**Figure 2.2:** A label is assigned to each pixel in raster scan order. In 2.2(a) the neighbourhood of a pixel at position $X = (x, y)$ is shown. In 2.2(b) the pixel graph $G_P$ of the image in 2.2(a) is shown, and 2.2(c) shows the corresponding label graph $F$. *[67]*

An example pixel graph $G_P$ and label graph $F$ both derived from the image in Figure 2.2(a) are shown in Figure 2.2(b) and Figure 2.2(c).

Whenever a vertex $v_i \in G_P$ is processed in raster scan order, it is assigned a label $L_X$. Where possible $L_X$ is assigned the label of an adjacent processed vertex. Otherwise, a *makeSet* operation creates a new vertex in $F$, which is associated with associated with $L_X$. If pixels of a connected component are processed which are not connected through the already scanned pixels ($G_P(x, y)$), they are first detected as different component segments and are associated with different trees in $F$. As soon as a pixel is reached which connects two previously disjoint sub-graphs of $G_P$, $F$ is updated by a union operation. The state of $F$ after processing the pixel at position $(x, y)$ is denoted $F(x, y)$.

**Definition 11.** *Level (of a vertex in a directed rooted tree):* The level of a vertex labelled $l$, $level(l)$, is 0 for the root vertex and for all other vertices becomes one higher than the level of its parent [73].

$$level(v_L) = \begin{cases} 0, & v_L \text{ is root}, \\ level(parent(v_L)) + 1, & otherwise. \end{cases} \tag{2.10}$$

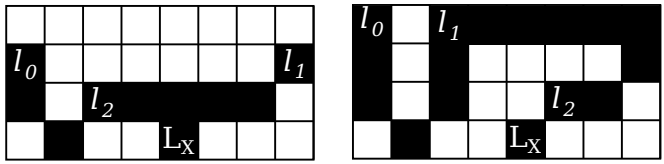



**Figure 2.3:** Two examples of images containing stale label $l_2$. A non-root label is assigned to $L_x$, because a stale label is in the neighbourhood. *[67]*

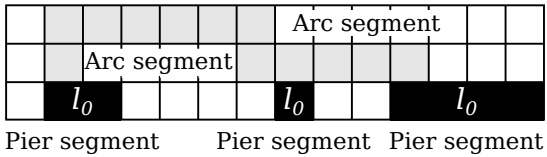



**Figure 2.4:** Example of a bridge pattern labelled $l_0$ with tree piers segments (black) and two arc segments (shown in grey). *[67]*

**Definition 12.** *Height (of a directed rooted tree):* The height of a directed rooted tree $T$, $height(T)$, is the maximum level of a vertex in $V(T)$.

$$height(T) = \max\{level(v_i) : v_i \in V(T)\}. \tag{2.11}$$

The arcs of the label graph $F$, used as union-find data structure, are stored in a 1-D array, the *merger table* $M$. A label $l_0$ associated with a vertex $v_0$ is joined with a label $l_1$ associated with a vertex $v_1$, by updating $M[l_1] = l_0$ which is the equivalent of adding an arc $(v_1, v_0)$ to $E(F)$. The label $l_0$ is called the *parent label* of $l_1$ and $l_1$ is called the *child label* of $l_0$. For convenience, every label $l_r$ of a root vertex $v_r$ is joined with itself, $M[l_r] = l_r$. A label which is its own parent label is called a *root label*.

**Definition 13.** *Stale label:* A label $l_s = L[(x, y)]$ is called a *stale* label if a single lookup in $M$ does not yield the root label.

A label becomes stale if the height of its vertex is larger than one due to several merger operations. In the examples in Figure 2.3, labels $l_1$ and $l_2$ merge, which makes $height(l_2) = 1$. The merger operation on $l_0$ and $l_1$ in the following row result in $height(l_2) = 2$. This makes $l_2$ stale, i.e. the single lookup at position $(x, y)$ results in assigning $L_x = l_1$ to the current pixel, which is a non-root label.

**Definition 14.** *Bridge pattern:* A bridge pattern is a component segment in which an object label appears two times in the same image row separated by background pixels. An example of a bridge is given in Figure 2.4.'[67]

## 2.2 Relation of Union-Find to SLCCA: The Set Merging Algorithm used by SLCCA

'For connected components analysis, the order of performing union and find operations is governed by the connectivity of pixels of a two-dimensional image and the order in which the pixels are scanned. Algorithm 4 exploits this to achieve linear run time and requires fewer union-find instructions in the worst case than *QuickFind* (Algorithm 1), *QuickUnion* (Algorithm 2) and *QuickUnion with path compression* (Algorithm 3). Since Algorithm 4 is optimised to the order union-find operations are issued in CCA, it is dependent on the context of the current pixel when scanning an image. Therefore, it is referred to as *context-based union-find (CB-UF)*.

The SLCCA algorithm is an advancement of the algorithm proposed by *Bailey et al.* [7]. It combines the best features of QuickFind and QuickUnion. Like QuickFind, the find operation requires only one union-find instruction. The union operation of two vertices $e$ and $f$ joins the root vertex of $e$ with the root vertex of $f$, similar to QuickUnion. In addition to makeSet, union and find operations, a fourth operation, *flatten*, is introduced which performs the equivalent of path compression by joining all vertices of the union-find data structure in $M$ with their root vertex. To accelerate *flatten*, the arcs joining vertices with levels larger than one are memorised by storing them in a stack for every union operation. The find operation returns the parent of a vertex, i.e. only returns the root vertex for vertices of level zero or one. The instance (here the CCA algorithm) using *context-based union-find* (Algorithm 4) has to ensure that the flatten operation is always called before a find operation is applied on a vertex with level larger than one.

Normally, a *find* operation is used to determine the root of a vertex [118] by a series of iterative lookups. In *context-based union-find* (Algorithm 4), the find operation is replaced by a single lookup in the merger table $M$, which is equivalent to a *find* for trees of height $\leq 1$ *QuickUnion* (Algorithm 2).

## 2.3 Algorithmic Description of SLCCA

### 2.3.1 Neighbourhood Patterns and Operations

The *labelled image* $L$ is a two-dimensional array of labels with the same dimensions as the input image $I$. A label associated with a pixel of $I$ at position $(a, b)$ is assigned to the same position of $L$. The neighbourhood $\eta$ of the current pixel at position $(x, y)$ is the set of positions of adjacent pixels that have already been processed, i.e. $\eta = \{(x-1, y-1), (x, y-1), (x+1, y-1), (x-1, y)\} = \{A, B, C, D\}$. The labels in $L$ at the neighbourhood positions $\eta$ are denoted $L_A$, $L_B$, $L_C$ and $L_D$. To move from

**Algorithm 4:** Context-based union-find algorithm. *[67]*

```
1  makeSet(vertex e)
2      M[e] := e
3  find(vertex e)
4      return M[e]
5  union(vertex e, vertex f)
6      root0 := find(e)
7      root1 := find(f)
8      if root0 ≺ root1 then
9          M[root1] := root0
10     else
11         Stack.push(root1,root0)
12         M[root0] := root1
13 flatten()
14     while not Stack.empty do
15         L_min,L_max := Stack.pop()
16         M[L_max] := find(L_min)
```

one window position to the next, their values are designated right before selecting the label for the current position, as follows:

$$\begin{aligned} L_A &\coloneqq {L_B}^-, \\ L_B &\coloneqq {L_C}^-, \\ L_C &\coloneqq M[L[C]], \\ L_D &\coloneqq {L_X}^-. \end{aligned} \tag{2.12}$$

The labels ${L_B}^-$, ${L_C}^-$ and ${L_X}^-$ , shown in Figure 2.5, are neighbourhood labels of the previous label at position $(x-1, y)$. Labels of positions outside of the image are considered as background, i.e. $L[i] = 0 \; \forall i \notin imagePos$.

The set $L_\eta$ denotes all labels of object pixels in the neighbourhood of the current pixel.

$$L_\eta \coloneqq \{L[i] : i \in \eta, \; I[i] = 1\}. \tag{2.13}$$

Let

$$\begin{aligned} L_{min} &\coloneqq \begin{cases} 0, & L_\eta = \emptyset, \\ \min\{L_\eta\}, & otherwise, \end{cases} \\ L_{max} &\coloneqq \begin{cases} 0, & L_\eta = \emptyset, \\ \max\{L_\eta\}, & otherwise. \end{cases} \end{aligned} \tag{2.14}$$

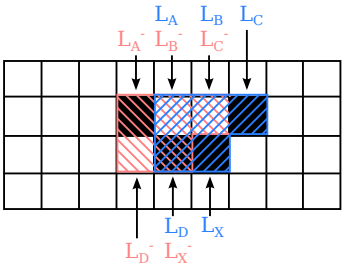



**Figure 2.5:** This image shows the neighbourhood labels of current pixel $L_X$, $L_A, \ldots, L_D$, and the neighbourhood labels of the previous pixel, $L_A^-, \ldots, L_D^-$.

The label $L_X$ to be assigned to the pixel at position $(x, y)$ of $L$ is

$$L_X \coloneqq \begin{cases} 0, & I[(x,y)] = 0, \\ newLabel, & L_{min} = 0, \\ L_{min}, & otherwise, \end{cases} \qquad (2.15)$$
$$L[(x,y)] \coloneqq L_X.$$

**Definition 15.** *Label patterns and label operations:* For object pixels, there are three different scenarios with either zero, one or two different object labels in the neighbourhood, $L_\eta$ which are referred to as *new label pattern*, *label copy pattern* and *merger pattern.* An operation induced by such a pattern is referred to as *new label operation*, *label copy operation* or *merger operation*, respectively.

A *new label operation* is performed if an object pixel has no object pixels in its neighbourhood, i.e. it is assigned the next available new label. This invokes a *makeSet* operation on the graph of the union-find data structure $F$ creating a new rooted tree. In general, the new label (called *newLabel* in Equation 2.15 and 2.16) is provided by a counter, which is incremented for each new label. To more easily detect labels of level $> 1$, a flag $VF$ is associated with each label. For each *new label* operation, the merger table $M$ as well as $VF$ is updated:

$$\begin{aligned} M[newLabel] &\coloneqq newLabel, \\ VF[newLabel] &\coloneqq True. \end{aligned} \qquad (2.16)$$

A *label copy operation* assigns $L_{min}$ to the current position of the labelled image $L[(x, y)]$.

Since adjacent object pixels at the positions $\eta$ will already have the same label as a result of prior processing, a merger pattern can only occur between non-adjacent pixels, i.e. between $L_A$ and $L_C$ or $L_D$ and $L_C$ [127], as shown in Figure 2.6.

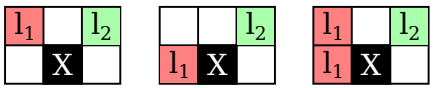



**Figure 2.6:** Merger patterns possible in the labels of pixel neighbourhood $L_\eta$. *[67]*

To simplify discussion, $L_{AorD}$ is introduced to refer to the label of $L_A$ or $L_D$, i.e. all merger patterns consist of the two labels $L_{AorD}$ and $L_C$.

$$L_{AorD} \coloneqq \begin{cases} L_D, & I[A] = 0, \\ L_A, & otherwise. \end{cases} \tag{2.17}$$

A merger pattern is detected when $L_{AorD}$ and $L_C$ have different labels and neither is background.

$$MergerPat \coloneqq (L_{AorD} \neq L_C) \ \wedge \ (L_{AorD}, L_C \neq 0). \tag{2.18}$$

A merger operation makes the label which first appears in the raster scan, $L_{min}$, the parent label of $L_{max}$. This corresponds to a union operation. The vertex associated with $L_{max}$ is no longer a root so $VF[L_{max}]$ is set to *false*.

$$\begin{aligned} M[L_{max}] &\coloneqq L_{min}, \\ VF[L_{max}] &\coloneqq False. \end{aligned} \tag{2.19}$$

**Definition 16.** *Propagating and non-propagating patterns:* Concerning forward raster scan order, a merger pattern is *propagating* if the first occurrence of $L_{AorD}$ in $L$ in forward raster scan order precedes the first occurrence of $L_C$ in forward raster scan. Otherwise the merger pattern is *non-propagating*.

**Definition 17.** *Chain pattern:* A series of more than one non-propagating merger patterns in the same image row of a connected component is called a *chain pattern*.

In Figure 2.7(a), an example of propagating merger patterns is shown. The label assigned to position ① is propagated to position ② and ③. In Figure 2.7(b), an example of non-propagating merger patterns is shown. The labels at position ④, ⑤ and ⑥ are not propagated. The vertices of the labels *5, 6, 7, 8* are connected as a chain in the label graph $F$, as shown in Figure 2.7(c).

The minimum label $L_{min}$ of a propagating merger pattern has to be propagated into the next neighbourhood. This is achieved by updating $L_B$ with the previous position's $L_{AorD}$, ${L_{AorD}}^-$, if a propagating merger pattern was detected at the previous position ($MergerPat^- \wedge {L_{AorD}}^- = {L_{min}}^-$). Therefore, Equation 2.12 is extended to

$$L_B \coloneqq \begin{cases} {L_{min}}^-, & MergerPat^- \wedge {L_{AorD}}^- = {L_{min}}^-, \\ {L_C}^-, & otherwise. \end{cases} \tag{2.20}$$

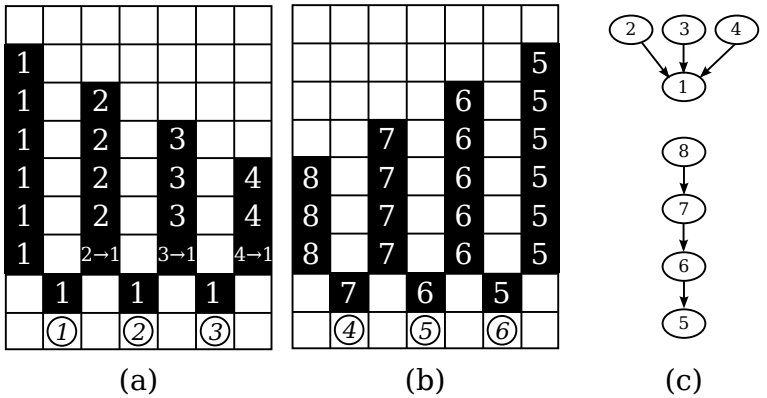



**Figure 2.7:** (a) Example of propagating merger patterns. (b) Example of a chain pattern consisting of non-propagating merger patterns. (c) Label graph of image in (a) at the top and Label graph of image in (b) below.

### 2.3.2 Flattening Trees in the Union-Find Structure

'A prerequisite for the *CB-UF* (Algorithm 4) is that all trees of the forest structure in $M$ are flattened to height $\leq 1$. This can be achieved by using *path compression*, the equivalent of which is embodied in the *flatten* operation. In *QuickUnion with path compression* (Algorithm 3) the path compression is performed in association with the *find* operation [117] as *findAndCompress*. The flatten operation of *CB-UF* (Algorithm 4) starts at the root vertex and processes towards the leaves, unlike standard path compression approaches [110], which process from the leaves towards the root. Since minimum labels propagate to the right because of the raster scan by assigning the minimum to $L_X$ which becomes $L_D$ (Equation 2.12 and 2.20), the height of a tree in $F$ is increased by one for each non-propagating merger pattern. Therefore, the arc from $L_{max}$ to $L_{min}$ created by a union operation induced by a non-propagating merger pattern is pushed onto the stack $S$ to accelerate flattening.

$$\begin{aligned} S[s] &:= (L_{max}, L_{min}) \\ s &:= s + 1 \end{aligned} \quad when\ MergerPat \wedge L_{min} = L_C. \tag{2.21}$$

At the end of each image row the flatten operation is invoked. This pops the arcs off stack $S$, visiting them in reverse order, effectively performing a reverse scan along the row.

$$\begin{aligned} s &:= s - 1, \\ (L_{max_s}, L_{min_s}) &:= S[s]. \end{aligned} \tag{2.22}$$

The vertex associated with label $L_{max_s}$ in $S$ is made the child of the minimum label $L_{min_s}$ which propagates the root back to the leaves, effectively flattening the forest structure in $M$ to a height of one.

$$M[L_{max_s}] := M[L_{min_s}]. \tag{2.23}$$

| Feature | Feature vector | Initial feature vector $IFV$ | Combining operator $FV_a \circ FV_b$ |
|---|---|---|---|
| Area | $A$ | 1 | $A_a + A_b$ |
| Bounding box | $\begin{pmatrix} x_{min} \\ y_{min} \\ x_{max} \\ y_{max} \end{pmatrix}$ | $\begin{pmatrix} x \\ y \\ x \\ y \end{pmatrix}$ | $\begin{pmatrix} \min(x_{min,a}, x_{min,b}) \\ \min(y_{min,a}, y_{min,b}) \\ \max(x_{max,a}, x_{max,b}) \\ \max(y_{max,a}, y_{max,b}) \end{pmatrix}$ |
| First order moment | $\begin{pmatrix} M_{10} \\ M_{01} \end{pmatrix}$ | $\begin{pmatrix} x \\ y \end{pmatrix}$ | $\begin{pmatrix} M_{10a} + M_{10b} \\ M_{01a} + M_{01b} \end{pmatrix}$ |

**Table 2.2:** Description of data structure and combining operator for the Feature Vectors *Bounding Box* and *Area* and *First Order Moment*. *[67]*

### 2.3.3 Feature Vector Collection

**Definition 18.** *Feature vector:* The *feature vector* of an image component is an n-tuple composed of functions of the component's pattern [45]. *Connected components analysis* is concerned with deriving the feature vector for each component. An operator $\circ$ is defined for combining the feature vectors when a merger operation is induced. The initial feature vector (IFV) is the feature vector of a single pixel.

Table 2.2 presents the combining operation, the initial feature vectors and the data structures for the extracting *area*, *bounding box* and *first-order moment* of connected components.

To accumulate the feature vectors of component segments, a *data table* $DT$ maintains one feature vector for each label. For a background pixel, nothing needs to be saved in $DT$. For a *new label* pattern, the initial feature vector $IFV$ of the current pixel $(x, y)$ is written to $DT$ at label $L_X$. For a *label copy pattern*, the current pixel's $IFV$ is combined with the feature vector stored to $DT$. If a *merger pattern* occurs, the feature vectors of the object labels in $L_\eta$ are combined with the $IFV$ and stored in $DT[L_{min}]$, the data table entry at index $L_{max}$ is invalidated. The operations to update data table $DT$ at position $(x, y)$ are given in Pseudocode 1 (data table update operations). '[67]

**Pseudocode 1:** Data table update operations. *[67]*

```
if L_η = ∅ then //New label pattern
    DT[L_X] ≔ IFV
else if ¬MergerPat then //Label copy pattern
    DT[L_X] ≔ DT[L_min] ∘ IFV
else if MergerPat then //Merger pattern
    DT[L_X] ≔ DT[L_min] ∘ DT[L_max] ∘ IFV
    DT[L_max] ≔ ∅
```

### 2.3.4 Non-root Label Selection

**Definition 19.** *Reachable:* The vertices assigned to $L$ at a position of $rightPos$ and their parent vertices are *reachable*. The set $V_{reachable}$ contains all *reachable* vertices. A label is called *reachable* if its vertex of the label graph $F$ is in $V_{reachable}$.

$$V_{reachable} = \{L[i] \cup parent(L[i]) : i \in rightPos\ \} \tag{2.24}$$

If a label of $L_\eta$ assigned to $L_X$ is not associated with a root vertex, that label is reachable and stale. This requires an additional lookup to determine the root vertex. The valid flag $VF$ is used to check whether a label is associated with a root vertex or not. If a non-root label is assigned to $L_X$, the feature vectors of the object labels in $L_\eta$ are combined and stored to data table entry $DT[L_X]$ for a later combination with the feature vector of the root of $L_X$. The non-root label is pushed onto the *label stack (LS)* until its root appears in $L_\eta$. To avoid duplicate entries which lead to increased memory requirements and processing times, a label is only added to $LS$ if it is different to the top entry $LS[u-1]$.

$$\begin{aligned} LS[u] &\coloneqq L_X \\ u &\coloneqq u+1 \end{aligned} \quad when\ \neg VF[L_X] \wedge L_X \neq LS[u-1]. \tag{2.25}$$

If the top entry of $LS[u-1]$ is equal to $L[C]$, then $L_C$ is associated with the root vertex. In this case, the feature vector of $DT[LS[u-1]]$ is combined with and stored in $DT[L_C]$, since they belong to the same connected component. The data table entry of $LS[u-1]$ is then invalidated. This enables an on-the-fly processing of feature vectors of reachable stale labels and is described in detail in Pseudocode 2 (stale label combine operations).'[67]

**Pseudocode 2:** Combine operations for reachable stale label. *[67]*

```
1 if LS[u-1]=L[C] then
2     u := u-1
3     if L_X ≠ 0) then
4         DT[L_C] := DT[L_C] ∘ DT[LS[u]] ∘ IFV
5     else
6         DT[L_C] := DT[L_C] ∘ DT[LS[u]]
7     DT[LS[u]] := ∅
```

### 2.3.5 Label Reuse

'The memory requirements of $M$ and $DT$ are proportional to the image area [7]. However, at any time, the number of feature vectors updated in one image row is only proportional to the image width [62, 82]. Memory requirements can be significantly reduced by recycling labels no longer in use, enabling entries of $M$ and $DT$ to be reused after a connected component is completed. Rather than use a counter to provide new labels, labels are obtained from a FIFO $R$ initialised to contain the elements of the set $R_{init}$, which contains all possible labels.

$$R_{init} = \{1, \ldots, \lceil \tfrac{W}{2} \rceil\}. \tag{2.26}$$

Labels which are ready for reuse are queued at the end of $R$.

An active tag $AT$ is associated with each label to indicate whether the label appears in the row currently being processed. During the raster scan when a label is assigned to $L_X$, its entry in $AT$ is updated with the current image row $y$ mod 3.

$$AT[L_X] := \ y \bmod 3 \quad when \quad L_X \neq 0. \tag{2.27}$$

A connected component or component segment with label $l$ is detected as *completed* when its active tag $AT$ was last updated in row $y - 2$ (it was not extended onto in the previous row $y$).

$$end(l) := (AT[l] = (y - 2) \bmod 3). \tag{2.28}$$

Of course, all remaining objects are detected as completed at the end of the image. The data table $DT$ is searched for feature vectors of already ended connected components once per row in parallel with the update process. When a completed component is detected, the feature vector from the data table is output.
The data table entry is then cleared to be reused by a subsequent connected component and the label and associated memory can be reused for subsequent components by returning the label to the end of the FIFO $R$.

$$\begin{aligned} DT[l] :=& \emptyset, \ \forall l \in \{1, \ldots, \lceil \tfrac{W}{2} \rceil\} \ \wedge \ end(l) \wedge VF[l], \\ & l \rightarrow R. \end{aligned} \tag{2.29}$$

Adding a label $l$ to a FIFO, e.g. $R$, is denoted as $l \rightarrow R$.

After every merger operation, $L_{max}$ is no longer required. However, it must not be reused for one image row since the labelled image $L$ still might contain $L_{max}$ in the current image row to the left of the current position. Writing $L_{max}$ to the end of FIFO $R$ ensures that it is not assigned to a new connected component within the following image row.

$$L_{max} \rightarrow R \;\; when \;\; MergerPat. \tag{2.30}$$

The reuse of labels in this way requires modifying the method used to determine $L_{min}$ and $L_{max}$. Without label recycling, labels produced by *new label* operations strictly increase in scan order. Therefore, realising the $\prec$-operator as a comparison is sufficient. By reusing labels, the numeric labels assigned to component segments do not necessarily increase according to the scan order. Therefore, to realise the functionality of the $\prec$-operator with label reuse, augmented labels are introduced. An augmented label is a two-tuple consisting of the row number $rw$ in which the label is first assigned and the *index* which is used as an address to access array data structures. The elements of an augmented label are referenced using the notation $L_X.index$ to refer to the index of label $L_X$ and $L_X.rw$ to refer to the row number of label $L_X$. If an augmented label is used to access an array, its index is used, e.g. $DT[L_X]$ translates to $DT[L_X.index]$. The row number $rw$ is used for decisions in merger operations, while the *index* is used for accessing the tables. This ensures that $L_{min}$ is always the label that is created earlier during processing, leading to correct behaviour [7] when a merger pattern is detected. In this way, Equation 2.14 is modified to:

$$\begin{aligned}
L_{min} \coloneqq & \begin{cases} 0, & L_\eta = \emptyset, \\ L_\eta, & \neg MergerPat, \\ L_C, & (L_{AorD}.rw > L_C.rw \wedge VF[L_C]) \\ & \vee \neg VF[L_{AorD}], \\ L_{AorD}, & otherwise. \end{cases} \\
L_{max} \coloneqq & \begin{cases} L_{AorD}, & (L_{AorD}.rw > L_C.rw \wedge VF[L_C]) \\ & \vee \neg VF[L_{AorD}], \\ L_C, & otherwise. \end{cases}
\end{aligned} \tag{2.31}$$

When a new label is assigned to a component segment, it is pulled from the head of the label reuse FIFO $R$ and the current image row $y$ is assigned to it, i.e. *newLabel* as used in Equation 2.15 is modified to

$$\begin{aligned}
&newLabel.rw \coloneqq y, \\
&newLabel.lbl \leftarrow R.
\end{aligned} \tag{2.32}$$

## 2.4 Pseudocode of SLCCA

In the following the previously explained constituents of *SLCCA* are presented as pseudocode. The line numbers used refer to Algorithm 5 and Pseudocode 3 to Pseudocode 5. Circled numbers refer to the example in Section 2.4.7.

### 2.4.1 Forward Raster Scan

The input image $I$ is scanned in forward raster scan order from top left to bottom right (line 1 - 2) and calls *updateNeighbourhood* (Pseudocode 3), *updateDataStructures* (Pseudocode 5) and *resolveStaleLabels* (Pseudocode 6) for each visited pixel and *flatten* (Pseudocode 4) when the end of an image row is reached (line 3 - 6). In parallel to the forward raster scan, *findFinishedComponents* (Pseudocode 7) checks whether the data table $DT$ contains feature vectors of finished connected components (line 7) and moves them to the set of completed feature vectors $FC$ to free the entries of the data structures for feature vectors of subsequent connected components.

**Algorithm 5:** Pseudocode of SLCCA algorithm.

```
   Input: Binary image I of width W and height H
   Output: Set of feature vectors FC
1  for y = 0 to H − 1 do
2      for x = 0 to W − 1 do
3          updateNeighbourhood()
4          updateDataStructures()
5          resolveStaleLabels()
6      flatten()
   // Simultaneous read-out of finished feature vectors
7  findFinishedComponents()
```

### 2.4.2 UpdateNeighbourhood

The label of the current pixel at position $(x, y)$ is called $L_X$. The labels $L_A$, $L_B$, $L_C$, $L_D$ at the positions $A = (x-1, y-1)$, $B = (x, y-1)$, $C = (x+1, y-1)$ and $D = (x-1, y)$ are called the neighbourhood of the current pixel $L_X$ at position $(x, y)$. After assigning a label to the current, $L_A$ through $L_D$ are updated without accessing the labelled image $L$ (line 8 to 11). Label $L_C$ is updated with its parent, $M[L[C]]$, to identify the label of the root vertex (line 10). Since $L_A$, $L_D$ always have the same label if they are object pixels, $L_{AD}$ is introduced for convenience (line 12 to line 15). An example of these operations is shown in (13) - (15).

**Pseudocode 3:** updateNeighbourhood()

```
8  L_A := L_B
9  L_B := L_C
10 L_C := M[L[C]]
11 L_D := L_X
12 if I[A] = 0 then
13     L_AD := L_D
14 else
15     L_AD := L_A
```

### 2.4.3 UpdateDataStructures

There are three label patterns with either zero, one or two different object labels in the neighbourhood of an object pixel which are referred to as *new label pattern* (line 20), *label copy pattern* (line 43) and *merger pattern* (line 27). A label copy pattern is any pattern in the neighbourhood which is not a new label pattern or a merger pattern. These patterns are handled by the *new label operation* (line 21 to 25), the *label copy operation* (line 44 to 45) and the *merger operation* (line 27 - 42) which change the content of data structures $M$, $VF$, $DT$, $AT$ and $S$. Examples of these operations are shown in ② - ④.

The initial feature vector ($IFV$) is the feature vector of a single pixel at position $(x, y)$. The $\circ$-operator combines feature vectors. Both $IFV$ and $\circ$-operator depend on feature vector which is extracted. A bounding box feature vector consists of two 2-D coordinates, the top left $(x_1, y_1)$ and bottom right $(x_2, y_2)$. The $IFV$ and the $\circ$-operator are in this case defined as:

$$IFV(x, y) = \begin{pmatrix} x, y \\ x, y \end{pmatrix},$$

$$A \circ B = \begin{pmatrix} min(A.x_1, B.x_1), min(A.y_1, B.y_1) \\ max(B.x_2, B.x_2), max(A.y_2, B.y_2) \end{pmatrix}.$$

For extracting the area, it is sufficient to count the number of pixels. In this case the $IFV$ is equals 1 for and the $\circ$-operator is an addition.

$$IFV(x, y) = 1,$$
$$A \circ B = A.area + B.area.$$

For each new label operation, an entry is read off the label reuse FIFO $R$ and assigned to $L_X$. The data table entry of $L_X$ is initialised with the current pixel's initial feature vector ($IFV$), and the $VF$ flag of $L_X$ is set to *True*. A label copy operation assigns the object pixel in the neighbourhood to $L_X$ and combines $L_X$'s data table

entry, $DT[L_X]$, with the initial feature vector of the current pixel $IFV(x, y)$. For each merger operation there are two different object labels in the current pixel's neighbourhood: $L_{AD}$ and $L_C$. If $L_{AD}$ precedes $L_C$ in raster scan order ($L_{AD} \prec L_C$), $L_{AD}$ is assigned to $L_{min}$ and $L_C$ to $L_{max}$. If $L_C$ precedes $L_{AD}$ ($L_C \prec L_{AD}$), $L_C$ is assigned to $L_{min}$ and $L_{AD}$ to $L_{max}$. The merger table $M$ is updated to contain a directed edge from $L_{max}$ to $L_{min}$ (line 39). A merger operation also combines the feature vectors at $DT[L_{min}]$ , $DT[L_{max}]$ and $IFV(x, y)$ and stores it to $DT[L_{min}]$. The entry of $DT[L_{max}]$ is invalidated and the valid flag $VF[L_{max}]$ of $L_{max}$ is set to *False*. The label $L_{max}$ is added to $R$ for reuse. However, the reuse FIFO $R$ must ensure that when a label is added, it is not assigned to $L$ for the next $W$ pixels in the raster scan. For a propagating merger pattern $L_{AD}$ precedes $L_C$, $L_{AD} \prec L_C$. To propgate the minimum label $L_{min}$ into the next pixel's neighbourhood, $L_C$ is updated with $L_{min}$ (line 31, (10) - (11)). For a non-propagating merger pattern $L_{AD}$ succeeds $L_C$, $L_{AD} \succ L_C$. The non-propagating merger operation pushes the label pair $\{L_{min}, L_{max}\}$ onto the stack $S$ (line 34, (4)-(6)). To detect finished connected components, the active tag $AT$ of $L_X$ is updated with $y \bmod 3$, where $y$ is the current row number.

Lines 46 and 47 of the *updateDataStructures()* are discussed together with *resolveStaleLabels* in Section 2.4.5.

### 2.4.4 Flatten

At the end of each image row, each directed rooted tree of the forest structure in $M$ is flattened. In this way, when processing the next image row a single lookup is sufficient to yield the root label for each label appearing in the neighbourhood. This is realised by popping the directed edges off $S$ and using them to update $M$ with $M[L_{max}] \coloneqq M[L_{min}]$ (line 16 - 18). An example for the flatten operation is given in (7) to (9).

**Pseudocode 4:** flatten()

```
16 while ¬S.empty do
17 |  {L_min, L_max} := S.pop
18 |  M[L_max]       := M[L_min]
```

**Pseudocode 5:** updateDataStructures()

```
19 if I[X] = 1 then
       // New label pattern
20     if ¬I[A] ∧ ¬I[B] ∧ ¬I[C] ∧ ¬I[D] then
           // New label operation
21         nl       := R.pop
22         L_X      := nl
23         M[L_X]   := L_X
24         VF[L_X]  := True
25         DT[L_X]  := IFV(x, y)
26     else
           // Merger pattern
27         if (I[A] ∨ I[D]) ∧ I[C] ∧ (L_AD ≠ L_C) then
               // Merger operation
28             if L_AD ≺ L_C then
29                 L_min := L_AD
30                 L_max := L_C
31                 L_C   := L_min
32             else
33                 L_min := L_C
34                 L_max := L_AD
35                 S.push({L_min, L_max})
36             if VF[L_max] then
37                 R.push(L_max)
38             L_X          := L_min
39             M[L_max]     := L_min
40             VF[L_max]    := False
41             DT[L_min]    := DT[L_min] ∘ DT[L_max] ∘ IFV(x, y)
42             DT[L_max]    := ∅
           // Label copy pattern = ¬New label pattern ∧ ¬Merger
              pattern
43         else
               // Label copy operation
44             L_X          := posMin(L_AD, L_B, L_C)
45             DT[L_X]      := DT[L_X] ∘ IFV(x, y)
46         if ¬VF[L_X] ∧ LS.head ≠ L_X then
47             LS.push(L_X)
48     AT[L_X]  := y mod 3
49 else
50     L_X := 0
51 L[(x, y)] := L_X
```

### 2.4.5 ResolveStaleLabels

*Stale labels* require two lookups in $M$ to yield their root label. This is a result of a combination of two merger patterns which result requires special treatment. A detailed analysis on pattern and configurations of the forest structure in $M$ this applies to is given in [66].

For realising a hardware architecture it is more efficient to delay the second lookup to the next postion in the image row it is contained in, instead of carrying out two lookups in $M$ per processed pixel for stale labels [66].

The parent of a stale label is always a non-root label, which valid flag was set to *False* by a previous merger operation. This is used to detect when a non-root label is assigned to $L_X$. Then, it is push onto the *label stack LS* (line 47 of *updateDataStructures*, (15)).

If the label at the head of $LS$ is equals to the label at $L[C]$ (line 52, (16)), $L_C$ is detected to be its root label ((17)) and their feature vectors are combined 57. If the current pixel $L_X$ is an object pixel, its $IFV$ is combined with the resulting feature vector, as well(line 55). The resulting combined feature vector is stored to $DT[L_C]$, the entry of $DT[L[C]]$ is invalidated. Using two lookups per pixel instead of one, simplifies the algorithm and makes *resolveStaleLabels* superfluous at the cost of performance. For the hardware architecture in [66] adding a second lockup for each pixel halves the throughput. On a general purpose processor this might result in similar performance.Two lookups per pixel are carried out by replacing line 10 with $L_C \coloneqq M[M[L[C]]]$.

**Pseudocode 6:** resolveStaleLabels()

```
52 if LS.head = L[C] then
53 |   nonRoot      := LS.pop
54 |   if I[X] = 1 then
55 |   |   DT[L_C]  := DT[L_C] ∘ DT[nonRoot] ∘ IFV(x, y)
56 |   else
57 |   |   DT[L_C]  := DT[L_C] ∘ DT[nonRoot]
58 |   DT[nonRoot]  := ∅
```

### 2.4.6 FindFinishedComponents

In parallel to assigning labels to $L$ and extracting feature vectors, *findFinishedComponents()* continuously checks which feature vectors are associated with already finished connected components. Every label which was assigned to a pixel in the previous row, but is not assigned to $L_X$ in the current row, belongs to a finished connected component, i.e. the label and the associated entries in $M$, $DT$, $AT$ and $VF$ can be reused immediately. For this purpose the active tag $AT$ was introduce, which is updated with $y$ mod 3 if $L_X$ is an object pixel (line 48 of *updateDataStructures()*). A finished connected component can, therefore, be reliably identified if either the end of $I$ is reached or its active tag $AT$ was not updated in the last time two rows above the current (line 61). The label of a finished connected component is recycled to $R$, its feature vector is added to the set of finished connected components $FC$, and the assoicated data table entry is invalidated (line 63 - 66, (18) ).

**Pseudocode 7:** findFinishedComponent()

```
59 i := 1
60 while True do
61     if (AT[i] = (y − 2) mod 3) ∨ (end of image) then
62         if DT[i] ≠ ∅ then
63             FC    := FC ∪ DT[i]
64             DT[i] := ∅
65             VF[i] := False
66             R.push(i)
67         i := (i + 1) mod N_L
```

### 2.4.7 Step-by-step Example of SLCCA

Figure 2.8 to Figure 2.12 show the contents of all data structures for carrying out Algorithm 5 on an example image which contains a connected component of a complex pattern. The binary image shown is $I$, where a black pixel is an object pixel and a white pixel background. The labels of $L$ are shown on top of the binary image. The current pixel at position $(x, y)$ is marked blue. Its neighbour labels $L_A$, $L_B$, $L_C$, and $L_D$ are hatched red. The contents of the data structures present the state after a pixel has been assigned to the current pixel, just before processing the next pixel. There are some steps showing intermediate states. These are mentioned in the figure caption.

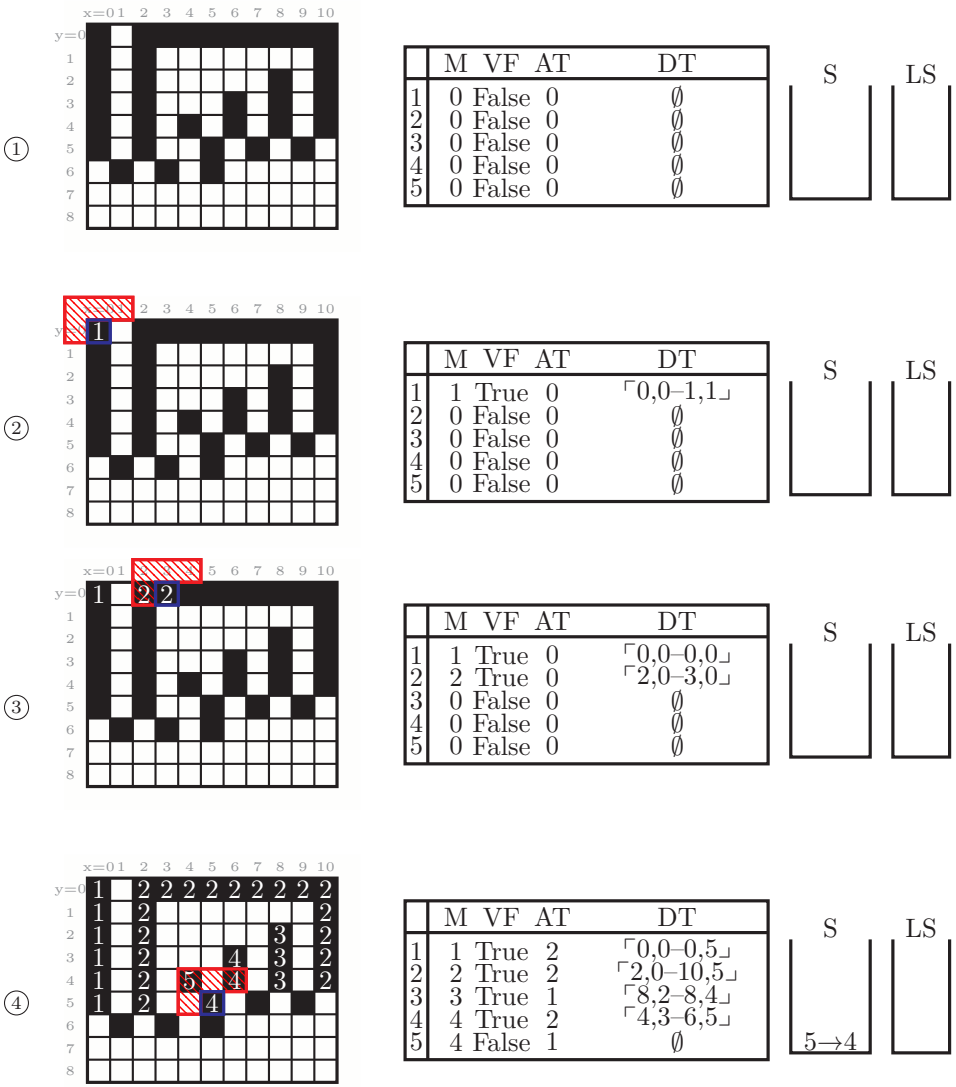



**Figure 2.8:** Step ①: Start of raster scan. Step ②: New label operation. Step ③: Label copy operation. Step ④: Non-propagating merger operation.

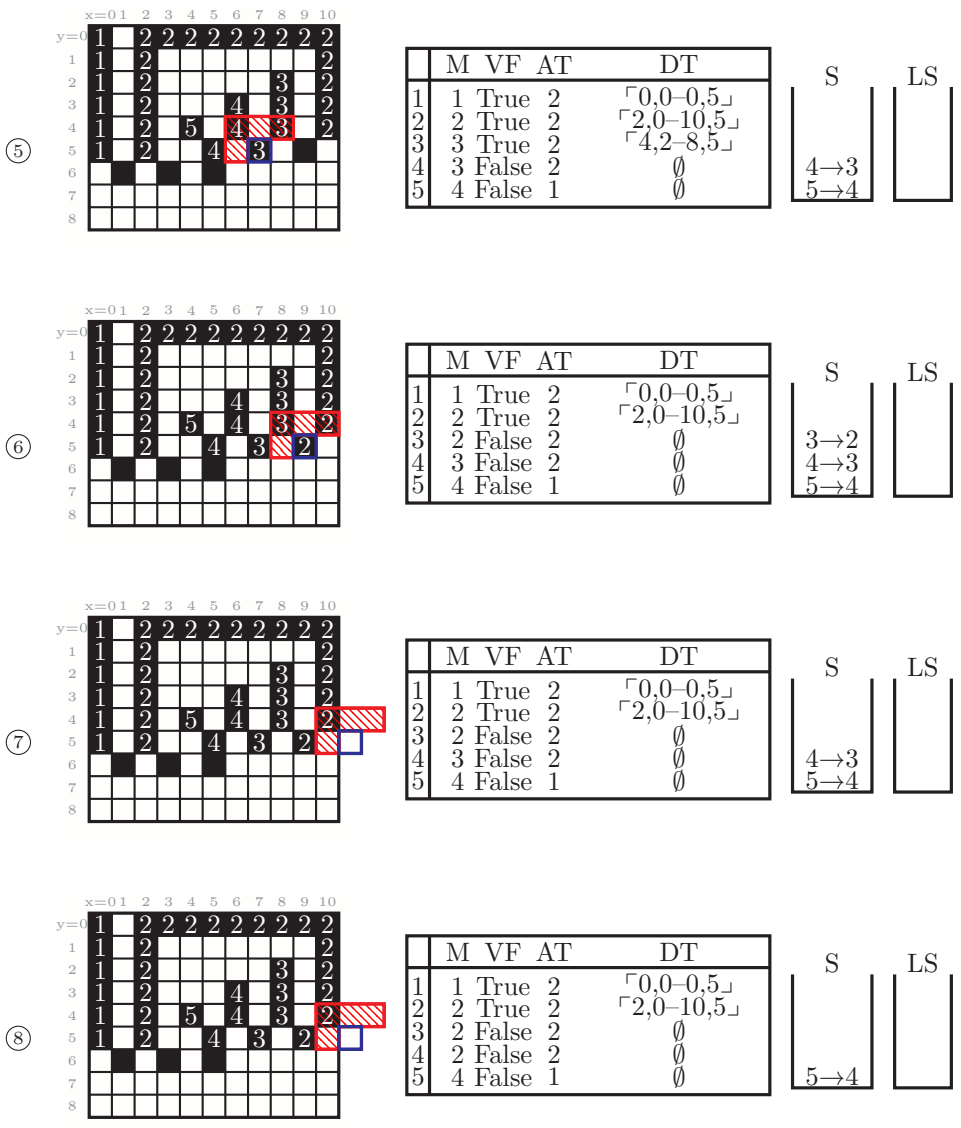
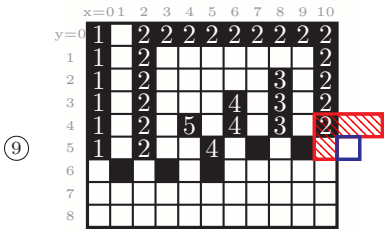


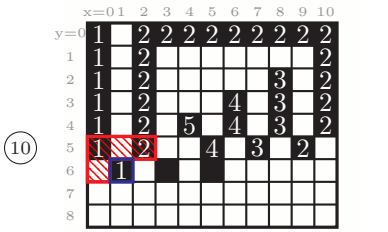

**Figure 2.9:** Step ⑤: Non-propagating merger operation. Step ⑥: Non-propagating merger operation. Step ⑦: First step of *flatten()*: Flattening of label 3. Step ⑧: Flattening of label 4.



| | M | VF | AT | DT |
|---|---|---|---|---|
| 1 | 1 | True | 2 | ⌜0,0–0,5⌟ |
| 2 | 2 | True | 2 | ⌜2,0–10,5⌟ |
| 3 | 2 | False | 2 | ∅ |
| 4 | 2 | False | 2 | ∅ |
| 5 | 2 | False | 1 | ∅ |

S LS

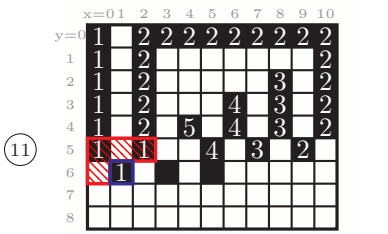
⑨



| | M | VF | AT | DT |
|---|---|---|---|---|
| 1 | 1 | True | 0 | ⌜0,0–10,6⌟ |
| 2 | 1 | False | 0 | ∅ |
| 3 | 2 | False | 2 | ∅ |
| 4 | 2 | False | 2 | ∅ |
| 5 | 0 | False | 1 | ∅ |

S LS

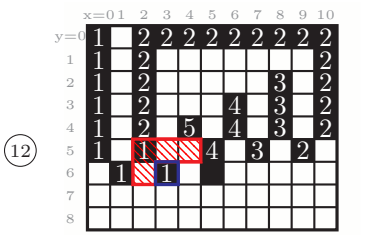
⑩



| | M | VF | AT | DT |
|---|---|---|---|---|
| 1 | 1 | True | 0 | ⌜0,0–10,6⌟ |
| 2 | 1 | False | 0 | ∅ |
| 3 | 2 | False | 2 | ∅ |
| 4 | 2 | False | 2 | ∅ |
| 5 | 2 | False | 1 | ∅ |

S LS

⑪



| | M | VF | AT | DT |
|---|---|---|---|---|
| 1 | 1 | True | 0 | ⌜0,0–10,6⌟ |
| 2 | 1 | False | 0 | ∅ |
| 3 | 2 | False | 1 | ∅ |
| 4 | 2 | False | 2 | ∅ |
| 5 | 2 | False | 0 | ∅ |

S LS

⑫

**Figure 2.10:** Step ⑨: Flattening of label 5. Step ⑩: Propagating merger operation (Table contens before assinging $L_c = L_{min}$ are shown). Step ⑪: Propagating merger operation (Table contents after assinging $L_c = L_{min}$ are shown). Step ⑫: Propagation of the previous $L_C$ in the next neighbourhood.

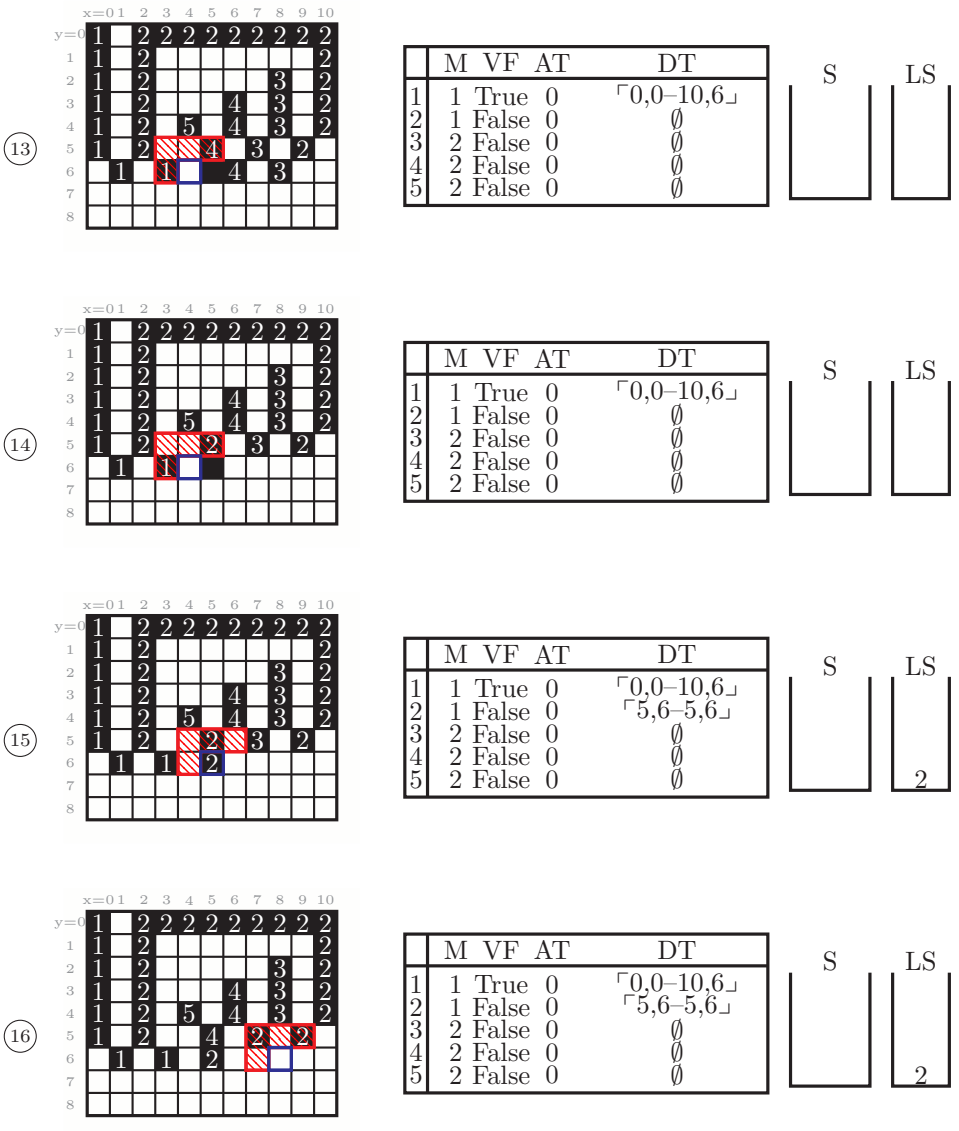



**Figure 2.11:** Step (13): Neighbourhood before assiging $L_c = M[L[C]]$. Step (14): Neighbourhood after assiging $L_c = M[L[C]]$. Step (15): Propagation of the previous $L_C$ in the next neighbourhood. Step (16): Detection that $LS.head = L[C]$.

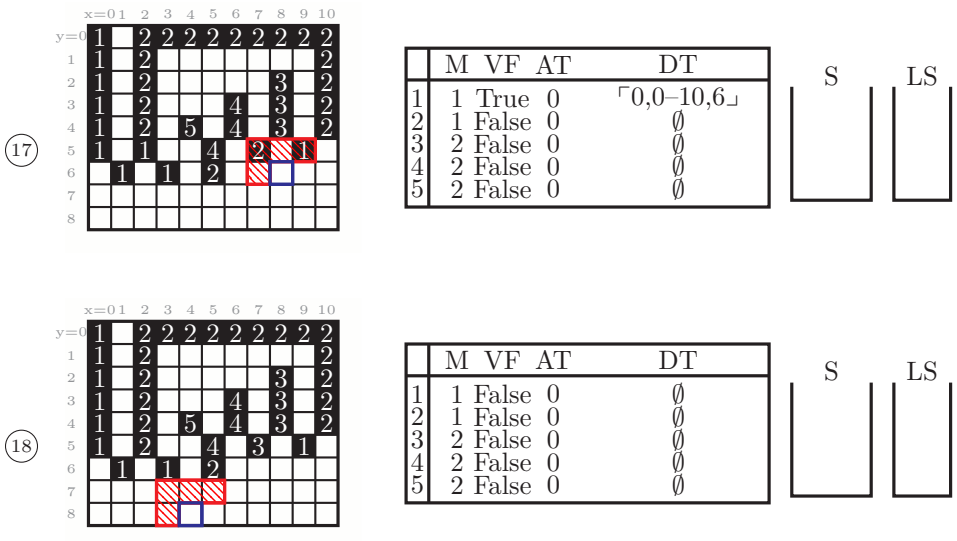



**Figure 2.12:** Step (17): *resolveStaleLabels*: Combination of feature from $DT[2]$ and $DT[1]$. Step (18): Read-out of finished feature vector of the finished connected component.

## 2.5 Experimental Results and Discussion

'Recently published CCA and CCL algorithms, such as the algorithms by *Lacassagne et al.* [14,76] and *He et al.* [48] are tailored to the cache hierarchy of general-purpose processors (GPP) which consist of several levels of on-chip and off-chip memory. For such processors, the average number of clock cycles to process a pixel of a random image is a meaningful metric to compare CCA or CCL algorithms [11]. However, to determine how well a CCA or CCL algorithm is suited to a hardware architecture, depends on the interaction of the algorithm with the basic building elements of the hardware device used (e.g. FPGA, ASIC, GPP) and the arrangement of these elements. Especially for FPGA architectures, the freedom to arrange the basic building elements (e.g. Registers, LUTs and BRAMs) of the hardware device facilitates the use of multiple levels of parallelism. As the use of multiple levels of parallelism accelerates processing of image data with CCA and CCL algorithms, processing bottlenecks or I/O bottlenecks are avoided, too.

The number and speed of lookup operations are crucial for carrying out CCA and CCL, as discussed in the introduction. Since hardware architectures realised on ASIC or FPGA do not have a fixed memory model, like a GPP, the three available memory types *on-chip registers*, *on-chip memory* and *off-chip memory* are arranged and connected to maximise lookup operations and to provide data at the exact time they are required. The bandwidth of on-chip registers and memory is significantly higher and the latency is significantly lower than off-chip memory; however, the number of on-chip memory bits is lower, as well. Therefore, *SLCCA* is designed to fit completely in on-chip registers and on-chip memories.

Unlike in a cache hierarchy where the cost of a read/write operation depends on the position of the data being accessed, this memory model provides random read/write operations at constant cost. Therefore, the total number of memory operations required to process an image provides a good prospect on how suitable a CCA or CCL algorithm is for a hardware architecture.

To compare different variants of CCA or CCL algorithms with different numbers of passes, different scan modes and different set merging algorithms, the number of *memory access instructions (MAIs)* are taken into account.

**Definition 20.** *Memory access instruction:* A memory access instruction is a single read access from or a single write access to an indexed data structure. '[67]

### 2.5.1 An Analysis of the Memory Access Instructions of *SLCCA* and State-of-the-Art Algorithms

'To compare the number of *memory access instructions* of the *SLCCA* algorithm to other CCA and CCL algorithms, the following cost metric is applied:

- Successive read instructions from the same position of a data structure are buffered in a register and are, therefore, counted as one *memory access instruction*.
- Successive write instructions to the same position of a data structure are buffered in a register and are, therefore, counted as one *memory access instruction*.
- Receiving the input image $I$ as a stream (as in the *SLCCA* algorithm) is not a *memory access instruction* per se, i.e. requires zero *MAIs*. Though, for a fair comparison, these read accesses are counted as one *MAI* each (effectively streaming from memory).

This metric does not try to show which CCA algorithm runs the fastest on an existing processing architecture such as a general purpose processor, but provides a prospect on the speed of a CCA or CCL algorithm when realised as hardware architecture. In fact, the results of [14] show that *Light Speed Labeling (LSL)* by *Lacassagne et al.* requires the smallest number of processing cycles per pixel on Intel and ARM processors.'[67]

The diagrams in Figure 2.13(a) to 2.13(e) depict the amount *memory access instructions* (*MAIs*) the algorithms *SLCCA*, *HCS*, *LSL*, *CT* and *RQU* require to extract the bounding box and area feature vectors for each connected component of the input image. *RQU* refers to Rosenfeld's classical algorithm [102] applying Quick-Union. The input image is a random image with an object pixel density from 0% to 100%. The different colours of the stacked diagrams in Figure 2.13 show the number of *MAIs* on each of the data structures used. The total number of *MAIs* is shown by the upper boundary of the diagrams.
All raster-scan algorithms (2.13a to 2.13d) access each pixel of the input image $I$ exactly once, as indicated by the brown bar at the bottom of the diagrams. Only *CT* has to access some pixels of $I$ multiple times due to the data-dependent contour tracing.
Feature vectors are stored in the data table *DT*. According to the metric defined at the beginning of this section, *DT* ($F$ in Figure 2.13b) is only accessed and updated at the end of a run of object pixels for raster-scan algorithms. Therefore, the access pattern of these algorithms in relation to the object pixel density is quite similar: For a blank image there is no *MAI* on *DT*; for 100% object pixel density the number of *MAIs* is twice the image height. This factor of two corresponds to one read and one write access on *DT* per row. Since the number of merger patterns in Figure 2.13 increases until 50% object pixel density is reached and decreases thereafter, the maximum number of *MAIs* on *DT* is around 50% object pixel density.

The *SLCCA* algorithm is designed to access all data structures (except for stack $S$) in parallel. The number of *MAIs* carried out simultaneously depends on the maximum number of *MAIs* on a single data structure of the set of parallel data structures.

In addition, the *MAIs* resulting from operations on stack $S$ have a sequential data dependency, and are executed after the parallel *MAIs*, mentioned before.

Discussion of the memory access instruction diagram of SLCCA

Figure 2.13(a) shows the *MAIs* on the data structures of *SLCCA*. The number of accesses on the merger table $M$ is equal to the number of pixels in the image. This is a consequence of the single lookup property of *SLCCA*. The number of *MAIs* on the reuse FIFO $R$ is highest around 50% of object pixel density. This is due to FIFO $R$ only being accessed when *new label patterns* or completed connected components are detected. Since *SLCCA* does not store a fully labelled image, the data structure $L$ containing one label for each pixel is only accessed for labels assigned to the last image row. These accesses to $L$ increase with the number of object pixels. Every label of $L$ is read once (except for those in the last row) independent of the input image $I$. Additionally, one write access is carried out on $L$ for every object pixel in $I$.
The amount of *MAIs* on the chain stack $S$ is highest for the worst case image which has an object pixel density of 60%, as shown in Figure 2.16(a). For all other examined random images, the number of *MAIs* on $S$ is below 1% of the total number of *MAIs* for processing these images, and is highest between 40% and 60% pixel density.
A *MAI* on $LS$ is only induced by a reachable stale label (see Equation 2.25), i.e. occurs very seldom in random images. However, it is required to process all possible patterns in $I$ correctly.

Discussion of the memory access instruction diagram of LSL

The diagram in Figure 2.13(b) shows the *MAIs* on the data structures of *LSL*. The data structures compared are: $ER$ which holds relative labels, $ERA$ which holds absolute labels, $RLC$ which holds run-length encoded image segments, $EQ$ which holds equivalence information and $F$ which holds feature vectors.

There is one write instruction on $ER$ for each pixel in the image and two read instructions for each run of object pixels in an image row. This is reflected in an increased number of *MAIs* on $ER$ of around 50%, and an equal number of *MAIs* at 0% and 100% object pixel density. The $ERA$ data structure is written once for each run of object pixels in an image row, and read when a merger pattern is detected. For each run of object pixels there is a write access on $EQ$ and two read accesses for each merger pattern detected in the image. The $RLC$ data structure, which stores the coordinates of the object pixel runs, is read twice and written once for each run of object pixels.
The access pattern to $ER$, $ERA$ and $RLC$ over the object pixel density is quite similar: There are much fewer runs and merger operations at 0% and 100% object pixel density, i.e. there are very few *MAIs*. The number of *MAIs* increases from a

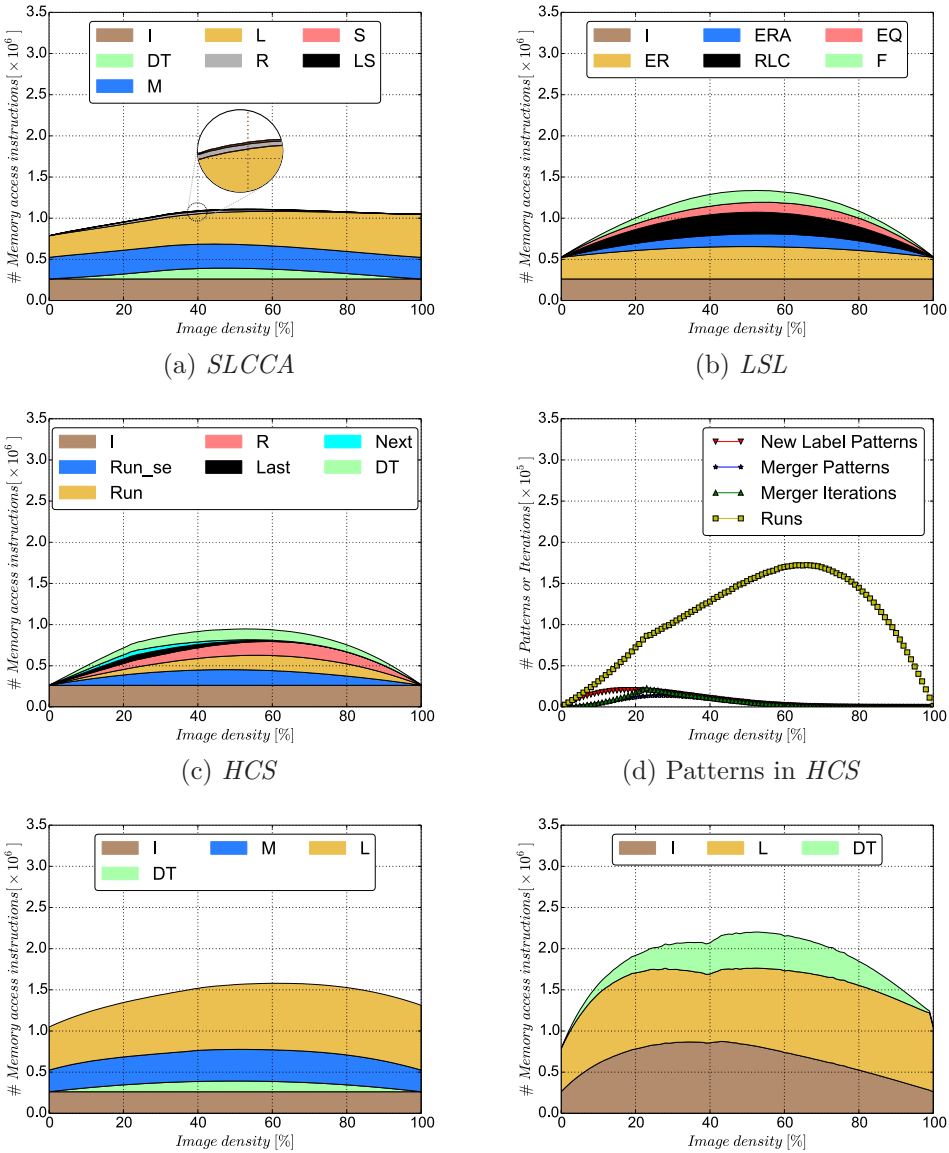



**Figure 2.13:** Memory access instructions on different data structures for random input images of size 512×512 of different pixel densities: (a) SLCCA [65], (b) LSL [76], (c) HCS [48], (e) RQU, (f) CT [17]. Sub-figure (d) shows the number of patterns and iterations as an aide to understand sub-figure (c).

minimum at 0% until it reaches its maximum at around 50% object pixel density. It then decreases at about the same rate until 100%. The number of *MAIs* at 0% and 100% object pixel density are approximately equal.

Discussion of the memory access instruction diagram of HCS

In the diagram in Figure 2.13(c), the number of *MAIs* on the data structures of *HCS* are depicted. The data structures compared are: $R$ which holds the representative label for each run, $Next$ which holds the label of the successor of each run, $Run\_se$ (a combination of $run\_s$ and $run\_e$ from [50]) which holds the horizontal start and end coordinate of each run, $Last$ which holds the label of last element in a chain of runs, $Run$ which holds the label of the parent of each run and $DT$ which holds the feature vectors.

Figure 2.13(d) shows the number of *new label patterns*, *merger patterns* and *runs* detected by *HCS*. For both diagrams, random images with the dimensions $512 \times 512$ pixels and 0% to 100% of object pixel densities are used. The number of *MAIs* on the data structure $Run$ increases with the number of runs in the input image $I$, as well as the number of *MAIs* on $R$, which contains the representative labels for each run. The *MAIs* on $Next$ and $Last$ are highest between 10% and 40% object pixel density. This is a result of a peak in the number of merger patterns in $I$, as shown in Figure 2.13(d). $Next$ and $Last$ are only accessed for new label patterns and when combining runs after detecting a merger pattern.

Discussion of the memory access instruction diagram of RQU

The diagram in Figure 2.13(e) shows the *MAIs* on the data structures of *(RQU)*.
In the first pass of this two-pass algorithm, one write instruction is carried out for each *new label pattern* on the merger table $M$. For each *merger pattern*, one write and at least two read instructions are performed on $M$. There are more read instructions dependent on the height of the trees in the union-find data structure stored in $M$, which are a result of the pixel patterns in $I$. To find the representative labels for each pixel in the second pass, at least one read instruction on $M$ is required for each object pixel of $I$. The majority of the accesses on $M$ is independent of the pixel patterns in $I$. The increasing number of *merger pattern* around 50% object pixel density is also reflected in the number of *MAIs* on $M$, as shown in Figure 2.13(e).
There are up to three read instructions per pixel on the labelled image $L$ in the first pass: zero *MAIs* for each background pixel in $I$, three for each *merger pattern* and one for *new label patterns* or *label copy patterns*.
In the first pass, there is one write *MAI* on $L$ independent of the pixel pattern in $I$. This corresponds to the number of *MAIs* at 0% object pixel density. For each object pixel in $I$, another write access on $L$ is carried out. In the second pass, there is one read access on $L$ per pixel to check if it is background. Another write access

per object pixel replaces each foreground label with its representative label. Since $RQU$ is used for feature extraction, the second pass is important to determine the representative label of each pixel and for accumulating the feature vectors in $DT$, however a fully labelled image is not necessary. The write instruction in the second pass is, therefore, not considered for fairness in the comparison. The number of *MAIs* on $L$ shown in the diagram of Figure 2.13(e) increase continuously from 0% up to a density of approximately 60%, and decreases thereafter due to the decreasing number of merger patterns. At 0% density and 100% density the number of *MAIs* on $M$ is equal. The number of *MAIs* on $L$ at 100% density is 50% higher than at 0% density due to the read access in the second pass.

Discussion of the memory access instruction diagram of CT

The $CT$ algorithm [17] starts reading the input image $I$ in raster scan order. This results in the number of *MAIs* on $I$ to be equal to the number of pixels in the input image. This is most obvious for the number of *MAIs* on $I$ in the diagram in Figure 2.13(f) for 0% and 100% object pixel density. The *MAIs* on $I$ exceeding this level are from tracing contours and are highest around 40% of object pixel density. The *MAIs* on $L$ consist of two parts: label propagation and contour tracing. The number of *MAIs* increases with the number of object pixels in $I$, which is clearly visible by comparing the number of *MAIs* on $L$ at 0% and 100% object pixel density. The *MAIs* on $L$ also depend on the length of the contours: the longer a contour is, the more *MAIs* on $L$ are required to trace it. Since the total number of pixels in the image is constant, the average area of the associated connected components declines with the length of its contours in the image. A result of this might be the almost constant number of *MAIs* above 40% object pixel density. Features are extracted by following the extracted contours from an image. For every pixel of a contour, $DT$ is accessed once for reading and once for writing.

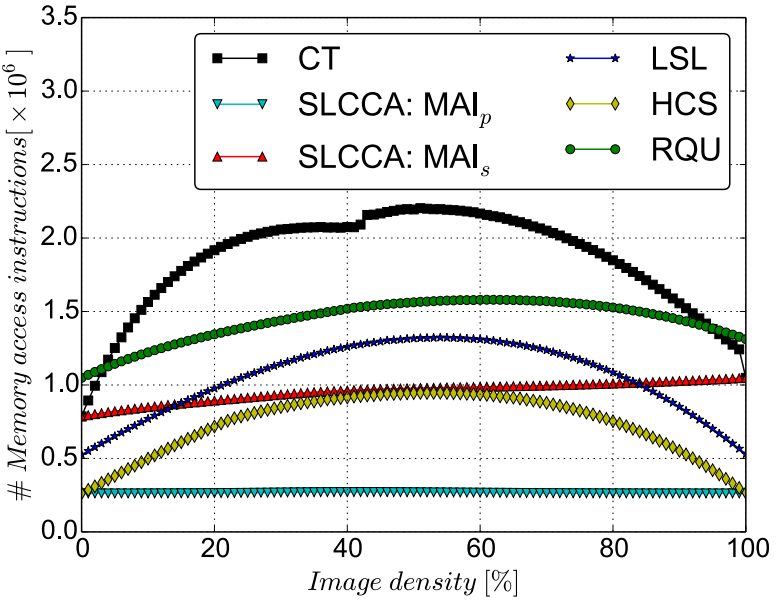



**Figure 2.14:** Comparison of the number of memory access instructions (MAIs) for processing random images with different object pixel densities. $MAI_s$: sum of MAIs. $MAI_p$: number of parallel MAIs. *[67]*

### 2.5.2 Comparison of the Memory Access Instructions of *SLCCA* to State-of-the-Art Algorithms

'Figure 2.14 depicts the number of *MAIs* for extracting the feature vectors of the connected components in a random image of $512 \times 512$ pixels with *SLCCA*, *LSL* [76], *HCS* [48], *CT* [17] and *RQU* [102]. Both *HCS* and *LSL* encode and process pixel runs from the input image, which explains the large difference of *MAIs* between an empty/filled image and an image with object pixel density around 50%. The number of MAIs of *SLCCA* increases with the object pixel density, since it processes pixel by pixel instead of runs. In the diagram in Figure 2.14, $MAI_s$ shows the *sum* of MAIs on all data structures and $MAI_p$ shows the number of *parallel* MAIs for *SLCCA* processing random images of different object pixel densities. The MAIs in *HCS*, *LSL*, *CT* and *RQU* have serial data dependencies, i.e. the total number of MAIs on their data structures are used for the comparison. The number of parallel MAIs required for *SLCCA* is almost constant for processing random images of diffent object pixel densities, as all memory accesses except for reading the stack $S$ can be performed in parallel. The sequential read accesses on stack $S$ are at most 3% of the *parallel* MAIs for random images with an object pixel density around 40%. For *HCS* the number of MAIs is highest at 55% and for *LSL* it is highest at 56%, i.e

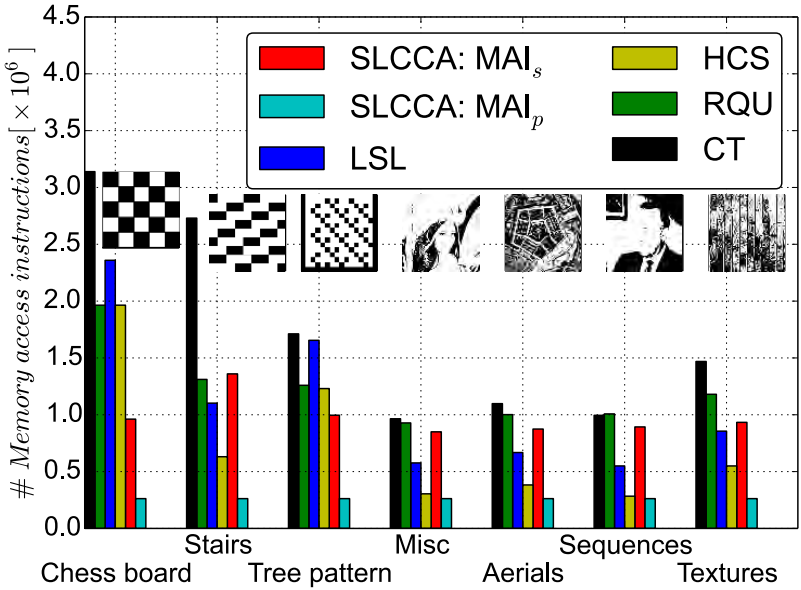
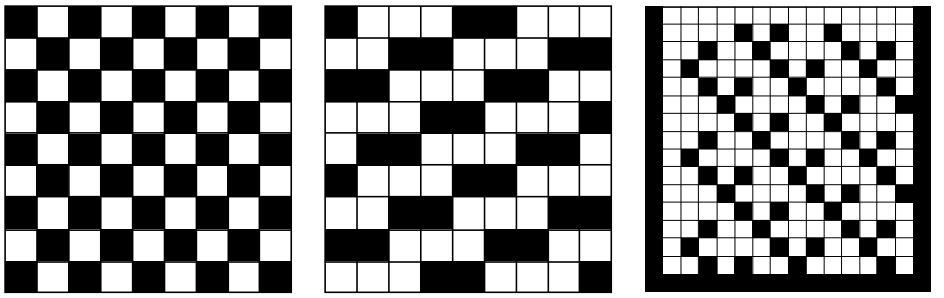



**Figure 2.15:** Comparison of the MAIs for worst case images and reference image from *SIPI* database [124]. $MAI_s$: sum of MAIs. $MAI_p$: number of parallel MAIs.

*SLCCA* reduces the number of MAIs by a factor of 3.5 compared to *HCS* and by a factor of 5 compared to *LSL* for the chosen image size.'[67] Since the assumptions taken for measuring *memory access instructions* are based on the memory model of hardware architectures it is expected that this is close to the actual speedup which can be achieved realising these CCA algorithms as hardware architectures. 'The bar diagram in Figure 2.15 shows the number of MAIs required for processing chess board patterns, stair pattern and tree pattern images [26], as well as for processing natural images from the *USC-SIPI* database [124].

In the scope of the explored images, the chess board pattern of pixel granularity one has been shown to require the maximum number of MAIs for *LSL*, *HCS* and *RQU*. The tree pattern was identified as the worst case pattern for *HCS* with respect to the run time on a GPP [26]. However, for all examined image sizes *HCS* requires fewer MAIs for processing the tree pattern than for the chess board pattern. The stair pattern from Figure 2.16 requires the maximum number of MAIs for *SLCCA*. To compare the minimal guaranteed processing time, the worst case pattern of each algorithm is considered. Therefore, the number of MAIs *SLCCA* required to process stair pattern images is compared to the other algorithms processing a chess board pattern and shown in Figure 2.15. Table 2.3 shows the number of MAIs for worst case patterns normalised to the number of parallel MAIs in *SLCCA*. This shows



(a) (b) (c)

**Figure 2.16:** (a) Chess board pattern, (b) Stair pattern [7] and tree pattern [26].

| Algorithm | Worst case pattern | # MAIs normalised |
|---|---|---|
| SLCCA: parallel *MAIs* | stairs | 1.0 |
| SLCCA: sum of *MAIs* | stairs | 5.18 |
| LSL | chess board | 8.99 |
| HCS | chess board | 7.49 |
| RQU | chess board | 7.99 |
| CT | chess board | 11.97 |

**Table 2.3:** Comparison of MAIs for worst case patterns. *[67]*

that *SLCCA* reduces the number of *memory access operations* for processing worst case images by a factor of 7 compared to the other algorithms.'[67]

The diagrams in Figure 2.17 show the scalability of the previously examined CCA and CCL algorithms for the *chess board pattern*, the *stair pattern*, and the *tree pattern* (Figure 2.16) for image sizes from 1 Megapixel to 80 Megapixels. The resulting diagrams suggest that the number of *MAIs* for processing these images size scale linearly in the number of pixels for the algorithms examined in Figure 2.17. Figure 2.17(a) & (c) show that *SLCCA* requires the lowest number of *MAIs* for processing the chess board and tree pattern. For processing the stair patterns, which is the worst case image for *SLCCA*, the number of parallel *MAIs* is still 50% below the second best algorithm, *HCS*, for this pattern. This is shown in Figure 2.17(b).

For each of the discussed algorithms, Figure 2.17(d) shows the ratio of *MAIs* required to process its worst case pattern for image sizes from 1 Megapixel to 80 Megapixels. It is normalised to the number of parallel *MAIs SLCCA* requires to process an image with a stair pattern. This diagram shows that *SLCCA* requires the fewest amounts of *MAIs* for processing its worst case image among the examined CCA and CCL algorithms.

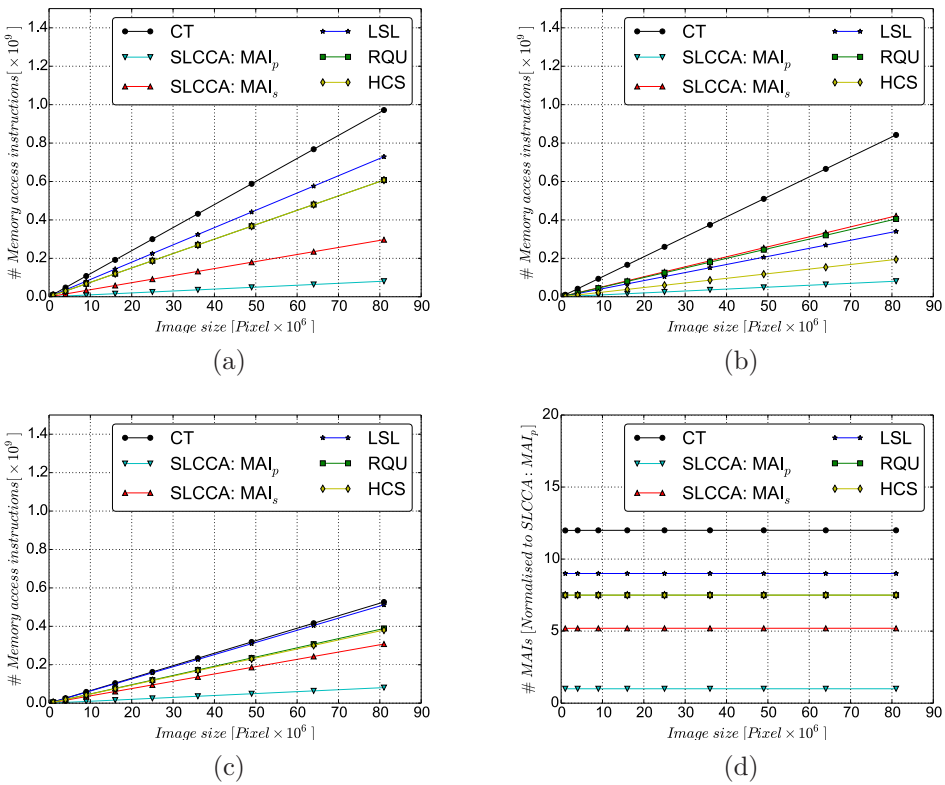



**Figure 2.17:** Comparison of scalability of the number of MAIs for increasing image sizes for (a) chess board pattern, (b) stairs pattern and (c) tree pattern. The diagram in (d) shows the number of MAIs (for the algorithms' respective worst case) normalised to *SLCCA:* $MAI_p$ for image sizes up to 80 Megapixel. $MAI_p$: number of parallel MAIs.

## 2.6 Summary and Contributions of the SLCCA Algorithm to the State of the Art

The single lookup CCA algorithm, *SLCCA*, which was introduced and described in detail in this chapter is an improvement to the state of the art on several levels, as pointed out in the following.

- **Reduction of number of memory access instructions (*MAIs*):** *SLCCA* uses an improved union-find algorithm allowing the find operation to be replaced by single lookups. This reduces the number of memory accesses by a factor of 3.5 for random images and by a factor of 7 for worst case images compared to other state-of-the-art CCA/CCL algorithms.
- **Using a novel control structure to detect the last pixel of an image object in an image stream at the earliest possible point in time:** This allows to reduce processing latency and allows the memory resources used by an object to be freed earlier which contributes to reducing memory resources.
- **A novel label recycling scheme reducing the number of label lookups per pixel:** The label translation scheme of [83] is simplified by reducing the number of lookups from two to one per label.
- **A method for out-of-order labelling for the efficient recycling of labels:** As a consequence of the novel label recycling, augmented labelling is introduced, a technique to build consistent rooted tree data structures for components labelled out-of-order.
- **Detection and correct processing of image patterns not considered in previous publications:** At the algorithm level, image patterns (e.g. Figure 3.10) were not taken into account in previous hardware architectures [7, 58] resulting in incorrect labelling. In the *SLCCA* algorithm (laying the foundation for the SLCCA architecture in Chapter 3), these patterns are detected and handled correctly, i.e. arbitrary image patterns can be analysed.

# 3 Hardware Architecture of SLCCA

In this chapter, a hardware architecture for performing connected components analysis (CCA) on a high-speed pixel stream is presented which uses the *SLCCA* algorithm introduced in Chapter 2. The content in this chapter marked by single quotation marks are from [65].

'Increasing image resolutions beyond *high-definition* (2 $Megapixel$) in consumer electronics [55] and frame rates above 100 $fps$ in high speed imaging [77] require high-performance hardware architectures. For connected components analysis, a number of optimised hardware architectures and software implementations have been proposed in the last five years, all with the goal of avoiding the performance bottlenecks due to memory resources or memory bandwidth [33, 51, 71, 76, 99, 136, 138].

For previous hardware CCA architectures, the required resources are proportional to the image resolution [7]. In the proposed architecture of the *SLCCA* algorithm in this chapter, the required memory resources are proportional to the image width. This directly affects the throughput that can be achieved with a certain architecture or hardware device. Any reduction in the hardware resources allows better performance to be achieved with the same hardware device or allows a switch to a more energy-efficient or less expensive hardware device.'[65] This chapter presents a dedicated hardware architecture for *SLCCA* which reduces the hardware resources required compared to other CCA or CCL hardware architectures significantly and enables extracting image features from *HD* and ultra-high-definition *UHD* image streams with 42% less hardware resources (depending on the features extracted) than state-of-the-art CCA hardware architectures [83]. 'Based on these savings, it is possible to realise an architecture processing video streams of larger images sizes, or to use a smaller and more energy-efficient hardware devices, or to increase the functionality of already existing image processing pipelines in reconfigurable computing and embedded systems.'[65]

## Dedicated CCA Hardware Architecture vs. Software Implementation

'A challenge for the optimisation of CCA is that most algorithms are sequential and consist of a combination of compare, lookup and control operations [51]. A label is assigned to every pixel depending on its neighbourhood's labels. This data dependency on the current pixel's predecessors makes parallelisation non-trivial,

but pipeline processing possible. From these data dependencies, it follows that all operations for the currently processed pixel have to be finished before the operations on the subsequent pixel can start. Therefore, the throughput depends on the execution time of the individual operations. Processing the pixel data of an image in real-time as it is streamed from an image source requires a high-throughput architecture, especially when a high-speed image sensor is used. Carrying out CCA on a general purpose processor (GPP) with a multi-core architecture requires a sequential execution of the comparison and control operations and several memory operations per pixel. If the size of the data structures exceeds the size of on-chip memory of the GPP, slow off-chip memory has to be utilised. The execution time therefore can be dominated by the latency of the memory operations [10] limiting the overall throughput of the CCA algorithm and making the performance strongly dependent on the input data. Making use of a single-pass CCA algorithm on common GPP architectures might allow the required data structures to be stored in on-chip memory and solves the problem of memory size. Nevertheless, the available on-chip memory bandwidth is usually shared among several processing cores reducing the performance for parallel memory access [43] which might limit the throughput. General purpose processors (GPP) are only a good choice for CCA or CCL as long as power dissipation or processing latency is of minor concern. Then, a high throughput and good scalability can be achieved by distributing the workload over a set of several GPP or GPGPU systems by either distributing parts of the pixel stream or assigning each image of the stream to a separate processing unit. In contrast, when using a dedicated hardware architecture for CCA, all combinational operations for processing a single pixel of the image can be carried out in a single clock cycle, some of them in parallel. Several on-chip memory structures ensure a low latency read and write of image labels at high bandwidth. This allows faster processing of a single pixel, leading to a high processing throughput with low latency. The realisation of a dedicated hardware architecture is possible either as an application specific integrated circuit (ASIC) or on a field-programmable gate array (FPGA). Compared to a GPP architecture, both alternatives are typically superior in terms of power dissipation, which is especially important in embedded and mobile applications. Recent reconfigurable logic devices, FPGAs, consist of lookup tables (LUTs), registers and on-chip block-RAMs (BRAMs), which can be connected via a user-programmable connection network [3, 134]. For CCA, decisions and control operations are mapped to LUTs; for each operation requiring memory access a dedicated on-chip BRAM is assigned. The architecture proposed in this chapter is customised for (but not limited to) a realisation as a hardware architecture on an FPGA. A speed-up is gained by distributing the computations to several pipeline stages working in parallel. The memory bandwidth is achieved by distributing the memory operations over several on-chip BRAMs. A high throughput by pipeline processing requires each pipeline stage to have a constant execution time to be able to keep up with the bandwidth of the image source. The goal of the processing architecture is to achieve a throughput of one pixel per clock cycle while maximising the clock frequency. When using BRAM resources having one clock cycle latency

to represent data structures (e.g. digraphs), only one lookup per clock cycle is possible to achieve a throughput of processing one pixel per clock cycle. Recent CCA hardware architectures, as well as the architecture proposed in this chapter, are close to processing one pixel per clock cycle. This is achieved by maintaining rooted trees with a height of one for labels to be processed in the current row. In the proposed architecture, labels already processed in the current image row may have a bigger tree height. A tree height of one at the beginning of the next image row is achieved by compressing the tree structure at the end of each row [7, 83]. This reduces the number of lookup operations to one lookup per clock cycle plus a maximum overhead of 17% at the end of the image row for tree compression, as shown in Section 3.1.2. '[65]

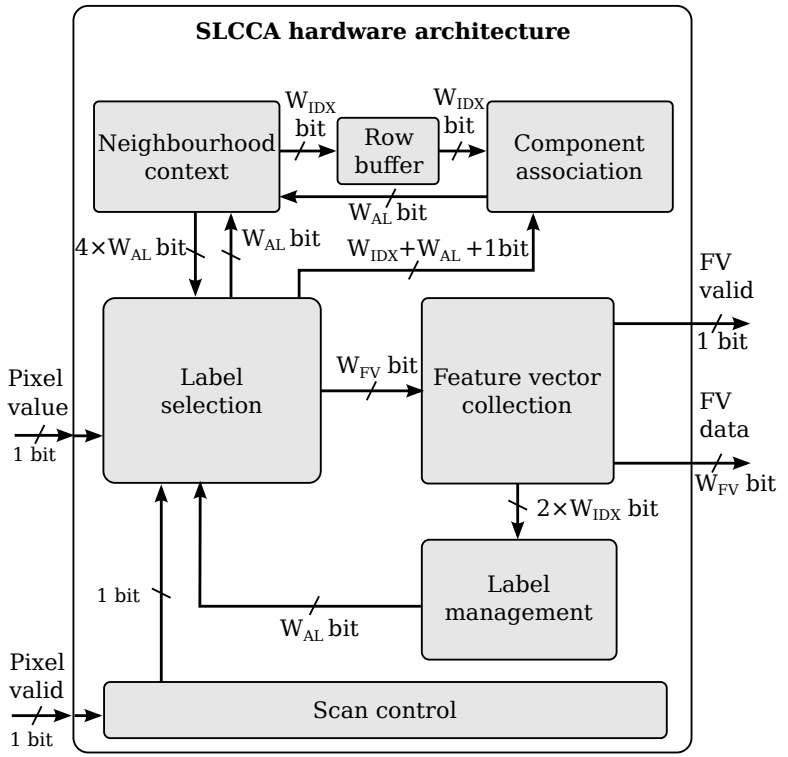
| Abbreviation | Name |
|---|---|
| $AT$ | Active tags |
| $DT$ | Data table |
| $FV$ | Feature vector |
| $H$ | Image height |
| $I$ | Source image |
| $L$ | Labelled image |
| $LS$ | Label stack |
| $M$ | Merger table |
| $N_L$ | Number of labels |
| $N_M$ | Number of merger patterns per row |
| $R$ | Reuse FIFO |
| $RB$ | Row Buffer |
| $S$ | Stack |
| $TT$ | Translation Table |
| $VF$ | Valid flags |
| $W$ | Image width |
| $W_{IDX}$ | Width of a label index |
| $W_{AL}$ | Width of an augmented label |
| $W_{FV}$ | Width of a feature vector |



**Table 3.1:** Nomenclature used in this Chapter. *[65]*

## 3.1 Design of the Hardware Architecture

'In the following, the nomenclature defined in Table 3.1 is used. The top level block diagram of the *SLCCA* architecture is depicted in Figure 3.1. It processes a binary input image $I$ in forward raster scan order, as shown in Figure 3.2. The input image $I$ is of size $W \times H$ and consists of *object pixels* and *background pixels* represented by 1 and 0. Two object pixels $p_1$, $p_2$ are connected if one pixel is in the other pixel's 8-neighbourhood or a path of neighbouring object pixels between $p_1$ and $p_2$ exists. A set of object pixels of $I$ is called a connected component if every pair of pixels in the set is connected. A subset of connected object pixels of a connected component is called a *component segment*. The *feature vector* (FV) of a connected component or component segment is an n-tuple composed by functions of the component's pattern [45]. Connected components analysis is concerned with deriving the feature vector for each connected component from the binary input image $I$. The hardware architecture associates every pixel with its connected component by assigning a label and extracts the component's feature vector. Label 0 is reserved for background



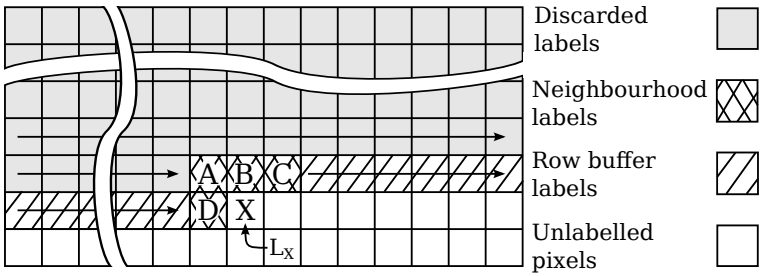

[65] ©2015 IEEE

**Figure 3.1:** Block diagram of the *SLCCA* hardware architecture for connected component analysis. *[65]*

pixels. A connected component is *completed* when a label has been assigned to all of its associated pixels. This is detected as defined in Equation 2.28.

For the selection of the label $L_X$ assigned to the current pixel $I[X]$ at position $X$, the *neighbourhood context* provides the labels at position $A$, $B$, $C$ and $D$ labelled $L_A$, $L_B$, $L_C$ and $L_D$ respectively, as depicted in Figure 3.2. To simplify the discussion, $L_{AorD}$ is introduced to refer to the label $L_A$ or label $L_D$ since if $I[A]$ and $I[D]$ are both object pixels they will always have identical labels: $L_{AorD} = L_A = L_D$. If a label is used as a Boolean variable in the following, *true* indicates a label associated with an object pixel and *false* indicates a label associated with a background pixel.

There are three *label patterns* with either zero, one or two different object labels in the *neighbourhood context* of an object pixel which are referred to as *new label pattern*, *label copy pattern* and *merger pattern*, respectively. These patterns are handled by the *new label operation*, the *label copy operation* and the *merger operation* which change the content of the data structures in the neighbourhood context, the component




[65] ©2015 IEEE

**Figure 3.2:** The four different groups of image labels. *[65]*

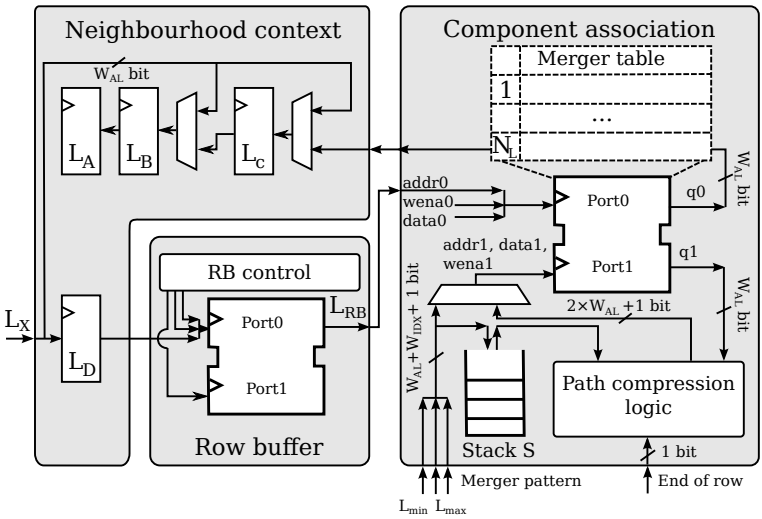
association and the feature vector collection. Two different object labels in the neighbourhood context where $I[X] = 1$ obviously belong to the same connected component. For this case, the merger pattern induces a merger operation: the smallest object label of the neighbourhood context is assigned to $L_X$ and merging labels are stored on the *merger table M* (see 3.1.2). A merger operation on two labels $l_0$ and $l_1$ is referred to as merging $l_0$ and $l_1$. A merger operation on two component segments $s_0$ and $s_1$ of the same connected component is referred to as merging the component segments $s_0$ and $s_1$. The feature vector associated with each label is stored in the *data table DT* and is updated every time its object label is assigned to $L_X$. The new label operation and the label copy operation are discussed in Section 3.1.2. '[65]

### 3.1.1 Neighbourhood Context and Row Buffer

'The *SLCCA* architecture is based on the single-pass CCA algorithm from [7]. For this single-pass CCA algorithm, the decision of which label to assign to $L_X$ only depends on the labels of the previous and the current image row, in particular the labels of the current row left of $X$ and the labels from $A$ to the end of the previous row. This architecture distinguishes between four different types of labels, as shown in Figure 3.2:

- *Neighbourhood labels* (cross-hatched) $L_A$ through $L_D$ are required in the current clock cycle to determine the current pixel's label $L_X$.
- *Row buffer labels* (hatched) are required for labels associated with pixels processed in subsequent clock cycles.
- *Discarded labels* (marked grey) which are not required for further decisions.
- *Unlabelled pixels* (marked white) which have not been processed yet.

Figure 3.2 shows the source image $I$, where all pixels before $X$ are already processed in raster scan order. Only the neighbourhood labels and the row buffer labels are




[65] ©2015 IEEE

**Figure 3.3:** Architecture of neighbourhood context, row buffer and component association unit at register-transfer level. *[65]*

relevant to determine the label $L_X$ and must be stored for processing subsequent image rows. Since no labelled image is saved, the label of the current pixel is stored on the *row buffer RB* for one image row until it is required again for the decision process in the row below. The output of *RB* is connected to and addresses the merger table discussed in Section 3.1.2.

To parallelise and effectively accelerate the label selection, simultaneous read and write access to all labels of the neighbourhood context is required. This is realised by using a register for each of the labels $L_A$ to $L_D$. After a merger operation, an update of $L_B$ and $L_C$ is required. The next cycle's $L_B$ is assigned the current label $L_X$. When the next cycle's $L_C$ is an object pixel it needs to be updated in case of a merger operation when $I[x+2, y-1] = 1$. These updates are realised by multiplexers at the input of the registers. The size of the row buffer depends on the image width $W$. A label added to the row buffer is not accessed for $W-1$ cycles, i.e. it does not need to be read for the duration of the processing of one image row. This allows a realisation as a dual-port block-RAM (BRAM). Figure 3.3 shows the architecture of the *neighbourhood context* and *row buffer* on the register-transfer level.'[65]

### 3.1.2 Label Selection and Image Component Association

'The *label selection* unit assigns the minimum object label of the neighbourhood context to $L_X$ and generates control signals to update tables of the *component association* unit. When processing the image pixels in raster scan order, different

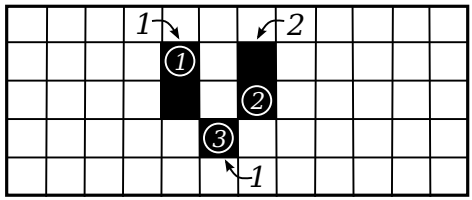





**Figure 3.4:** This example image contains patterns inducing a new label ①, label copy ② or merger ③ operation. After the merger pattern at position ③ the merger table entry of 2 points to 1. *[65]*

initial labels may be assigned to different component segments of a connected component. The merger table $M$ is used as introduced in Section 2.1. Every connected component and component segment is identified by the *root label* of its tree structure, which points to itself in $M$.

If the current pixel is an object pixel and all neighbour labels are background, a new label $l$ is assigned to $L_X$ and the merger table entry of $l$ is initialised to point to itself, i.e. $M[l] \coloneqq l$. A label copy operation assigns the object label in the neighbourhood to $L_X$. In the neighbourhood context of a merger pattern $L_{AorD} \neq L_C$. To label each pixel correctly, the minimum label $L_{min} = min(L_{AorD}, L_C)$ is assigned to $L_X$ [7]. All pixels labelled $L_{max} = max(L_{AorD}, L_C)$ processed before a merger pattern are already added to the row buffer $RB$ and cannot be changed immediately, therefore the merger table entry of $L_{max}$ is set to point to $L_{min}$, i.e. $M[L_{max}] \coloneqq L_{min}$ which makes $L_{min}$ the component segment's root label.

The merger table $M$ is realised as a BRAM operated in true-dual port mode. One port is used as a read port to look up the labels at the output of the row buffer; a merger operation updates the rooted tree data structure $F$ stored in $M$ via the second BRAM port. This enables continuous lookups in every clock cycle for the labels at the output of the row buffer, while the rooted tree data structure in $M$ can be updated simultaneously via the write port.

All not circled numbers in the figures used for the following examples represent the labels after the operations of the corresponding image pattern are carried out. Circled numbers in the examples, like ①, refer to positions in the image where patterns induce operations. In Figure 3.4 at position ① through ③ an example for a new label pattern at ①, a label copy pattern at ② and a merger pattern ③ are given.

A series of merger patterns of the same connected component where $L_C < L_{AorD}$ creates a label *chain* in $M$, representing a path in the rooted tree, so that the labels stored in the row buffer do not yield the root label with a single lookup in $M$. A *chain* is resolved by joining all non-root labels of the chain with the root label of the

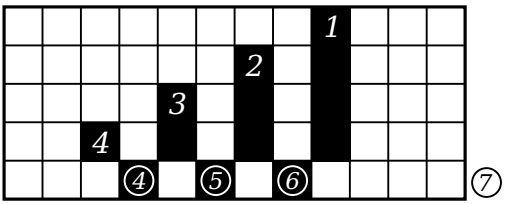





**Figure 3.5:** Image with *chain* pattern. By saving the label pair of a merger operation on the stack $S$, the content of $M$ $(4 \to 3 \to 2 \to 1)$ is updated at the end of the image row. Then the content of $M$ is $4 \to 1$, $3 \to 1$, $2 \to 1$. *[65]*

chain. A reverse scan over the chains is executed to compress the tree in the merger table $M$ to a height of one. In union-find algorithms this operation is called path compression [53]. The label pairs consisting of $L_{min}$ and $L_{max}$ for the reverse scan are pushed on the *stack* $S$ by every merger operation where $L_C < L_{AorD}$ during the scan of the image. For every pair of labels popped off the stack $S$ at the end of the row, the merger table is updated with $M[L_{max}] \coloneqq M[L_{min}]$ which eventually resolves the chains.

An example of an image pattern resulting in a chain is illustrated in Figure 3.5. The chain generates 3 stack entries. The labels in the neighbourhood context of positions ④ to ⑥ lead to merger operations linking the component segments initially labelled 1 to 4 and push the label pairs $(3, 4)$, $(2, 3)$ and $(1, 2)$ on the stack $S$. At the end of the image row (position ⑦) the merger table $M$ contains a chain where label 4 points to label 3, label 3 to 2 and label 2 to 1. For labels 3 and 4, a single lookup in $M$ does not yield their component segment's root label 1 because of the chain $(4 \to 3 \to 2 \to 1)$. By popping the stack entries off $S$ in reverse order, the content of $M$ is compressed by updating the merger table entries of all labels to point to the root label 1.

The processing of each stack entry consists of one read operation from the stack $S$ (Equation 2.22), one read and one write operation from merger table $M$ (Equation 2.23) requiring in total three clock cycles. These operations can be pipelined, and with a dual-port BRAM for the merger table require on average one clock cycle per update [7]. A maximum chain consisting of $\lceil \frac{W}{2} \rceil$ merger patterns is possible in one image row, therefore the stack depth is $\lceil \frac{W}{2} \rceil$. The image pattern creating $\lceil \frac{W}{2} \rceil$ merger operations is not the pattern that generates the maximum number of stack entries averaged over the whole image. The worst case creates a pattern for which a merger operation is carried out for every $5^{th}$ pixel of every row. On average, for processing this worst case image, the number of stack entries after each image row is $\lceil \frac{W}{5} \rceil$ reducing the average throughput by 17% [7]. If the image source inserts a sufficiently long gap between two rows (e.g. the blanking period of an image sensor), then real-time processing is possible. Otherwise, a buffer for the pixel stream at the

input to cover bandwidth peaks allows real-time processing. For the stack $S$, write access is required during processing an image row and read access is necessary at the end of the row making the realisation as a single port BRAM sufficient.'[65]

### 3.1.3 Label Recycling and Feature Vector Collection

'In previous CCA architectures, the memory requirements of $M$ and $DT$ are proportional to the image area. This is because in a worst case image, a quarter of the pixels can be different connected components [7, 58]. However, at any time in the raster scan, the number of different labels assigned to pixels in an image row is only proportional to the image width [82], [62]. Memory requirements can be significantly reduced by recycling labels no longer in use, enabling entries of $M$ and $DT$ to be reused after a connected component is completed. The *label management* unit keeps a record of the unused labels on the *label reuse FIFO* $R$ , which is initially filled with all labels, one through $\lceil \frac{W}{2} \rceil$. For each new label operation, an entry is read off $R$ and assigned to the current pixel. For the reuse of already completed connected components two scenarios have to be distinguished:

- Recycling $L_{max}$ after a merger operation
- Recycling the label of a completed connected component

The recycling of $L_{max}$ requires that $L_{max}$ is not contained in the row buffer anymore. To be sure that $L_{max}$ is not in the row buffer anymore when it is reused, the label management unit has to delay the reuse of these labels until $W$ more pixels have been processed.

Each connected component keeps its label until it is completed. To detect completed connected components, every label which was assigned to a pixel in the previous row, but is not assigned to $L_X$ in the current row, belongs to a completed object, i.e. the label and the associated memory resources can be reused immediately.

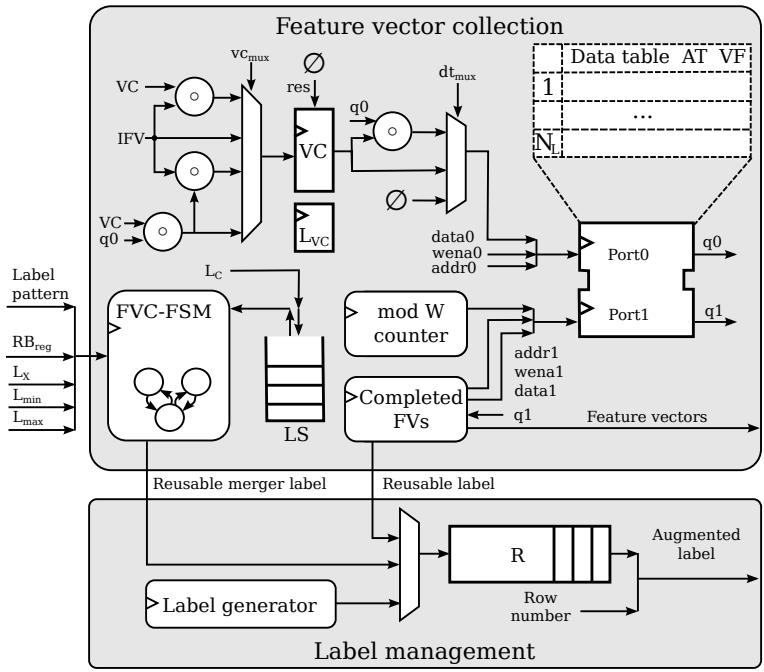
The position in the image where a label is recycled and added to $R$ depends on the input image $I$, therefore the recycled labels on $R$ are not necessarily in numerical order. The property from [7] that a merger operation always chooses the minimum label for $L_X$ can therefore result in a corrupted merger table $M$ not reflecting the actual label associations of the image. The image in Figure 3.9 (page 87) demonstrates a case in which several merger operations create two root labels for a single connected component by always assigning the minimum label to $L_X$. To avoid this case, the concept of *augmented labels* ($AL$) is introduced. Labels are augmented with the row number they are generated in, i.e. each label is a two-tuple consisting of row number ($rw$) and the index ($index$). The elements of this two-tuple are referenced by name in the following: the notation $L_X.index$ refers to the index of label $L_X$ and $L_X.rw$ refers to the row number of label $L_X$. The $rw$ element of a label is used to determine which label to associate with the current pixel: $L_{min} := L_C$ when $L_{AorD}.rw > L_C.rw$, else $L_{min} := L_{AorD}$. The $index$ element

is used to access tables such as the merger table $M$, e.g. $M[L_X]$ is realised as $M[L_X.index]$. Examples for augmented labels are given in Table 3.2. Augmented labelling ensures that a merger operation always assigns $L_{min}$ to the label created earlier in the scan process. With augmented labels, the rooted tree data structure in $M$ always correctly reflects the current structure of not yet completed connected components at the current position in the raster scan. The index of an augmented label of a completed connected component is written to the label reuse FIFO $R$. The index of $L_{max}$ of a merger pattern is added to $R$ by a merger operation. The condition to delay reuse of $L_{max}$ is implicitly fulfilled by using a FIFO to recycle labels.

To detect completed connected components, an *active tag AT* is introduced for each connected component. If during the raster scan a label is assigned to $L_X$, its entry in $AT$ is updated with $y \bmod 3$, where $y$ is the current row number. Any connected component for which its active tag is not updated in the current image row is completed (as defined in Section 2.3.5). Their feature vectors are read out and their labels are recycled. A connected component is completed in row $y$, when its label does not appear in row $y+1$ of the labelled image $L'$, therefore, the read-out and recycling is carried out in parallel to scanning row $y+2$. The active tag of a label ready to be recycled is, therefore, $y-2 \bmod 3$. For a practical and efficient implementation $AT$ is mapped to a BRAM. This allows up to one label to be recycled per clock cycle and one feature vector to be read out in parallel with processing the following row. In this architecture, the recycling process of a label requires an additional 5 clock cycles of latency until the recycled label is available to be assigned to a new connected component. This is caused by pipeline registers and FIFO delays. The number of labels required for processing a worst case image is therefore $\lceil \frac{W+5}{2} \rceil$.

The feature vector *(FV)* for each connected component is accumulated during the raster scan and stored to the data table $DT$. For a new label operation the data table entry is initialised with the current pixel's feature vector referred to as the *initial feature vector* (IFV). A *label copy* operation combines $L_X$'s $DT$ entry with the $IFV$ and a merger operation combines the $DT$ entries of the two labels and the $IFV$. The operator $\circ$ is defined for combining the feature vectors (see Section 2.3.3). The combining operation, the data structure in $DT$ and the $IFV$ depends on the feature vector, as shown in Table 2.2 in Section 2.3.3 which shows an example for *area*, *bounding box* and *first order image moment* feature vectors.

A new label operation requires one write operation for storing the feature vector, a label copy operation requires one read followed by a write operation, and a merger operation requires two reads followed by a write and an invalidation operation of the data table $DT$ and the active tags $AT$. Additionally, the read-out of feature vectors of completed connected components requires one read operation and one invalidation operation per completed connected component. Therefore, up to five memory operations per pixel are required. The proposed novel scheduling scheme

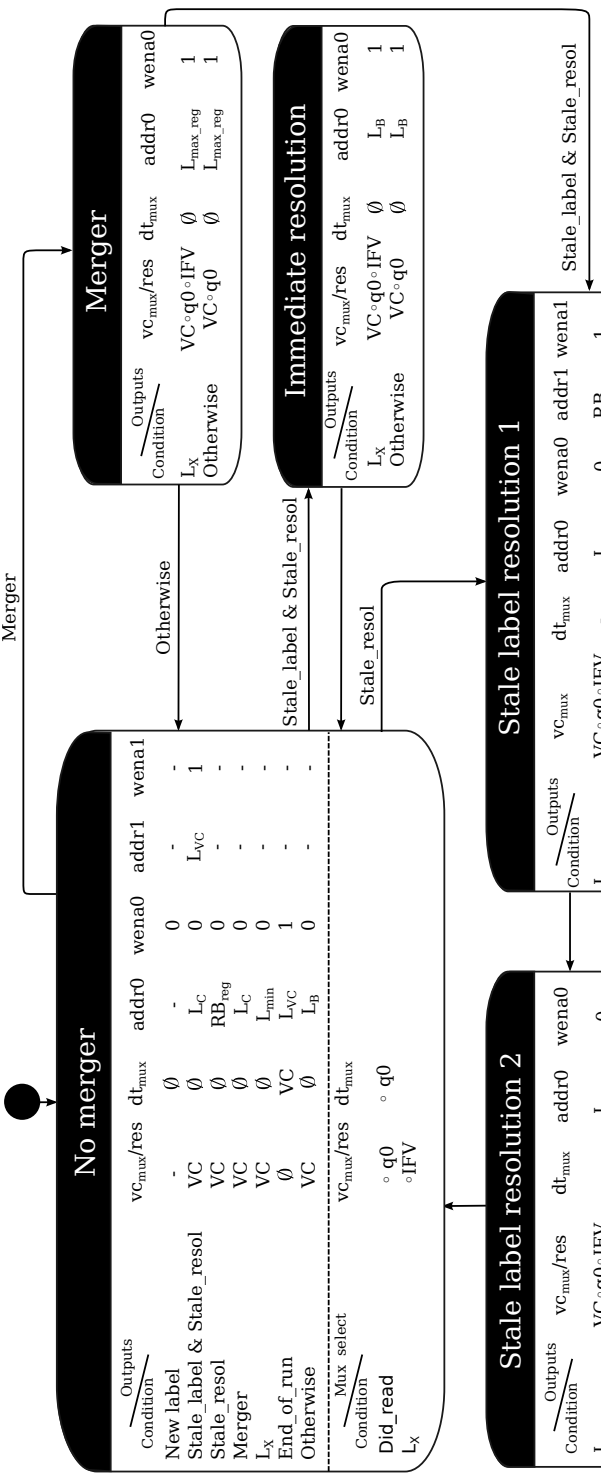



[65] ©2015 IEEE

**Figure 3.6:** Hardware units used for label recycling: the feature vector collection unit and the label management unit. *[65]*

for the memory operations makes a single BRAM port sufficient for updating feature vectors and a second BRAM port for read-out of completed connected components. This reduces the memory resources required for the data table $DT$ by 50% (from two dual-port memories to a single dual-port memory) compared to the architectures from [83] and is a key improvement of the *SLCCA* architecture.

The architecture of the *feature vector collection* unit is shown in Figure 3.6 which shows the hardware units used for label recycling. It contains the finite state machine scheduling the feature vector update process (*FVC-FSM*) of the BRAM storing the data table $DT$, the active tags ATand the valid flags VF. The Mealy state diagram of the *FVC-FSM* is shown in Figure 3.7. The label pattern at the current position $X$ serves as a condition to determine the FSM's outputs and the next FSM state. To save space in Figure 3.7, the new label pattern, label copy pattern and merger pattern are abbreviated as new label, copy and merger, respectively. The condition at the top has the highest priority, and if it does not match, the subsequent condition is evaluated. A *stale label pattern* is detected by the condition *Stale_label* which results from comparing $RB_{reg}$, buffering the label $L[C]$ (introduced in Section 2.3.1 and used in Equation 2.12), and the label at the head of the label stack $LS$. The resolution of a stale label is detected by the condition *Stale_resol*. Details on



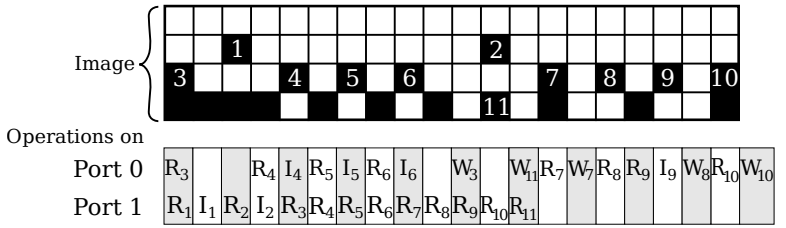

[65] ©2015 IEEE

**Figure 3.7:** Finite state machine for scheduling memory accesses in the *feature vector collection.* *[65]*

stale label processing are introduced in Section 3.1.4. The *FVC-FSM* is connected to the BRAM ports the address ports (*addr*), write-enable port (*wena*), input data (*data*) singals and reads the BRAM's output signals ($q$) for both port 0 and port 1. The *FVC-FSM* controls *addr0*, *wena0, addr1 and wena1* directly, and the ports *data0* and *q0* are connected with the *feature vector cache* $VC$ and the $IFV$ via the multiplexers $vc_{mux}$ and $dt_{mux}$ depending on the label pattern. Port *data1* is always $\emptyset$ since it is only used for invalidations. The feature vector cache $VC$ is realised as a register to store the feature vector associated with the current label $L_X$ to delay a write access to the data table $DT$. The label associated with the accumulated feature vector on $VC$ is $L_{VC}$. The register $L_{max\_reg}$ contains the label of $L_{max}$ of the pixel processed one clock cycle earlier. The BRAM port for feature vector updates can either be used for writing feature vectors or for read requests. The result of a read request appears on the output $q0$ in the next clock cycle. To accumulate the feature vectors with as few memory accesses as possible, the *feature vector cache* $VC$ is updated with the feature vector of label $L_X$ while the current connected component is scanned. A new label operation requires $VC$ to be filled with the initial feature vector $IFV$, a label copy operation on $L_D$ requires the feature vector on the $VC$ to be combined with $IFV$ and a label copy operation applied on $L_A$, $L_B$ or $L_C$ requires $VC$ and the feature vector at the $q0$ to be to combined. The feature vector on $VC$ is written to its data table entry when a background pixel is reached at the end of a run when the condition *end_of_run* is true.

Performing a merger operation combines the feature vectors of $L_{min}$, $L_{max}$ and $IFV$. This is scheduled over three clock cycles carried out when processing the current, the previous and following pixel. In general, for every object pixel, the $IFV$ is combined with the feature vector cache $VC$. Additionally, the following operations are required: In the first cycle, $L_C$ is applied to $addr0$ requesting $L_C$'s feature vector. The feature vector of $L_{AD}$ was either already read in the previous cycle as $L_B$ if the previous pixel was background or is already on the $VC$ if the previous pixel was an object pixel. In state *merger*, the feature vector at $q0$ is combined with the $VC$ and the data table entry associated with the previous $L_{max}$ is invalidated. Depending on the following pixel, the combined feature vector on $VC$ is either written to the data table or further accumulated.

The new value assigned to the feature vector cache $VC$ in each cycle is either $IFV$ (new label pattern), $IFV \circ VC$ (label copy pattern), $VC \circ q0$ or $VC \circ IFV \circ q0$ (merger pattern), and by using the reset signal VC is cleared to $\emptyset$ (completed object). The input of the data table is either an empty feature vector to clear a $DT$ entry $\emptyset$, $VC$ or $VC \circ q0$. If the current pixel is an object pixel, $IFV$ is always combined with $VC$. A read request issued in the previous clock cycle is indicated by the *did_read* condition, which triggers the combination of $VC$ with the output $q0$ in the current clock cycle. For the state *no merger* these two possibilities of combining $VC$ with the $IFV$ and the feature vector at $q0$ are represented by additional conditions below a horizontal dashed line extending the conditions above the dashed line. For example: If the current pixel is an object pixel, a read request was issued in the




[65] ©2015 IEEE

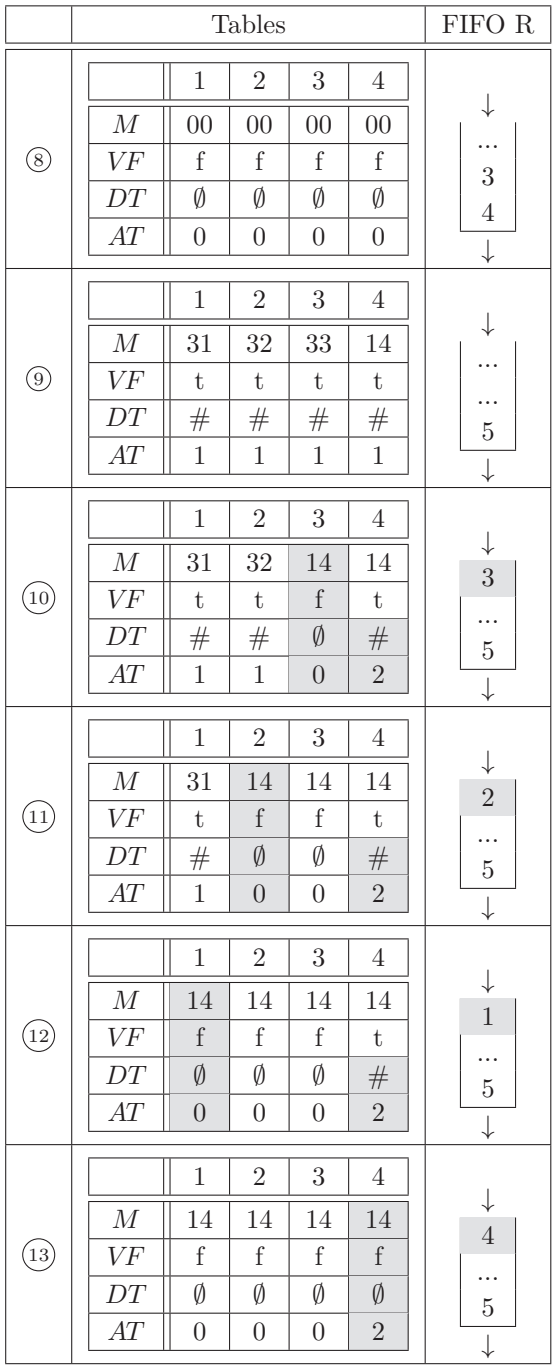
**Figure 3.8:** The image shows the read (R), write (W) and Read-before-invalidate(I) operations for BRAM port 0 and 1 for the last image row. *[65]*

previous clock cycle and the current pixel is a merger pattern, the multiplexer $vc_{mux}$ is set to $VC \circ q0 \circ IFV$.

In Figure 3.8, the scheduling from 3.7 is applied on an example image. In the lower part of Figure 3.8 the read, write and invalidation operations are illustrated, which are induced by the patterns of the image in the upper part of Figure 3.8. BRAM *port 0* is used to read and write feature vectors from $DT$ to update the feature vector associated with the label assigned to the current pixel. In Figure 3.8, the index $i$ of each read instruction $R_i$, write instruction $W_i$ or invalidate instruction $I_i$ indicates the address it is applied on. To read a feature vector in the same clock cycle, in which it is invalidated in $DT$, the invalidate instruction $I_i$ is implemented as *read-before-invalidate*, read operation is carried out on addrress $i$ before the data on address $i$ is invalidated. The *read-before-invalidate* operation is required for an immediate resolution of a stale label (see state *immediate resolution* in Figure 3.7).

The second BRAM port (*port 1*) is used to read out feature vectors of completed connected components and invalidate operations. An invalidate operation on *addr1* of the second port is carried out when *wena1* is one, and during this cycle, the read-out process is paused. There can be up to $\lceil \frac{W}{2} \rceil$ different connected components in a single image row. If all are detected completed in the same image row, $\lceil \frac{W}{2} \rceil$ read and $\lceil \frac{W}{2} \rceil$ invalidation operations have to be carried out. Each feature vector and active tag is associated with exactly one connected component, and therefore $AT$ can be packed together into the same BRAM as $DT$.

Table 3.2 shows how augmented labels and the reuse FIFO $R$ are used to assign the correct labels and to extract the feature vectors for each connected component of the image in Figure 3.9. In this example, the augmented labels are represented by two digit numbers. The first digit represents the row number and the second digit the index. The changes of table and FIFO entries requiring a write operation to a memory are highlighted in grey. Before processing the image (position ⑧), all tables are initially blank and the label reuse FIFO $R$ contains the reused label *4* at the head followed by *3, 2, 1*.




[65] ©2015 IEEE

**Table 3.2:** Correct labelling of image in Figure 3.9 by using augmented labels. A # in the data table DT indicates that the corresponding entry contains meaningful feature vector data, while ∅ indicates that the entry is empty. *[65]*

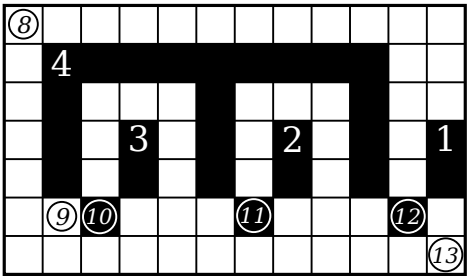





**Figure 3.9:** Assigning the labels out of order creates a corrupted merger table. *[65]*

For a new label operation, the connected components are labelled with augmented labels, i.e. label 4 becomes 14, label 3 becomes 33, etc. At position (9) $DT$ contains a feature vector for each of the component segments labelled 31, 32, 33 and 14.

The object pixel at (10) has two object labels 14 and 33 in its neighbourhood, i.e. a merger pattern. At this position $L_{min} = 14$ and $L_{max} = 33$, making $M[3]$ to point to augmented label 14 and recycling label 3 to $R$. The feature vector at $DT[3]$ is merged with the feature vector of $DT[4]$ and stored to $DT[4]$, while $VF[3]$ is set to *false* and $DT[3]$ is invalidated.

The patterns at positions (11) and (12) lead to merger operations which update the entries of labels 32, 31 and 14 in tables $M$, $VF$, $DT$ and $AT$ and return non-root labels to FIFO $R$.

At the end of the following image row, at (13), the connected component labelled 14 is detected as completed so its label is returned to $R$ as well.'[65]

### 3.1.4 Stale labels

'A label is referred to as *stale* if a single lookup in $M$ does not yield the root label. This has not been taken into account in previous hardware architectures [7, 58] and requires further processing. A *bridge pattern* is a component segment in which an object label appears twice in the current image row separated by background pixels. The bridge pattern's object pixels in the current row are referred to as its piers. The pixels belonging to the bridge, which are above the current row, are referred to as the bridge's arc. In the following figures the arc is either a group of pixels (see 3.10) or is shown as a dashed line (see 3.12) indicating a path of object pixels. Merging a bridge pattern's pier label with a smaller label requires a lookup in $M$ for the other pier label to be labelled correctly in the neighbourhood context. Thus the height of the connected component*'s* tree structure becomes 1. At the beginning of each image row the height of all tree structures is smaller or equal one due to chain resolution. A merger pattern with a bridge pattern's left pier in the current

image row which merged another component segment in the previous row results in a tree height of 2 in the merger table $M$ (Figure 3.10). A single lookup of a label which has a distance of 2 to its root label in the rooted tree structure in $M$ does not yield the root label. If such a non-root label appears as the minimum label in the neighbourhood of an object pixel, the wrong label is assigned to $L_X$.

After the lookup to determeine the parent of a label, a root label on $M$ can be detected by using an additional lookup to check if the label points to itself. A *valid flag VF* for each label allows stale labels to be detected without this additional lookup: A new label operation sets the new label's $VF$ flag to *true* and a merger operation sets the $VF$ flag of $L_{max}$ to *false*. Reading out $VF$ along with $M$ indicates whether the assigned label is a root label or not. If a stale label is at the output of the row buffer, the label assigned to $L_X$ is not a root. Therefore, $L_X$'s feature vector is stored to the data table $DT$ until its root label appears in the neighbourhood context and their feature vectors are combined.

If there are nested bridge patterns, several stale labels can be detected before their root labels appear in the neighbourhood context. To keep a record of the stale labels which have to be merged with their root labels a *label stack LS* is introduced. The label $L_X$ is pushed to $LS$ whenever its $VF$ flag is *false.* Its feature vector is temporarily stored on the data table entry of $L_X$, which is unused for any non-root label. When $LS_{head}$, the label at the head of the label stack, is equal to the output of the row buffer, it is popped off $LS$ to combine the feature vector of $LS_{head}$ and its root label. In the FSM in Figure 3.7, the states *Stale label resolution 1* and *Stale label resolution 2* handle merger patterns of feature vectors of non-root labels. If $L_X$ is detected to be stale and its resolution is detected simultaneously, the feature vectors are handled as shown in state *immediate resolution* of Figure 3.7. The $VF$ flag and the data table $DT$ each have one association per label, i.e. they can be mapped to the same logical BRAM resource. The maximum number of stale labels which can appear in an image row is up to $\lceil \frac{W}{10} \rceil$. To minimise the required hardware resources the stack $LS$ can either be realised as BRAM or distributed RAM, dependent on the image size.

The two patterns in Figure 3.10 generate tree structures of height 2 in $M$ resulting in stale labels. In both images, label 2 and 3 are merged in the image row previous to the position of $L_X$. After merging label 1 and 2 in the current image row (buttom row in Figure 3.10), all the label of all pixels labelled 3 leaving the row buffer are translated to label 2. This is done by the lookup in merger table $M$ which is done for every label. In this case this lookup assigns a non-root label to $L_X$.

Tables 3.3 and 3.4 (on page 89 and page 91) reflect the steps for processing the image in Figure 3.11 making use of the label stack $LS$. At first, the connected components are considered which consist of the component segments initially labelled 01, 02 and 23. The merger pattern at (17) induces a merger operation updating $M[3]$ to label 02 and sets the valid flag of label 23 to false, i.e. $VF[3] = false$ indicates that label 23 is not a root label. The feature vectors of the component segments 02 and 23

| | Tables | | | | | | | $LS$ |
|---|---|---|---|---|---|---|---|---|
| (14) | | 1 | 2 | 3 | 4 | 5 | 6 | ↓↑ |
| | $M$ | 00 | 00 | 00 | 00 | 00 | 00 | |
| | $VF$ | f | f | f | f | f | f | |
| | $DT$ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | |
| | $AT$ | 0 | 0 | 0 | 0 | 0 | 0 | ∅ |
| (15) | $M$ | 01 | 02 | 23 | 44 | 45 | 66 | ↓↑ |
| | $VF$ | t | t | t | t | t | t | |
| | $DT$ | # | # | # | # | # | # | |
| | $AT$ | 0 | 0 | 0 | 0 | 0 | 0 | ∅ |
| (16) | $M$ | 01 | 02 | 23 | 44 | 45 | 45 | ↓↑ |
| | $VF$ | t | t | t | t | t | f | |
| | $DT$ | # | # | # | # | # | ∅ | |
| | $AT$ | 0 | 0 | 0 | 0 | 0 | 0 | ∅ |
| (17) | $M$ | 01 | 02 | 02 | 44 | 45 | 45 | ↓↑ |
| | $VF$ | t | t | f | t | t | f | |
| | $DT$ | # | # | ∅ | # | # | ∅ | |
| | $AT$ | 0 | 0 | 0 | 0 | 0 | 0 | ∅ |
| (18) | $M$ | 01 | 01 | 02 | 44 | 45 | 45 | ↓↑ |
| | $VF$ | t | f | f | t | t | f | |
| | $DT$ | # | ∅ | ∅ | # | # | ∅ | |
| | $AT$ | 1 | 1 | 0 | 0 | 0 | 0 | ∅ |
| (19) | $M$ | 01 | 01 | 02 | 44 | 45 | 45 | ↓↑ |
| | $VF$ | t | f | f | t | t | f | |
| | $DT$ | # | # | ∅ | # | # | ∅ | |
| | $AT$ | 1 | 1 | 0 | 0 | 0 | 0 | 2 |
| (20) | $M$ | 01 | 01 | 02 | 44 | 44 | 45 | ↓↑ |
| | $VF$ | t | f | f | t | f | f | |
| | $DT$ | # | # | ∅ | # | ∅ | ∅ | |
| | $AT$ | 1 | 0 | 0 | 1 | 1 | 0 | 2 |

[65] ©2015 IEEE

**Table 3.3:** Feature vector extraction for the connected components of Figure 3.11 which contains several stale labels. Augmented labels in $M$ are represented by a two digit number - the first digit is the row, the second the index. The valid flags in $VF$ are either *(t)rue* or *(f)alse.* [65]

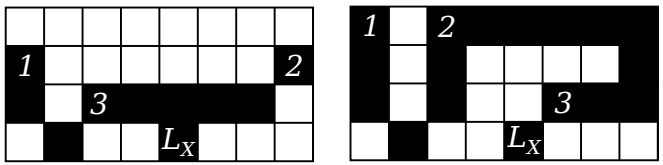





**Figure 3.10:** Two basic examples of stale labels: At position $X$, the content of the merger table $M$ is: $3 \rightarrow 2 \rightarrow 1$. A lookup in the entry of $M$ associated with label 3 returns label 2, which has been merged with label 1 earlier in the current row. *[65]*

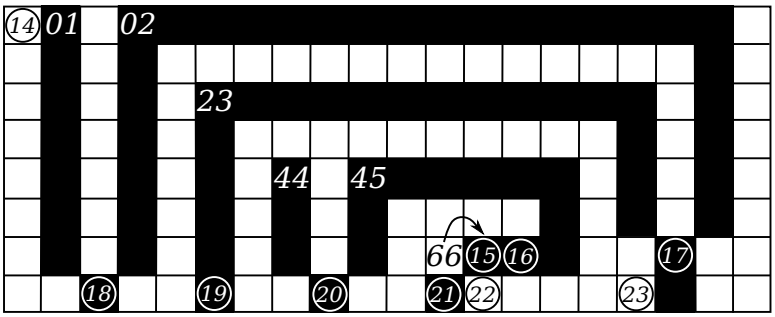





**Figure 3.11:** Image containing nested connected components with stale labels. *[65]*

are combined and stored to $DT[2]$, and the data table entry of label 3 is cleared, $DT[3] \coloneqq \emptyset$. Therefore, the tree height of the component segment labelled 02 is one before reaching position (18). The merger operation induced by the merger pattern at position (18) updates $M[2]$ to label 01 and sets $VF[2] \coloneqq false$. The tree structure of the connected component labelled 01 is therefore of height two which makes label 23 stale. At position (19) $L_B$ is 02 as a result of a single lookup of label 23. Since $VF[2]$, is set to false at (18), a non-root label is detected and assigned to $L_X$ (position (19)). From this it follows that the feature vector of the current pixel is stored to $DT[2]$ and label 2 is added to the label stack $LS$. At position (23), the label stored on the register attached to the output of the row buffer $RB_{reg}$ (label at position (23)) is equal to the label at the head of the label stack $LS$. In this case the feature vector of the non-root label 02 temporarily stored at $DT[2]$ is combined with the feature vector of its root label 01 and stored to $DT[1]$. Simultaneously, $DT[2]$ is cleared and label 2 is popped off $LS$.

The steps of processing the inner connected component consisting of the component segments 44, 45 and 66 demonstrate that $LS$ requires a stack data structure for

| | Tables | | | | | | | LS |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | |
| (21) | M | 01 | 01 | 02 | 44 | 44 | 45 | ↓↑ |
| | VF | t | f | f | t | f | f | |
| | DT | # | # | ∅ | # | # | ∅ | 5 |
| | AT | 1 | 0 | 0 | 1 | 1 | 0 | 2 |
| (22) | M | 01 | 01 | 02 | 44 | 44 | 45 | ↓↑ |
| | VF | t | f | f | t | f | f | |
| | DT | # | # | ∅ | # | ∅ | ∅ | ~~5~~ |
| | AT | 1 | 0 | 0 | 1 | 1 | 0 | 2 |
| (23) | M | 01 | 01 | 02 | 44 | 44 | 45 | ↓↑ |
| | VF | t | f | f | t | f | f | |
| | DT | # | ∅ | ∅ | # | ∅ | ∅ | |
| | AT | 1 | 1 | 0 | 1 | 0 | 0 | ~~2~~ |

[65] ©2015 IEEE

**Table 3.4:** Feature vector extraction for the connected components of Figure 3.11 which contains several stale labels. Augmented labels in $M$ are represented by a two digit number - the first digit is the row, the second the index. The valid flags in $VF$ are either *(t)rue* or *(f)alse*. *[65]*

processing nested stale label patterns. The merger operation induced by the merger pattern at (16) updates $M[6]$ to label 45 and sets $VF[6] := false$. The feature vectors of the component segments 45 and 66 are combined and stored to $DT[5]$, and the data table entry of label 66 is cleared. As before, label 66 becomes stale because of the merger operation at (20), which increases the height of the tree structure of the connected component labelled 44 from one to two and sets $VF[5] := false$. The feature vectors of the component segments 44 and 45 are combined and stored at $DT[4]$, the data table entry of label 45 is cleared, $DT[5] := \emptyset$. At (21) the non-root label 45 (which is looked up from label 66) is assigned to $L_{min}$. Therefore, the feature vector of the pixel at (21) is stored at DT[5] and label 5 is added to the label stack $LS$. At position (22) the label stored on the register attached to the output of the row buffer $RB_{reg}$ is equal to the label at the head of the label stack $LS$, the feature vector of the non-root label 45 temporarily stored at $DT[5]$ is combined with the feature vector of its root label 44 and stored $DT[4]$. Simultaneously, $DT[5]$ is cleared and label 5 is popped of $LS$.'[65]

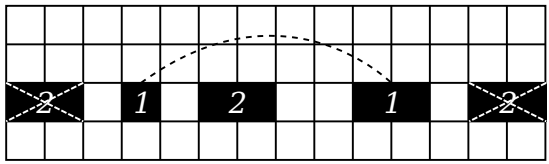





**Figure 3.12:** Impossible scenario: Label between bridge piers appears to the right or left of the bridge. *[65]*

### 3.1.5 Validation of the architecture

'The correct functionality of the architecture relies on the assumption that a single lookup in $M$ is always sufficient to assign a label to $L_X$ which associates the pixel at position $X$ with its connected component. To achieve this, the tree structures in the merger table $M$ are compressed to a height of one by different means: Merger operations can either affect the tree structure in $M$ of labels to the left or to the right of position $X$. Label chains create tree structures with height $> 1$ for labels left of the current position $X$ which are relevant for the processing of the next image row. Stale labels create a tree height of 2 for labels to the right of $X$ which are relevant for processing the current image row. With the stacks $S$ and $LS$ both cases, chains and stale labels, are handled by memorising temporary differences in $M$ and re-establishing the single lookup principle by updating the merger table $M$ after the occurrence of a chain or a stale label.

To ensure that each pixel is associated with its connected component, the different combinations of stale label and bridges are listed in Table 3.5 which are analysed in the following. Chains and stale labels are formed by merger patterns or combinations of merger patterns and bridge patterns. In an image neighbourhood, not all combinations of stale labels and bridge patterns are possible. This is discussed in the following. The cases that are not possible are marked with an ✗ in Table 3.5. There is no need to further examine whether labelling is correct for these impossible combinations. The combinations from Table 3.5 which are not impossible are marked by a ✓ and were examined to assign the correct label to the current label $L_X$ with examples performed with a pencil-and-paper method. A formal proof is beyond the scope of this dissertation and was, therefore, carried out in [67]. In the image in Figure 3.12, the current image row contains two piers of a bridge pattern labelled 1 and a connected component between the two bridge piers labelled 2. In the already scanned image, part no pixels can be labelled 2 outside the bridge. Therefore, a pixel's label between two bridge piers of the same bridge is always larger than the bridge's label. Merger patterns *10, 14, 21* and *23* of Table 3.5 are therefore not possible.

| Combinations of Merger Patterns | | | | | | | |
|---|---|---|---|---|---|---|---|
| Merger Pattern # | Stale | | Bridge | | $L_{AorD} > L_C$ | Pattern Possible | Correct labelling |
| | AD | C | AD | C | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | ✓ | ✓ |
| 1 | 0 | 0 | 0 | 0 | 1 | ✓ | ✓ |
| 2 | 0 | 0 | 0 | 1 | 0 | ✓ | ✓ |
| 3 | 0 | 0 | 0 | 1 | 1 | ✓ | ✓ |
| 4 | 0 | 0 | 1 | 0 | 0 | ✓ | ✓ |
| 5 | 0 | 0 | 1 | 0 | 1 | ✓ | ✓ |
| 6 | 0 | 0 | 1 | 1 | 0 | ✓ | ✓ |
| 7 | 0 | 0 | 1 | 1 | 1 | ✓ | ✓ |
| 8 | 0 | 1 | 0 | 0 | 0 | ✗ | |
| 9 | 0 | 1 | 0 | 0 | 1 | ✗ | |
| 10 | 0 | 1 | 0 | 1 | 0 | ✗ | |
| 11 | 0 | 1 | 0 | 1 | 1 | ✓ | ✓ |
| 12 | 0 | 1 | 1 | 0 | 0 | ✗ | |
| 13 | 0 | 1 | 1 | 0 | 1 | ✗ | |
| 14 | 0 | 1 | 1 | 1 | 0 | ✗ | |
| 15 | 0 | 1 | 1 | 1 | 1 | ✓ | ✓ |
| 16-19 | 1 | 0 | 0 | - | - | ✗ | |
| 20 | 1 | 0 | 1 | 0 | 0 | ✓ | ✓ |
| 21 | 1 | 0 | 1 | 0 | 1 | ✗ | |
| 22 | 1 | 0 | 1 | 1 | 0 | ✓ | ✓ |
| 23 | 1 | 0 | 1 | 1 | 1 | ✗ | |
| 24-31 | 1 | 1 | - | - | - | ✗ | |



**Table 3.5:** Validation of possible combinations of merger patterns. *Don't cares* in the table are marked by '-'. *[65]*

According to the definition in Section 3.1.4, a stale label is always created by a bridge. This excludes merger patterns *8, 9, 12, 13* and *16-19* from the possible patterns, as marked in Table 3.5. For label $L_C$ to be stale, one bridge is necessary, while for the label $L_{AorD}$ to be stale, two bridges are required. The four different combinations to form a pattern making both $L_{AorD}$ and $L_C$ stale are shown in Figure 3.13, with bridges indicated by dashed lines. All four attempts to make both $L_{AorD}$ and $L_C$ stale fail because of intersecting bridges making them a connected component; therefore, the merger patterns *24-31* of Table 3.5 can never exist. Table 3.5 shows that all cases of merger patterns for stale labels and bridges which are possible are labelled correctly.

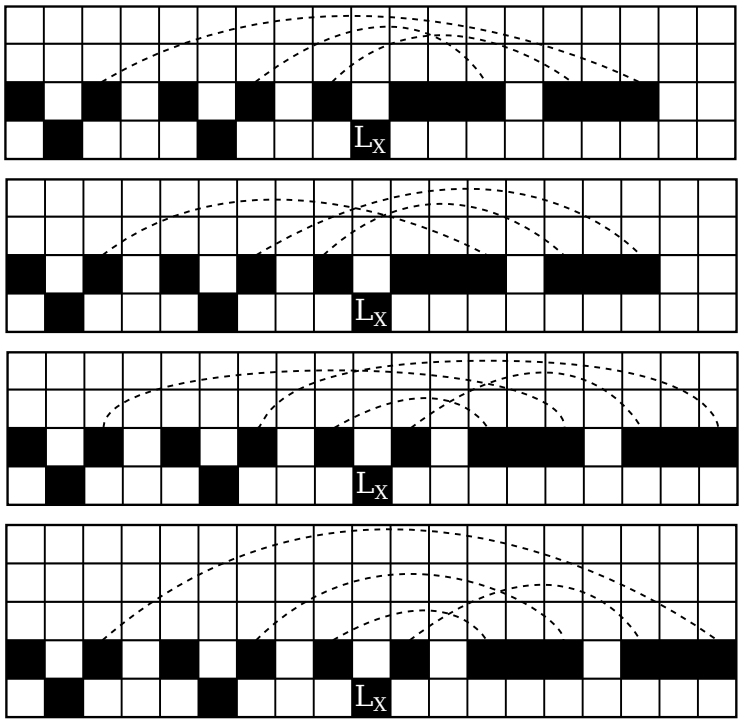
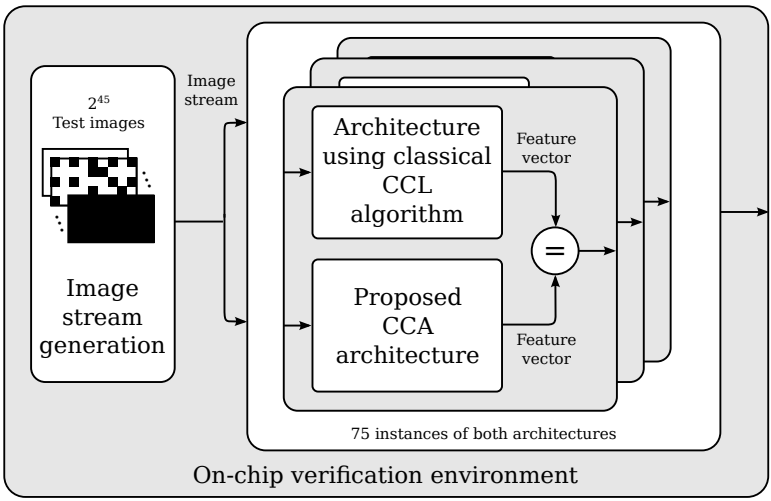





**Figure 3.13:** All failing attempts to make the two object labels of a merger pattern stale. *[65]*

### 3.1.6 Validation of the implementation

To ensure the functional correctness of the implementation of the *SLCCA* architecture all possible pixel combinations for a small size test image were streamed into the architecture and the outputs verified against a reference implementation [9], as shown in Figure 3.14. The size of the test image chosen for this full verification is a trade-off between the complexity of the pixel patterns in the image and processing time which grows exponentially with the number of pixels. For the chosen image size of $9 \times 5$ pixels, all of the possible $2^{45}$ different image patterns were successfully verified against the reference implementation applying the classical two-pass connected components labelling algorithm. To reduce the duration of the verification process, an *on-chip verification environment* realised in hardware on an FPGA was used. It contains 75 instances of the *SLCCA* architecture and the reference implementation working in parallel, which enables an accelerated verification process, from more than one year of verification time on a single instance down to seven days for all instances processing in parallel. The exhaustive verification of all binary $9 \times 5$ images covers the cases of Table 3.5, though parts of the implementation are still




[65] ©2015 IEEE

**Figure 3.14:** Block diagram of the on-chip verification environment which successfully verified all combinations of a $9 \times 5$ image against a reference implementation. *[65]*

not exercised by images of this size. This required further validation to make sure the implementation realises all scenarios of the previously described architecture correctly, e.g. for nested stale labels such as in Figure 3.11. These parts of the implementation were validated by checking the code coverage of the VHDL code in behavioural simulation to ensure correct functionality [120].'[65]

| Data structure | Rosenfeld et al. [102] | Ma and Bailey [83] | SLCCA |
|---|---|---|---|
| $N_L$ | $\lceil \frac{W \times H}{4} \rceil$ | $\lceil \frac{W}{2} \rceil$ | $\lceil \frac{W+5}{2} \rceil$ |
| $N_M$ | – | $\lfloor \frac{W-1}{2} \rfloor$ | $\lfloor \frac{W-1}{2} \rfloor$ |
| $W_{IDX}$ | $\lceil \log_2(N_L) \rceil$ | $\lceil \log_2(N_L) \rceil$ | $\lceil \log_2(N_L) \rceil$ |
| $W_{AL}$ | – | – | $W_{IDX} + \lceil \log_2(H) \rceil$ |
| $L$ | $N_L \times W_{IDX} \times W \times H$ | – | – |
| $RB$ | – | $N_L \times W_{IDX}$ | $N_L \times W_{IDX}$ |
| $S$ | – | $2 \times W_{IDX} \times N_M$ | $2 \times W_{IDX} \times N_M$ |
| $M$ | $N_L \times W_{IDX}$ | $2 \times N_L \times W_{IDX}$ | $N_L \times W_{AL}$ |
| $R$ | – | – | $N_L \times W_{IDX}$ |
| $DT$ | – | $2 \times N_L \times W_{FV}$ | $N_L \times W_{FV}$ |
| $TT$ | – | $N_L \times W_{IDX}$ | – |
| $LS$ | – | – | $N_L \times \lceil \frac{W}{10} \rceil$ |
| $VF$ | – | – | $N_L$ |
| $AT$ | – | – | $2 \times N_L$ |



**Table 3.6:** Comparison of the memory bits required for components analysis for the classical CCL algorithm [102], the single-pass architecture from [83] and the *SLCCA* architecture for different image sizes. *[65]*

## 3.2 Experimental Results and Discussion

'In this section, the results for the realisation of the *SLCCA* hardware architecture is evaluated and benchmarked. The hardware architecture's performance and memory-efficiency is compared to other connected components analysis hardware architectures for different image sizes from *VGA* with $640 \times 480$ pixels per image to *ultra-high-definition (UHD)* with image sizes up to $7680 \times 4320$ pixels [55].'[65]

### 3.2.1 Memory Requirements

'Table 3.6 compares the number of bits required for the memories integrated in the different processing blocks of the CCA architectures for an image of the size $W \times H$ pixels for both the architectures in [83] and the *SLCCA* architecture, as well as the *Rosenfeld's* classical CCL algorithm [102]. Both architectures require the row buffer $RB$ and the stack $S$ of the same size for connected components analysis.

The CCA architecture by *Ma and Bailey* [83] has been the most memory-efficient architecture in the academic literature. For [83] the following on-chip memories are required: Every second pixel can be a different connected component or be a component segment merging another segment later in the image. Therefore, the

number of labels $N_L$ is only dependent on the image width. The architecture requires two merger tables $M$, one to store the label pairs for each merger pattern of the previous row and one to store each label pair of the merger pattern of the current row. To store a relation between two labels, each entry of the merger table $M$ is as wide as a label ($W_{IDX}$). The aggressive relabelling scheme requires a translation table $TT$ with $N_L$ entries of width $W_{IDX}$. For the feature vector collection, two data tables $DT$, one for the feature vectors of the previous row's labels and one for the feature vectors of the current row's labels, are required. The width $W_{FV}$ of each entry of the data table $DT$ is dependent on the feature to be extracted.

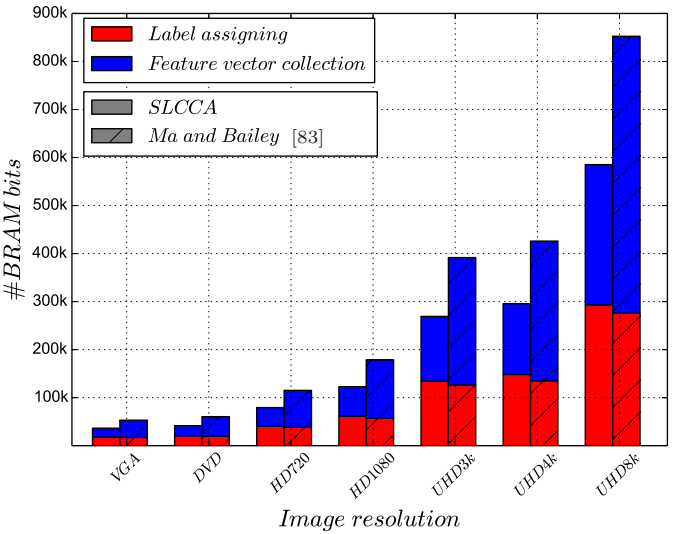
The *SLCCA* architecture requires the following on-chip memory: The number of labels $N_L$ depends on the image width plus a constant to compensate for the 5 clock cycle latency of the label recycling process, and is therefore $\lceil \frac{W+5}{2} \rceil$. The augmented labelling ($AL$) requires the merger table of the *SLCCA* architecture to be as wide as the width of an augmented label $W_{AL}$. The label reuse FIFO $R$ of the label management unit needs to be able to store all labels, i.e. requires a depth of $N_L$ labels. For the feature vector collection a single data table $DT$ with $N_L$ entries of width $W_{FV}$ is sufficient.

Table 3.7 compares the amount of on-chip memory required between the classical CCL algorithm [102], the single-pass architecture by Ma [83] and the *SLCCA* architecture for extraction of the bounding box and the area features of images of different sizes. Table 3.7 shows the amount of on-chip memory for common image size from *VGA* to *UHD8k* based on the equations from Table 3.6. The ratio of on-chip memory required for the *label assigning* process and the *feature vector collection* process is also compared. The data strucutres required for the *label assigning* process are the row buffer $RB$, the stack $S$, the merger table $M$, the valid flags $VF$, the FIFO for reused labels $R$ and the translation table $TT$. The data strucutres required for the *feature vector collection* process are the data table $DT$, the label stack $LS$ and the active tags $AT$. The values in Table 3.7 for the *SLCCA* architecture and the values of the architecture by *Ma and Bailey* [83] are depicted in the diagram in Figure 3.15. For all image sizes from *VGA* to *UHD8k*, the *SLCCA* architectures requires fewer on-chip memory resources than the architectures of *Ma and Bailey* [83]. By halving the memory resources for feature vector collection, the resources required for the entire *SLCCA* architecture can be reduced by up to 31% compared to *Ma and Bailey* [83] for extracting the bounding box and the area feature vector for each connected component. The width of the bounding box feature vector, the area feature vector and the first order moment feature vector is dependent on the image dimension. For the simultaneous extraction of multiple features, the width of the data table increases accordingly with the image dimension. The feature vector for the three features, bounding box, area and first order image moment adds up to 175 bits per connected component. When realising a CCA architecture extracting those three features simultaneously, up to 42% of memory resources are saved compared to [83]. For a wider feature vector, even more memory resources are saved.

| | Image size | | | | | | |
|---|---|---|---|---|---|---|---|
| | VGA | DVD | HD720 | HD1080 | UHD3k | UHD4k | UHD8k |
| Data structure | 640 × 480 | 720 × 576 | 1280 × 720 | 1920 × 1080 | 3840 × 2160 | 4096 × 2160 | 7680 × 4320 |
| Rosenfeld's classical two-pass algorithm [102] | | | | | | | |
| $L$ | 5M | 7M | 16M | 39M | 174M | 194M | 763M |
| $M$ | 1.3M | 1.7M | 4M | 9M | 43M | 48M | 190M |
| $DT$ | 4.3M | 5.9M | 13.8M | 32.6M | 143.0M | 157.0M | 622.0M |
| $\sum$ | 10.8M | 14.7M | 34.5M | 81.9M | 0.36G | 0.40G | 1.57G |
| Ma and Bailey's optimised single-pass architecture [83] | | | | | | | |
| $M$ | 5760 | 6480 | 12800 | 19200 | 42240 | 45056 | 92160 |
| $TT$ | 2880 | 3240 | 6400 | 9600 | 21120 | 22528 | 46080 |
| $RB$ | 5760 | 6480 | 12800 | 19200 | 42240 | 49152 | 92160 |
| $S$ | 2880 | 3240 | 6400 | 9600 | 21120 | 24576 | 46080 |
| $DT$ | 35840 | 41040 | 76800 | 120960 | 264960 | 290816 | 576000 |
| $\sum$ | 53120 | 60480 | 115200 | 178560 | 391680 | 425984 | 852480 |
| SLCCA | | | | | | | |
| $M$ | 5760 | 6840 | 12800 | 20160 | 44160 | 49152 | 96000 |
| $R$ | 2907 | 3267 | 6430 | 9630 | 21153 | 24612 | 46116 |
| $LS$ | 576 | 648 | 1280 | 1920 | 4224 | 4920 | 9216 |
| $RB$ | 5760 | 6480 | 12800 | 19200 | 42240 | 49152 | 92160 |
| $S$ | 2880 | 3240 | 6400 | 9600 | 21120 | 24576 | 46080 |
| $DT$ | 18243 | 20883 | 39043 | 61443 | 134403 | 147459 | 291843 |
| $E$ | 323 | 363 | 643 | 963 | 1923 | 2051 | 3843 |
| $V$ | 323 | 363 | 643 | 963 | 1923 | 2051 | 3843 |
| $\sum$ | 36449 | 41721 | 79396 | 122916 | 269223 | 295778 | 585258 |



**Table 3.7:** Comparison of on-chip BRAM bits required for components analysis for the classical CCL algorithm, the single-pass architecture from [83] and the *SLCCA* architecture for different image sizes. The size of data table $DT$ corresponds to extracting bounding box and area features simultaneously. *[65]*




[65] ©2015 IEEE

**Figure 3.15:** The bar diagram shows the number of on-chip BRAM bits for the label assigning in red and the feature vector collection in blue. The hatched bars on the right show the memory required for the architecture in [83] and the left bars show the memory required for the *SLCCA* architecture for different image sizes. *[65]*

As discussed in the introduction, the access time to the memory structures, especially the latency, was identified to be the most important criterion for processing a pixel stream with a high throughput and low latency. Therefore, offloading the memory structures to off-chip memory to save chip area counteracts the key idea of the architecture. If implementing the *SLCCA* architecture on an ASIC, there are a number of possibilities to realise the memory structures, either based on SRAM or DRAM cells [31]. While DRAM cells require fewer transistors per bit than SRAM cells, in general, any data stored in DRAM must be refreshed periodically. In CCA the data structures are accumulated from one row to the next and are therefore only up-to-date for a maximum of two image rows before they are either changed or read out. For typical image sizes this process is less than a millisecond which is significantly lower than the refresh rate of a typical DRAM cell. This allows the smaller DRAM cells to be used without refresh. This applies to all internal memory structures (stored in BRAMs in the presented implementation), except the reuse FIFO $R$, which needs to store the unused labels for the duration of up to one frame. ’[65]

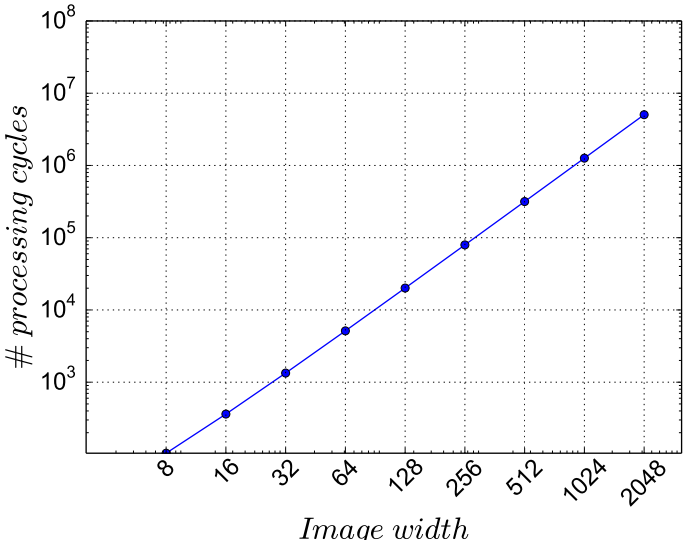




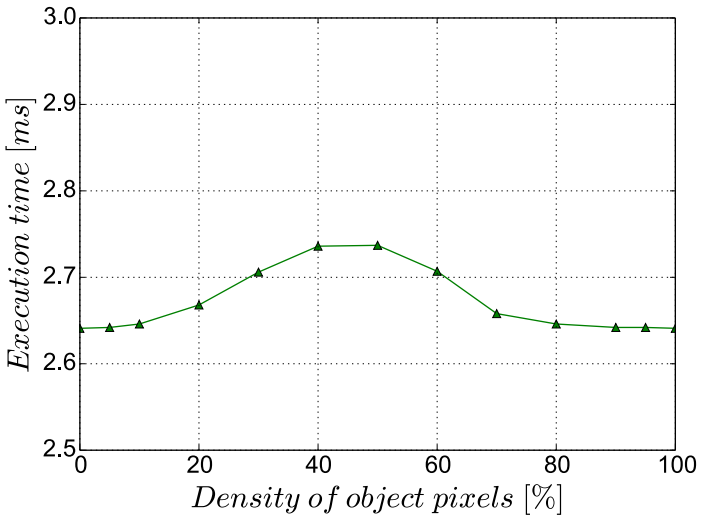

**Figure 3.16:** This diagram shows the number of clock cycles for processing square images of different sizes with the worst case pattern from Figure 3.18. *[65]*

### 3.2.2 Benchmark

'The performance of the *SLCCA* architecture is measured in a manner similar to the benchmark in [51]. In [51] test images from *Standard Image Database (SIPI)* [124] and random images with different densities of object pixels are used. All images for this performance evaluation are of size $512 \times 512$ pixels. Greyscale images are binarised using the threshold value determined by Otsu's method [97]. All results in this section are acquired by behavioural simulation of the implementation of the *SLCCA* architecture implemented in VHDL.

Every image pixel can be processed in one processing cycle; additional processing cycles at the end of the image row result from chain processing. From this, it follows that the worst case processing time occurs when the number of entries on stack $S$ is maximum. To analyse the worst case processing time for different image sizes from 64 pixels to 4 megapixels, images with the worst case pattern of Figure 3.18(e) are evaluated. Figure 3.16 shows that the number of processing cycles scale linearly for the examined image sizes.

Figure 3.17 shows the execution time for processing a random image as a function of the density of object pixels in the image for $512 \times 512$ images. Random images were used to evaluate the execution time against the number of object pixels in an

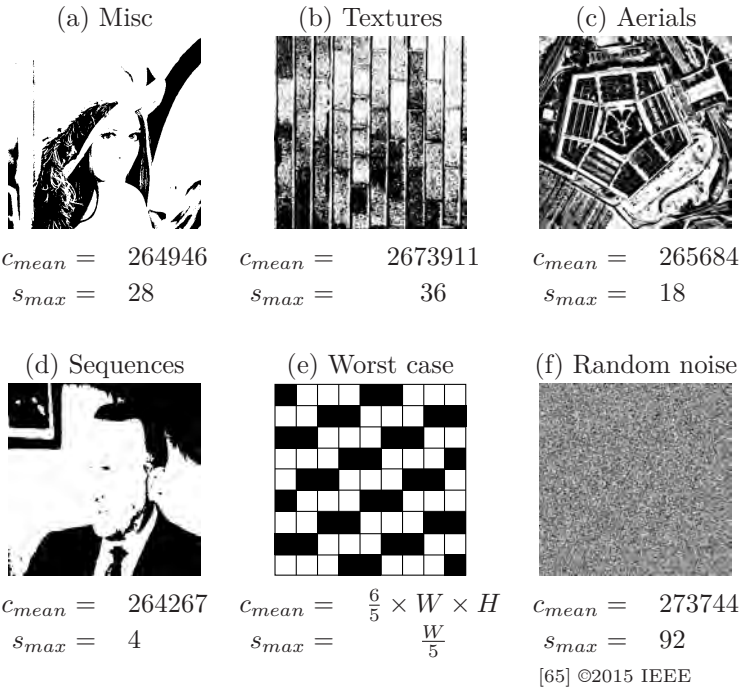



[65] ©2015 IEEE

**Figure 3.17:** This diagram shows the execution time of the implementation of the SLCCA architecture operated at 100 MHz for $512 \times 512$ images filled with random noise for different densities of object pixels. *[65]*

image. For 0% and 100% density of object pixels, the image contains either none or one connected component, with the highest number of connected components being around 50%. The diagram in Figure 3.17 shows that the execution time is maximum between 40% and 50% object pixel density, which confirms the results from Section 2.5. This corresponds to an overhead due to stack processing at the end of the row of less than 5%, which is significantly lower than in the worst case. To evaluate the architecture's performance for processing natural images, a representative image series from the *SIPI database* containing 215 typical images are used divided into the categories *misc*, *textures*, *aerials* and *sequences*. In Figure 3.18 the results for the mean processing cycles per image for each image series and the maximum number of stack entries for processing each image series is shown.'[65]

The diagram in Figure 3.19 shows the maximum processing latency of the presented *SLCCA* architecture for different image sizes. The maximum *processing latency* describes the maximum amount of clock cycles from receiving the last pixel associated with a connected component $c$ in forward raster scan order until the feature vector of connected component $c$ is output by the *SLCCA* architecture. The maximum processing latency of the *SLCCA* architecture is equal to the number of clock cycles needed to process the two image rows subsequent to the last pixel associated with connected component $c$. Two image rows are required since a connected component




[65] ©2015 IEEE

**Figure 3.18:** This figure gives the mean number of processing cycles $c_{mean}$ and the maximum stack size $s_{max}$ for (a) - (d) which are test images from USC-SIPI Image Database [124]. (e) Worst case image [7] with the maximum number of merger patterns. (f) Random noise image with 50% of object pixels as in Figure 3.17. *[65]*

is detected to be completed (Equation 2.28) if its feature vector in $DT$ was last updated two rows before the currently processed row. The processing of two image rows requires independent of the contained pixel data $2 \times W$ clock cycles. Another cycle is required to process an entry in stack $S$ for each non-propagating merger detected in these two image rows.

The $chain_{max}$ *pattern* is the pattern which creates the maximum number of entries in stack $S$. It has the following properties: The $chain_{max}$ *pattern* spans the full width $W$ of the input image and consists of the maximum of $\lfloor \frac{W}{2} \rfloor$ non-propagating merger patterns in one image row. If the image height exceeds the image width ($W > H$), $chain_{max}$ *pattern* contains multiple chain patterns in the same image row. The worst case pattern from Figure 3.18(e) contains a maximum of $2 \times \lceil \frac{W}{5} \rceil$ non-propagating merger patterns in two subsequent image rows. Therefore, the latency for processing the pattern from Figure 3.18(e) is lower than processing a $chain_{max}$ pattern. The latency values in the diagram in Figure 3.19 are, therefore, the processing latency

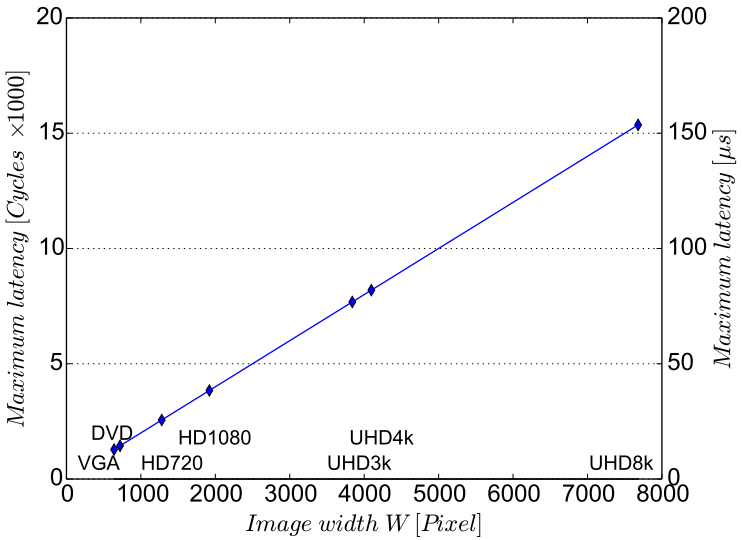
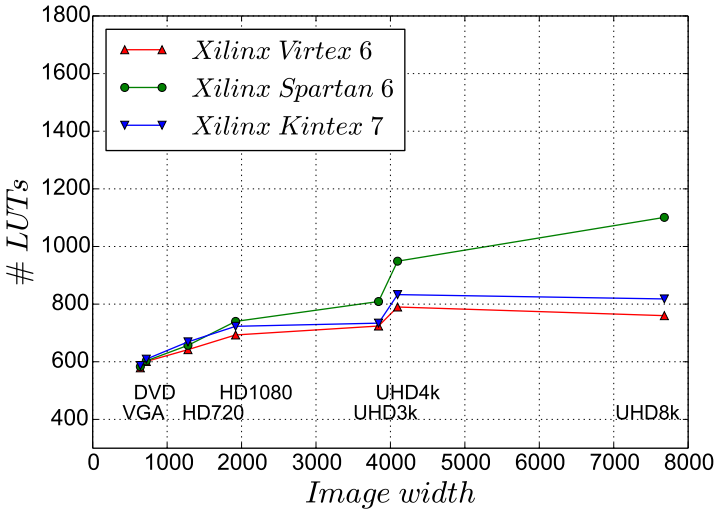



**Figure 3.19:** This diagram shows the guaranteed upper bound for the maximum processing latency in clock cycles and in $\mu s$ for the implementation of the SLCCA architecture when operated at 100 MHz.

values in clock cycles of a $chain_{max}$ *pattern*, as this is the pattern with the maximum number of non-propagating merger patterns in two subsequent image rows.

The processing latency increases linearly with the image width as shown in the diagram in Figure 3.19. When the *SLCCA* architecture is operated at 100 MHz, the processing latency is below $160\mu s$ even for *UHD8k* resolution. When operating the *SLCCA* architecture at a higher frequency (e.g. those shown in Figure 3.23), the processing latency decreases even further. Since the values from Figure 3.19 are a guaranteed upper bound for the processing latency, the *SLCCA* hardware architecture presented in this chapter is also suitable for the application in real-time image processing systems.



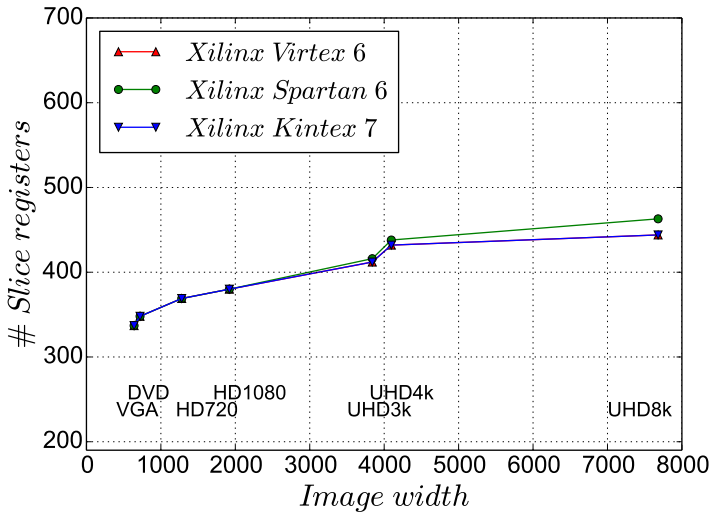

[65] ©2015 IEEE

**Figure 3.20:** This diagram shows the number of lookup tables (LUTs) required by the FPGA implementation of the SLCCA architecture for different image sizes and different FPGA families. *[65]*

### 3.2.3 Hardware Resources

'The FPGA hardware architecture of the *SLCCA* connected components analysis algorithm was described in *VHDL* and implemented for the Xilinx Virtex 6 VLX240T-2 (speedgrade -2, 40 nm technology), Xilinx Spartan 6 SLX150T-2 (speedgrade -2, 45 nm technology) and Xilinx Kintex 7 K325T-2L (speedgrade -2L, 28 nm technology) to explore the performance on different FPGA devices. To acquire comparable mapping and timing results, for the implementation on all FPGA devices the *PlanAhead 14 default* implementation strategy was used and nothing apart from the *SLCCA* architecture was implemented on the FPGA devices.

In the diagrams in Figure 3.20-3.22, the FPGA resources required for the implementation of the *SLCCA* architecture are shown for a number of typical image sizes from VGA to UHD8k for Kintex 7, Virtex 6 and Spartan 6 FPGAs. The diagram in Figure 3.20 shows the number of lookup tables (LUTs) which realise logic functions with up to 6 inputs [128, 134]. The number of slice registers is shown in the diagram in Figure 3.21. Both the number of LUTs and slice registers increase quasi-logarithmically with the image width. The number of slice registers is nearly identical for the three examined FPGA devices, and the number of LUTs varies between the device families depending on the image size. The Kintex 7 and Virtex 6 devices provide 18kBit and 36kBit BRAM resources, and the Spartan 6

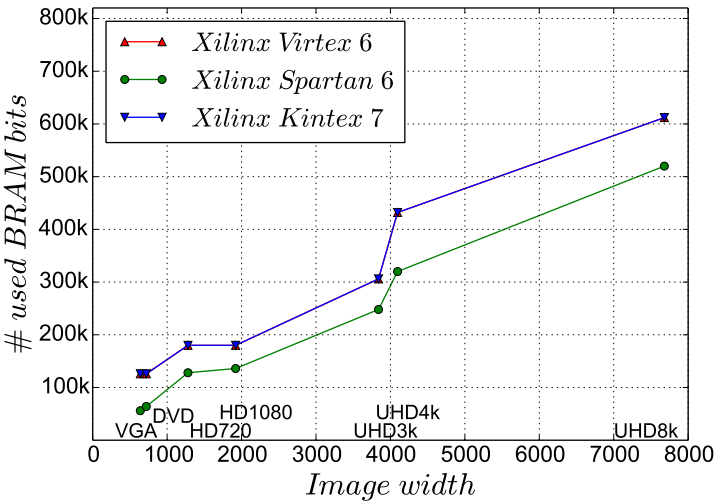



[65] ©2015 IEEE

**Figure 3.21:** This diagram shows the number of slice registers required by the FPGA implementation of the SLCCA architecture for different image sizes and different FPGA families. *[65]*

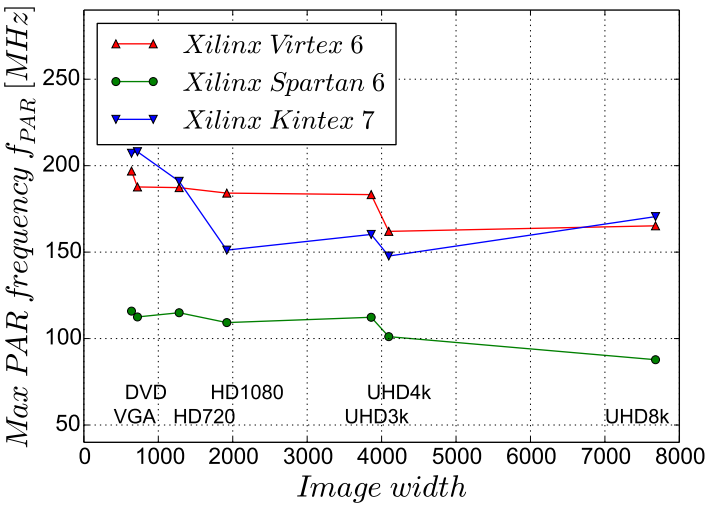
device BRAMs are 9kBit and 18kBit. Since the unused memory resources of a partially used BRAM are not available to other components on the FPGA, they are considered to be used for the comparison. This results in a different number of required BRAM bits for Spartan 6 and Virtex 6 or Kintex 7. The diagram in Figure 3.22 shows the number of used BRAMs for different image sizes. The number of required on-chip memories scales linearly with the image width. The throughput of the connected components analysis architecture is mainly proportional to the maximum operating frequency $f_{max}$ which is shown in the diagram in Figure 3.23 for different image sizes. For the implementation of the *SLCCA* architecture on Kintex 7 and Virtex 6, the maximum operation frequency $f_{max}$ is almost twice that implemented on the Spartan 6 which has a direct impact on the throughput.

The throughput can be classified into two parts: a static part with one pixel per clock cycle which is completely independent of the image content and a data-dependent part for resolving the label pair of a merger pattern (stored on $S$) depending on the image content. The data-dependent part lasts between 0 clock cycles, if the stack $S$ has no entries, and $\lceil \frac{W}{5} \rceil$ clock cycles per image row for the worst case pattern of Figure 3.18(e). Thus, considering the worst case pattern, an image stream of up to 166 megapixels per second can be processed in real-time (for VGA resolution). ’[65]




[65] ©2015 IEEE

**Figure 3.22:** This diagram shows the number of on-chip BRAM bits required by the FPGA implementation of the SLCCA architecture for different image sizes and different FPGA families. *[65]*




[65] ©2015 IEEE

**Figure 3.23:** This diagram shows the maximum operation frequency after the place&route (PAR) of the SLCCA hardware architecture for different image sizes and different FPGA families. *[65]*

| Algorithm | # Passes | Scan method | Connectivity | Worst case identified |
|---|---|---|---|---|
| Ma and Bailey [83] | Single-pass | Pixel-based | 8 | True |
| Zhao et al. [138] | Single-pass | Run-based | 4 | False |
| Bailey et al. [7]/ [58] | Single-pass | Pixel-based | 8 | True |
| Ito et al. [57] | Single-pass | Run-based | 4 | True |
| Appiah et al. [5] | Two-pass | Run-based | 8 | True |
| SLCCA | Single-pass | Pixel-based | 8 | True |

[65] ©2015 IEEE

**Table 3.8:** Comparison of the algorithm properties of CCA hardware architectures. *[65]*

| Architecture | Hardware device | Image size [Pixel] | Extracted FV | LUTs | Registers | BRAM [bit] |
|---|---|---|---|---|---|---|
| [83] | Virtex 2 | $640 \times 480$ | A, C | 1757 | 600 | 72k |
| [138] | Virtex 2 | $256 \times 256$ | A, FOM | 4587 | 3154 | 234k |
| [7, 58] | Spartan II | $670 \times 480$ | A, C | 810 | 286 | 16k |
| [57] | Stratix | $2k \times 2k$ | N/A | 1.5k-10k LE | | 53k-409k |
| [5] | Virtex 4 | $640 \times 480$ | N/A | 649 | 641 | 1142k |
| SLCCA | Kintex 7 | $256 \times 256$ | BB | 2158 | 988 | 108k |
| | | *UHD8k* | BB | 2819 | 1464 | 630k |

[65] ©2015 IEEE

**Table 3.9:** Comparison of several CCA hardware architectures with respect to hardware resources. Extracted feature vectors: (A) Area, (C) Component count, (FOM) First-order moment and (BB) Bounding box. *[65]*

### 3.2.4 Comparison to Other Hardware Architectures

' In Tables 3.8, 3.9 and 3.10, the algorithms and the implementations of several published CCA hardware architectures are compared.

These publications suggest a diverse variety of methods at the algorithmic level as well as at the architectural level and the hardware devices used for implementation. The differences between these architectures at the algorithmic level include the connectivity (either 4-connectivity of 8-connectivity), the scan method (either pixel by pixel processing or run processing), and the number of scans (either single-pass or two-pass). At the architectural level they differ in image size, extracted feature vector and the hardware device used.

All of these factors directly affect the maximum frequency the circuit of the CCA architecture can be operated at, which plays a major role in the achievable perfor-

| Architecture | Hardware device | Image size [Pixel] | $f_{max}$ [MHz] | Worst case throughput $[\frac{MPixel}{s}]$ |
|---|---|---|---|---|
| Ma and Bailey [83] | Virtex 2 | $640 \times 480$ | 40.64 | 32.5 |
| Zhao et al. [138] | Virtex 2 | $256 \times 256$ | 95.7 | N/A |
| Bailey et al. [7, 58] | Spartan II | $670 \times 480$ | N/A | N/A |
| Ito et al. [57] | Stratix | $2k \times 2k$ | 61-72 | 61-72 |
| Appiah et al. [5] | Virtex 4 | $640 \times 480$ | 49.73 | $\leq$24.86 |
| SLCCA | Kintex 7 | $256 \times 256$ | 156.4 | 124.2 |
| | | *UHD8k* | 135.04 | 99.1 |

[65] ©2015 IEEE

**Table 3.10:** Comparison of several CCA hardware architectures with respect to processing throughput. Extracted feature vectors: (A) Area, (C) Component count, (FOM) First-order moment and (BB) Bounding box. *[65]*

mance. As a basis for comparing the maximum throughput, the throughput of a worst case image stream is chosen because it provides a true upper bound for the processing time and, therefore, the applicability of the architecture for real-time processing. Depending on connectivity, scan method and the number of scans, the worst case image differs; some publications lack the identification of a worst case scenario. These aspects make a direct comparison difficult. For this reason, the results of each architecture from academic literature summarised in Tables 3.9 and 3.10 are compared individually to the *SLCCA* architecture presented in Section 3.1.

Comparison of the SLCCA architecture with the architecture by Ma and Bailey [83]

The architecture of [83] is the most resource efficient architecture reported in the literature to date. The key weakness of [83] is the requirement for two tables for merger management and to translate labels due to aggressive relabelling. This also requires use of two data tables, one for the old labels and one for the new labels. As memory resources scale linearly with image width, they are more critical for scalability. In Section 3.2.1, it is shown that the BRAM resources required for this work are fewer for all image sizes compared to [83]. In the presented implementation of the *SLCCA* architecture, the maximum throughput is more than three times higher, some of which will be a result of using a newer FPGA. The main advantage of the proposed algorithm is label reuse, reducing the memory requirements for storing feature vectors which reduces the amount of memory resource for the *SLCCA* by 42% (depending on the image size and the extracted feature vectors).

Comparison of the SLCCA architecture with the architecture by Zhao et al. [138]

In [138], a CCA architecture is presented processing the input image after performing run-length encoding. The architecture of [138] considers 4-connectivity which will give a different result to 8-connectivity used in the *SLCCA* architecture. Four-connectivity requires fewer comparisons per pixel providing a shorter critical path in the resulting hardware circuit. For an image size of $256 \times 256$, the authors state a throughput of around 90 Megapixel/s by eliminating the additional processing at the end of the image row. For 8-connectivity, CCA such a method cannot be found in the literature. The *SLCCA* architecture requires significantly fewer LUTs and registers for an architecture processing the same image size, as shown in Table 3.9. Another major advantage of the *SLCCA* algorithm compared to the algorithm used in [138] is the identification and analysis of the worst case image pattern which makes it applicable for real-time processing.

Comparison of the SLCCA architecture with the architecture by Bailey and Johnston [7, 58]

The CCA architecture in [7, 58] is an earlier version of [83] which is less hardware and memory efficient than [83]. It provides memory for 256 labels, i.e. does not cover a worst case scenario. Therefore, a meaningful comparison to this work is not possible.

Comparison of the SLCCA architecture with the architecture by Ito et al. [57]

The authors state that their architecture requires between $1.5k$ and $10k$ logic elements to process a four megapixel image, where one logic element is equal to a 4-input LUT and one flip-flop. The *SLCCA* architecture requires approximately 400 slice registers and 700 6-input LUTs to process a comparable image size, as shown in Figures 3.20 and 3.21. In [57], all of the information is obtained by local operations propagating tags to the next image row. In the *SLCCA* algorithm, the connection between distant component segments is identified by a global equivalence table (merger table $M$) which allows feature vectors to be extracted for arbitrarily shaped components. In [57], a measure for the level of concavity in image components is introduced. Therefore, it is possible that different labels are assigned to pixels of the same connected component if the level of concavity is exceeded which is a limitation of general applicability. The *SLCCA* architecture is able to extract the feature vectors of connected components of arbitrary shapes, i.e. there is no limit of the level of concavity.

Comparison of the SLCCA architecture with the architecture by Appiah et al. [5]

The architecture in [5], based on a two-pass algorithm, requires the complete image to be stored before the labelling process starts, i.e. large images cannot be processed completely on an FPGA due to a lack of sufficient on-chip memory. The two-pass algorithm of [5] requires a minimum of 2 clock cycles to process a single pixel. For stream processing, an additional buffer is required to store the pixels received while the second pass of the previous image is carried out. The *SLCCA* architecture uses a single-pass algorithm which does not require the complete image to be stored and, therefore, requires significantly less memory resources.'[65]

## 3.3 Summary and Contributions of the SLCCA Hardware Architecture to the State of the Art

'The *SLCCA* hardware architecture proposed in this chapter is an improvement to the state of the art on several levels, as pointed out in the following.

- **High-throughput processing of (worst case) binary image streams:** For processing a image stream consisting of worst case patterns, the proposed architecture achieves a throughput of up to 166 Megapixel/s. To the best of our knowledge this is the highest throughput of a connected component analysis architecture for 8-connectivity processing one pixel per clock cycle which has been achieved on an FPGA.
- **Using a novel control structure to detect the last pixel of an image object in the video stream at the earliest possible point in time:** This allows achieve a processing latency below $160\mu s$ even for image streams of 32 Megapixel images and contributes to reducing the number of required memory resources as these can be reused earlier.
- **Memory reduction by the recycling of labels:** At the architecture level, a novel *label recycling* scheme is introduced. In combination with the proposed method for detecting the last pixel of an object (*SLCCA* algorithm), the memory for storing feature vectors is halved compared to [83] by eliminating redundant data structures.
- **The realisation of the novel label recycling scheme from the *SLCCA* algorithm:** The label translation scheme of [83] using a dedicated translation table is simplified by reducing the number of lookups from two to one per label, as only a merger table is required.
- **A reduction of memory resources for the entire architecture:** As the *SLCCA* algorithm which lays the foundation for the *SLCCA* architecture is a single pass algorithm, the total memory required can be reduced by a factor of more than 200 compared to the classical connected components labelling algorithm which is a two-pass algorithm [102]. Depending on the extracted feature vector and image size, 42% or more of memory resources can be saved compared to an optimised state-of-the-art architecture [83][65].

# 4 *PCCA* - The Parallel SLCCA Algorithm

In this chapter, the parallel *SLCCA* algorithm is presented, which is a parallelisation of the *SLCCA* algorithm introduced in Chapter 2. The parallel *SLCCA* algorithm is denoted in the following as *PCCA*, while the *SLCCA* algorithm is denoted as *SLCCA*.

*PCCA* processes several pixels of the input image simultaneously. The major innovation of the *PCCA* algorithm compared to the *SLCCA* algorithm is parallel on-the-fly-processing of binary images in a single pass achieved by splitting up global data dependencies and processing them locally. A high throughput is achieved by dividing the input image into several vertical image slices which connected components only have local data dependencies. The feature vectors extracted from these vertical image slices are combined after parallel processing as the global dependencies are gathered while the image is processed. Each feature vector extracted by *PCCA* is, therefore, available only a few image rows after the last pixel of the associated connected component is received, instead of waiting until the end of the image. Even the most advanced parallel state-of-the-art CCA or CCL algorithms [12] for general-purpose processors combine the results of processed slices after the entire image is processed. The *PCCA* algorithm combines feature vectors of connected components spanning several image slices while the image is processed. This enables a low processing latency and facilitates the design of a memory-efficient and resource-efficient hardware architecture (introduced in Chapter 5) by label recycling.

The images in Figure 4.1 demonstrate the basic processing steps of the *PCCA* algorithm. Prior to processing a colour or greyscale image, such as the head MRI image in Figure 4.1(a), a segmentation step is required. The result of this step is shown in Figure 4.1(b), where a black pixel represents an object pixel and a white pixel background. The *PCCA* algorithm separates the binary input image into several vertical sub-images, each processed by an instance of the *SLCCA* algorithm (Figure 4.1(c)). This splits the connected components spanning several image slices into multiple component segments. The *PCCA* algorithm extends the *SLCCA* algorithm by a data structure and employs a method to efficiently memorise the connections of multiple component segments and combine their associated feature vectors into one feature vector per connected component. These steps are depicted in Figure 4.1(c) and (d).

The behaviour of the *PCCA* algorithm can be described as a *cut, memorise and combine* approach. The details and key properties of the *PCCA* algorithm addressed in this chapter are:

- *Parallelism and Scalability*: Achieving a high throughput by processing multiple pixel simultaneously
- *Memory-efficiency*: Reusing memory entries for several subsequent connected components in the image
- *Real-time processing*: Maximum time for combining $p$ slices is equal to processing a single slice with the SLCCA algorithm

These properties build the basis for the parallel hardware architecture in Chapter 5 to achieve a processing throughput of up to 6 Gigapixel/s for extracting feature vectors from an image stream.

The abbreviations and names of data structures used in the following are summarised in Table 4.1.

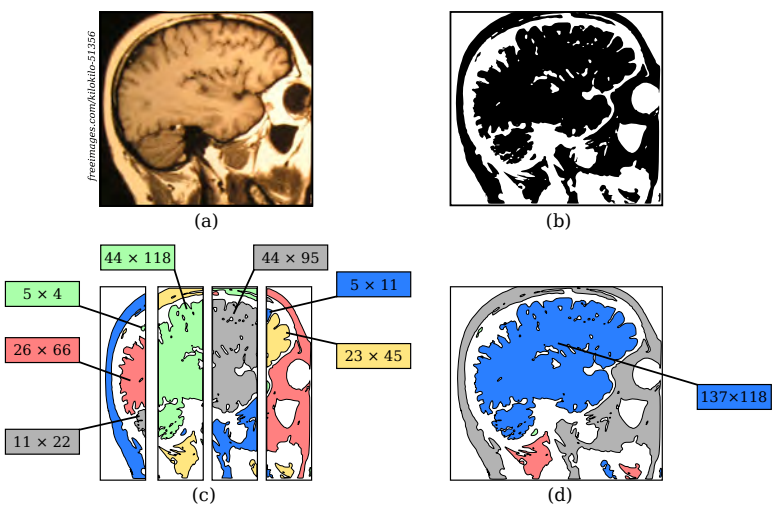



**Figure 4.1:** (a) Original colour image, (b) binary image after segmentation, (c) extracted feature vectors for each image slices and identification of connected components spanning several image slices, and (d) feature vectors of the input image after coalescing.
Extracted bounding box feature vectors are shown in black boxes (width×height in pixel). Connected components and component segments of connected components which span multiple image slices are assigned different colours.

| Data structures for slice processing | |
|---|---|
| Abbreviation | Name |
| $DT_i$ | Data table of SPI processing image slice $i$ |
| $F_{L,i}$ | Local label graph for image slice $i$ |
| $F_G$ | Global label graph |
| $FV$ | Feature vector |
| $G_P$ | Pixel graph |
| $I$ | Source image |
| $L$ | Labelled image |
| $M_i$ | Local merger table for image slice $i$ |
| Data structures for slice coalescing | |
| Abbreviation | Name |
| $GMT$ | Global merger table |
| $GDT$ | Global data table |
| $GLM$ | Global label management |
| Image parameters | |
| Abbreviation | Name |
| $H$ | Image height |
| $p$ | Total number of image slices |
| $p_i$ | Number sub-images of level $i$ |
| $W$ | Image width |
| $W_i$ | Width of sub-image of level $i$ |
| $W_S$ | Width of an image slice |
| $q$ | Number of level groups |

**Table 4.1:** Nomenclature used in this chapter.

The binary input image $I$ of dimensions $W \times H$ pixels is divided into $p$ slices of equal width. If the image width $W$ is not an integer multiple of the number of image slices $p$, the original input image $I$ is padded with $W \bmod p$ columns of background pixels at the right side. This makes each image slice $\lceil W_S = W/p \rceil$ pixels wide and $H$ pixels high. An image slice is scanned in forward raster scan order[1]. Let $(a, b)$ be a position in the left-most image slice $(0 \leq a < W_S \wedge 0 \leq b \leq H)$. Then, the $p$ positions $simPos(a, b)$ are processed simultaneously, where

$$simPos(a, b) = \{(a, b), (W_S + a, b), \ \ldots, \ (W_S \times (p - 1) + a, b)\}. \tag{4.1}$$

The parallel forward raster scan order starts at the top left corner of each image slice, i.e. the positions $simPos(0, 0)$ are processed in parallel. At the end of the first row

[1]See the definition of *forward raster scan order* from Section 2.1.

the positions $simPos(W_S - 1, 0)$ are processed in parallel, followed by $simPos(0, 1)$ in the next row, and so on until the end of the image $simPos(W_S - 1, H - 1)$ is reached. The $W_S \times H$-tuple describes this parallel raster scan:

$$\begin{aligned} parRasterScan \; = (&simPos(0,0),\; \ldots, simPos(W_S - 1, 0), \\ &simPos(0,1), \ldots,\; simPos(W_S - 1, H - 1)). \end{aligned} \tag{4.2}$$

The definitions of the pixel graph $G_P$, *connectedness*, *connected component* and *component segment* used in the following, can be found in Section 2.1.

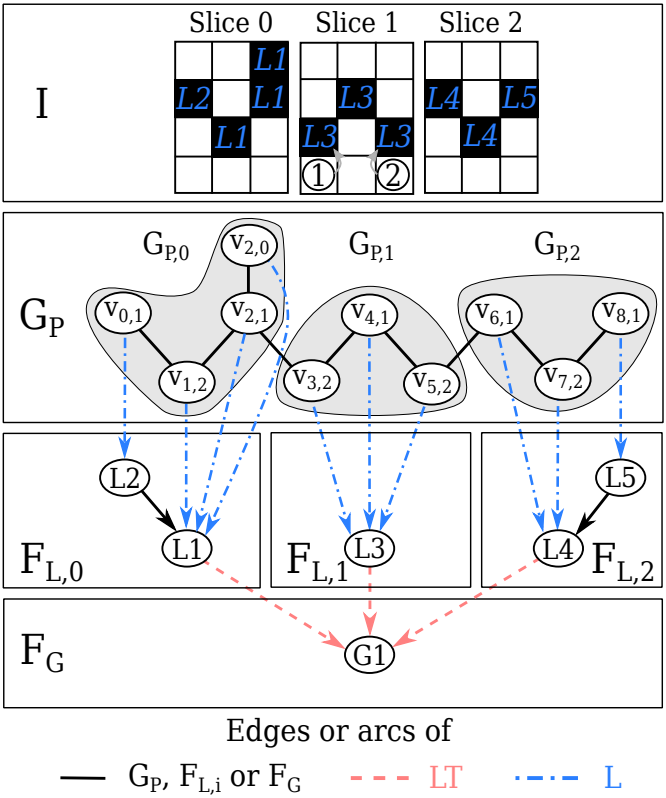



**Figure 4.2:** This figure shows the graphs resulting from the example image $I$ divided into three image slices: the pixel graph $G_P$, the local label graphs $F_{L,0}$, $F_{L,1}$, $F_{L,2}$, the labelled image $L$, the link table $LT$ and the global label graph $F_G$.

## 4.1 Parallel Labelling Process in PCCA

In this section the data structures used in the *PCCA* algorithm are introduced. For better understanding an example based on the image in Figure 4.2 is given after the definition of each data structure.

In *SLCCA* (Equation 2.6 and 2.8) a vertex $v_{a,b}$ in the *pixel graph* $G_P$ represents the object pixel of the input image $I$ at position $(a, b)$.

$$V(G_P) = \{v_{a,b} : I[(a,b)] = 1 \wedge (a,b) \in imagPos\}, \tag{4.3}$$

The set *imagePos* contains all image positions in $I$, as introduced in Equation 2.1. For each pair of adjacent pixels in $I$ there is an edge in the pixel graph $G_P$.

$$E(G_P) = \{(v_{a1,b1}, v_{a2,b2}) : v_{a1,b1} \in V(G_P) \wedge v_{a2,b2} \in V(G_P) \wedge \|(a1, b1) - (a2, b2)\| = 1 \wedge (a1, b1) \in imagPos \wedge (a2, b2) \in imagPos\}. \tag{4.4}$$

Dividing the input image $I$ to $p$ image slices, divides the pixel graph $G_P$, as well. These $p$ sub-graphs are referred to as $G_{P,0}, \ldots, G_{P,p-1}$ in the following. Each sub-graph $G_{P,i}$ contains all of the vertices of $V(G_P)$ associated with image slice $i$ and all of the edges of $E(G_P)$ which join two vertices of $G_{P,i}$. This makes $G_{P,0}, \ldots, G_{P,p-1}$ unconnected sub-graphs of $G_P$. In Figure 4.2 the pixel graph $G_P$ consists of ten vertices connected by nine edges. The vertices of the sub-graph $G_{P,0}$ are $v_{2,0}$, $v_{0,1}$ $v_{2,1}$ and $v_{1,2}$. The edges of $G_{P,0}$ are $(v_{2,0}, v_{2,1})$, $(v_{0,1}, v_{1,2})$ and $(v_{2,1}, v_{1,2})$.

**Definition 21.** *Instance of an algorithm*[2]: The instance of an algorithm refers to the application of this algorithm on a particular data structure, e.g. a particular graph or digraph, or a particular sub-graph or sub-digraph.

An instance of the *SLCCA* algorithm, for example, is the process of applying the *SLCCA* on a particular pixel graph $G_P$ or a particular sub-graph of a pixel-graph. Each instance of *SLCCA* maintains its own set of internal data structures, such as a label graph $F_L$. Each image slice of $I$ and its corresponding sub-graph of the pixel graph $G_P$ are processed by a separate instance of *SLCCA*. In *SLCCA*, the *labelled image* $L$ (see Section 2.3.1 for definition) is an array with the same dimensions as $I$. For *PCCA*, the image is separated into $p$ vertical slice of width $W_S$, where each image slice is processed by one instance of *SLCCA*. A provisional label is assigned to each position of $L$ by the associated *SLCCA* instance, where 0 is reserved for background. Each *SLCCA* instance maintains its own label graph $F$. Since each *SLCCA* instance has its own local label space, their label graphs corresponding to image slices $0, \ldots, p-1$ are called local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ in the following. For each vertex $v_{a,b} \in V(G_{P,i})$ the *SLCCA* instance processing image slice $i$ associates $v_{a,b}$ with a vertex $v_{L_j} \in V(F_{L,i})$ by assigning label $L_i[(a, b)] \coloneqq L_j$.

In Figure 4.2, the local label graph $F_{L,2}$ contains two vertices, labelled $L4$ and $L5$ joined by the arc $(L4, L5)$. The vertices $v_{6,1}$ and $v_{7,2}$ in sub-graph $G_{P,2}$ are associated with $L4$ by assigning $L[(6, 1)] \coloneqq L4$ and $L[(7, 2)] \coloneqq L4$. The vertex $v_{8,1}$ in sub-graph $G_{P,2}$ is associated with $L5$ by assigning $L[(8, 1)] \coloneqq L5$.

The edges of the pixel graph $G_P$ are categorised as *slice edges*, $E_s$, and *border edges*, $E_b$. Slice edges join two vertices of the same sub-graph $G_{P,i}$.

$$E_s(G_{P,i}) \coloneqq \{(v_1, v_2) : v_1, v_2 \in V(G_{p,i})\}. \tag{4.5}$$

[2]This definition for the *instance of an algorithm* is based on [47, p.29]

Border edges join two vertices of different sub-graphs $G_{P,i}$ and $G_{P,i+1}$.

$$E_b \coloneqq \{(v_1, v_2) : v_1 \in V(G_{P,i}), v_2 \in V(G_{P,i+1}), 0 \leq i < p, i \in \mathbb{N}_0\}. \tag{4.6}$$

**Definition 22.** *Slice-component*: A sub-graph $K$ of the pixel graph $G_P$ is a *slice-component* if $K$ is a connected component of exactly one of the sub-graphs $G_{P,0}, \ldots, G_{P,p-1}$, and one or more vertices of $K$ are joined by a border edge $E_b$.

Connected components spanning several image slices consist of several slice-components (see definition below) and their vertices are joined by border edges in $E_b$. These slice-components are again combined to connected components after the simultaneous processing of $G_{P,0}$ to $G_{P,p-1}$, according to *SLCCA*. This connection process is achieved by performing one union-find operation for each border edge in $E_b$ which is carried out by one or more instances of the union-find algorithm, as explained in the following section.

The global label graph $F_G$ is required to identify which of the extracted feature vectors have to be combined. The global label graph $F_G$ is a directed forest structure to identify vertices of the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ associated with slice-components and associate them with their connected components. The arcs from vertices in the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ to vertices in the global label graph $F_G$ are stored in the *link table LT*. A root vertex $L_j$ in local label graph $F_{L,i}$, $v_{L_j} \in V(F_{L,i})$, associated with a slice-component, is joined with a vertex $G_k$ of $F_G$ (corresponding to $L_j$'s connected component) by assigning $LT_i[L_j] \coloneqq G_k$. In Figure 4.2, the pixel graph $G_P$ of the example image $I$ contains two border edges: $(v_{21}, v_{32})$ and $(v_{52}, v_{61})$. The root vertices of the associated slice-components are $L1$, $L3$ and $L4$. Since all of these slice-components are associated with the same connected component associated with the vertex $G1$ in the global label graph $F_G$, the arcs $(L1, G1)$, $(L3, G1)$ and $(L4, G1)$ are stored by updating $LT$.

In order to process multiple slice-components of the same connected component in parallel, each slice-component is at first associated with vertices of its local label graph $F_{L,0}, \ldots, F_{L,p-1}$ by one of the $p$ *SLCCA* instances. In the next step, the *PCCA* algorithm associates all trees in $F_{L,0}, \ldots, F_{L,p-1}$ from the same connected component with a common tree structure in the global label graph $F_G$. Details are given in the following.

## 4.2 Parallel Union-find Operations in PCCA

The union-find operations to create the forest structures in the local label graphs are covered in Section 2.2. To accelerate processing, union-find operations on the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ and the global label graph $F_G$ are processed in parallel. In general, several parallel union-find operations on the same tree structure can lead to false results [100]. However, as $G_{P,0}$ to $G_{P,p-1}$ are not connected,

processing each sub-graph with a separate instance of the union-find algorithm (as done in *SLCCA*) separates the initial problem into $p$ smaller sub-problems. This allows to carry out $p$ union-find operations simultaneously, one on each of the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$.

**Definition 23.** *Union-find instance*: A union-find instance is an instance of the union-find algorithm.

The tree structures in the global label graph $F_G$ are associated with multiple slice-components. A naive parallelisation where several union-find instances carry out operations on the same tree structure of the global label graph $F_G$ simultaneously without any restrictions can, therefore, result in the race condition described in [100]. In the following, *hierarchical distributed union-find (HD-UF)* (Algorithm 6), an algorithm to carry out multiple union-find operations in parallel on the same tree structure of the global label graph $F_G$ is described.

*HD-UF* avoids the race condition described in [100] by restricting the access to a sub-set of vertices of tree structures in the global label graph $F_G$ to single union-find instances. The goal of *HD-UF* is to build trees with several levels of hierarchy in the global label graph $F_G$, where each level in the hierarchy is also a directed tree. The parallel processing is achieved by applying multiple union-find instances on sub-trees of the global label graph $F_G$, where each union-find instance transfers operations to a parent union-find instance. As global labels are created at slice borders, the tree structures in the global label graph $F_G$ are created with a hierarchy required for this kind of parallelisation, if the image is divide hierarchically to slices, as well. In the following this idea is evolved to the *hierarchical distributed union-find (HD-UF)* algorithm presented in Algorithm 6. In the following the idea of hierarchical slicing leading to a hierarchical label graph is evolved into the *hierarchical distributed union-find (HD-UF)* algorithm presented in Algorithm 6.

At first, the entire input image $I$ is divided into $p_0$ image slices. In this first step, these image slices and the borders between them are assigned level 0. Each slice of level $i$ is further divided into $p_{i+1}$ slices of level $i+1$. For $q$ levels, the total number of image slices, $p$, is the product of the number of sub-slices of each level, from 0 to $q-1$, i.e.

$$p = \prod_{i=0}^{q-1} p_i. \tag{4.7}$$

For each slice level $i$, the image slices are assigned a slice identifier from 0 to $(p_0 \times \ldots \times p_i) - 1$, beginning with the left-most slice. The example in Figure 4.3 shows that separating slice ① (with slice identifier 2) creates two slices of level 1: slice ② and slice ③. Slice ① is called the super-slice of slice ② and slice ③. Slices ② and slice ③ are called the sub-slices of slice ①. The terms super-slice and sub-slice explained in this example are used in the following, too.

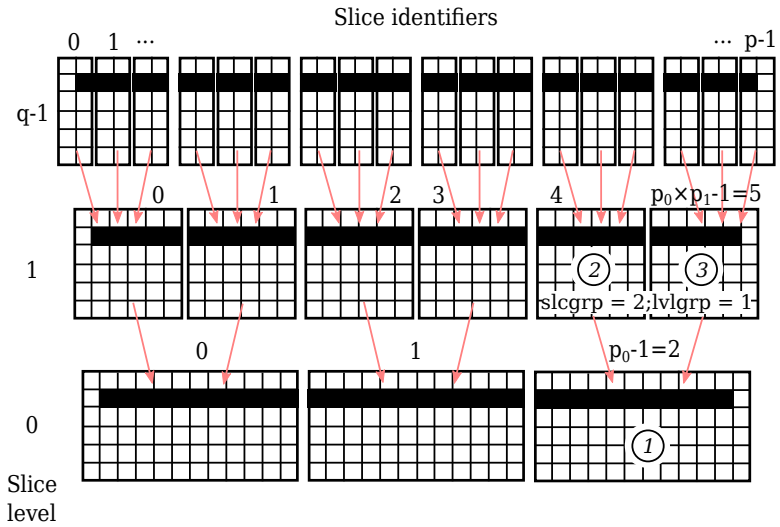



**Figure 4.3:** The input image $I$ is separated into $p$ vertical slices processed as sub-images. At first, the image is divided into $p_0$ image slices of level 0. These are further separated into image slices with a level $> 0$. A red arrow indicates the super-slice of a slice. In this example, $p_0 = 3$, $p_1 = 2$, $p_2 = 3$, leading to a total of $p = 18$ image slices.

To distinguish the vertices of different union-find instances, the properties *index*, *lvlgrp* and *slcgrp* are introduced. A vertex $v$ of the global label graph $F_G$ is generated by a *makeSet* operation (explained in detail later in this section). This *makeSet* operation is induced by a connected component crossing an image border. To associate vertex $v$ with the image border it is generated at, the concepts of *level groups (lvlgrp)* and *slice groups (slcgrp)* are introduced.

**Definition 24.** *Level group - lvlgrp:* The level group of a vertex $v$ from the global label graph $F_G$, *lvlgrp(v)*, is the level of the slice border at which vertex $v$ is generated.

**Definition 25.** *Slice group - slcgrp:* A vertex $v$ of the global label graph $F_G$ is generated at a slice border which divides an image slice with slice identifier $j$ to sub-slices. To associate vertex $v$ with image slice $j$, vertex $v$ is in slice group $j$, *slcgrp(v)=j*.

The vertices in the global label graph $F_G$ with *lvlgrp*$= 0$ are all in *slcgrp*$= 0$. The *index* of a vertex in the global label graph $F_G$ is a unique identifier among vertices with the same *lvlgrp* and *slcgrp* of $F_G$ which is used to access the associated data structures. In the example in Figure 4.3, the image slice marked as slice ① is in *lvlgrp*$= 0$ and *slcgrp*$= 0$. Slice ① is divided into the image slices marked slice ② and slice ③. Both, slice ② and slice ③ are in *lvlgrp*$= 1$ as they are the result of

dividing a slice of *lvlgrp*= 0. Since the slice identifier of slice ① is 2, both slice ② and slice ③ are in *slcgrp*= 2.

The sub-graph of the global label graph $F_G$ whose vertices all have the same *lvlgrp* $i$ and the same *slcgrp* $j$, is referred to as sub-graph $F_{i,j}$ and is defined as

$$\begin{aligned} V(F_{i,j}) =& \{v : v \in V(F_G) \wedge lvlgrp(v) = i \wedge slcgrp(v) = j\}, \\ E(F_{i,j}) =& \{(a,b) : a \in V(F_{i,j}) \wedge b \in V(F_{i,j})\}. \end{aligned} \tag{4.8}$$

To distribute workload, multiple union-find instances working in parallel are used to execute union-find operations on the global label graph $F_G$. One union-find instance is associated with each sub-graph $F_{i,j}$ of $F_G$. As $F_G$ only contains directed trees, a tree-like hierarchy of union-find instances comes into being, where each union-find instance of *lvlgrp* $i$ ($i > 0$) transfers operations (explained in the next section) to one instance of *lvlgrp* $i-1$. Each union-find instance can only access its associated sub-graph of the global label graph $F_G$: $F_{i,j}$. Instructions on vertices associated with other union-find instance cannot be carried out directly, but require further communication, a topic dealt with in Section 4.4.2.

To handle these multiple levels of hierarchy correctly, the *find* and the *union* operation, introduced in Section 2.2, are modified resulting in a *hierarchical distributed union-find (HD-UF)* algorithm (Algorithm 6), as discussed in the following.

**Definition 26.** *Sub-root vertex:* The vertices of the global label graph $F_G$ in *lvlgrp* $i$ which are children of a vertex in $F_G$ with *lvlgrp* $< i$ are referred to as *sub-root* vertices.

In classical union-find algorithms introduced before, such as *QuickFind* (Algorithm 1) or *QuickUnion* (Algorithm 2), the *find* operation always returns the root label. The *find* operation of *HD-UF* (Algorithm 6) returns either the root vertex of a tree structure or a sub-root vertex.

In the following, the *union* operation of *HD-UF* (Algorithm 6), which carries out different instructions on the union-find data structure to join the tree structures that vertices $e$ and $f$ are a part of, is explained. The sub-root or root vertices of $e$ and $f$ which are the result of the find operation of *HD-UF*, are referred to as $sr_0$ and $sr_1$. The parents of sub-roots $sr_0$ and $sr_1$ are referred to as $psr_0$ and $psr_1$, respectively. The way in which the *union* operation of *HD-UF* (Algorithm 6) processes vertices $e$ and $f$ depends on the *lvlgrp* and *slcgrp* of $sr_0$, $sr_1$, $psr_0$ and $psr_1$. The vertex of $sr_0$ and $sr_1$ whose parent is in the lower *lvlgrp* is called $sr_{min}$. The vertex whose parent is in the higher *lvlgrp* is called $sr_{max}$. The assignment of $sr_{min}$ and $sr_{max}$ is shown in the *HD-UF* (Algorithm 6) in lines 16-17. The parent vertices of $sr_{min}$ and $sr_{max}$ are referred to as $psr_{min}$ and $psr_{max}$ (*HD-UF* in Algorithm 6 line 18), respectively.

**Algorithm 6:** *Hierarchical distributed union-find (HD-UF) algorithm.*

```
   // MakeSet operation to create a vertex and assign a lvlgrp and slcgrp to it.
1  makeSet (vertex e, lvlgrp l, slcgrp s)
2      e.lvlgrp := l
3      e.slcgrp := s
4      parent[e] := ∅

   // Find operation to determine the parent vertex e in the same lvlgrp and
      execute path compression
5  find (vertex e)
6      if (parent[e]=∅) ∨ (parent[e].lvlgrp<e.lvlgrp then
7          return e // Return vertex e when the parent of e is either root or
                sub-root
8      else
9          r := find(parent[e])
10         parent[e] := r
11         return r

   // Union operation to join vertices dependent on their lvlgrp and slcgrp.
12 union (vertex e, vertex f)
       // Determining lvlgrp and parents
13     sr_0 := find(e) ; psr_0 := parent[sr_0]
14     sr_1 := find(f);  psr_1 := parent[sr_1]
15     if psr_0.lvlgrp ≤ psr_1.lvlgrp then
16         sr_min := sr_0; sr_max := sr_1
17     else  sr_min := sr_1; sr_max := sr_0
18     psr_min := parent[sr_min] ; psr_max := parent[sr_max]
19     if sr_0 ≠ sr_1 then
20         if sr_0.lvlgrp = sr_1.lvlgrp then
21             if sr_min.slcgrp ≠ sr_max.slcgrp then
22                 if psr_min.lvlgrp = sr_max.lvlgrp then
                       // Case (f) in Figure 4.5
23                     makeSet(g, sr_min.lvlgrp-1)
24                     union(g,sr_min)
25                     union(g,sr_max)
26                 else // Case (g) in Figure 4.5
27                     union(psr_min,sr_max)
28             else if (psr_0.lvlgrp ≠ sr_0.lvlgrp) ∧ (psr_1.lvlgrp ≠ sr_1.lvlgrp) then
                   // Case (c) in Figure 4.4
29                 parent[sr_max] := sr_min
30                 union(psr_min, psr_max)
31             else  parent[sr_max] := sr_min //Case(a&b) in in Figure 4.4
32         else
33             if psr_0 = ∅ ∧ psr_1 = ∅ then
                   // Case (e) in Figure 4.4
34                 parent[sr_max] := sr_min
35             else union(psr_min, psr_max) // Case (d) in Figure 4.4
```

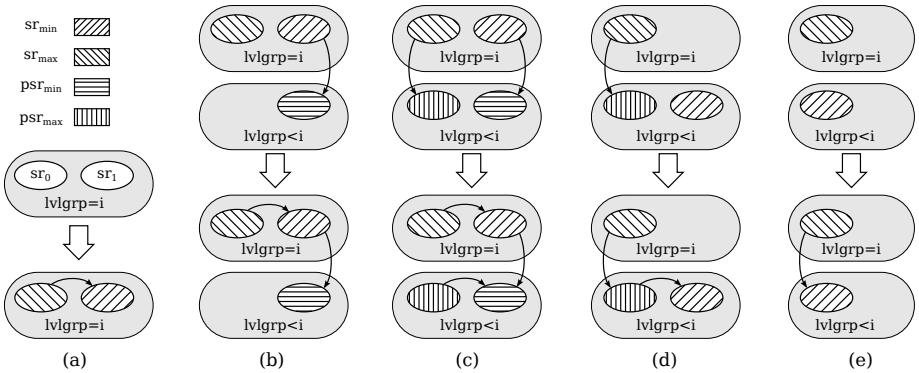



**Figure 4.4:** Different scenarios from *HD-UF* (Algorithm 6) for joining vertices of the global label graph $F_G$ with the same *slcgrp* dependent on their *lvlgrp*.

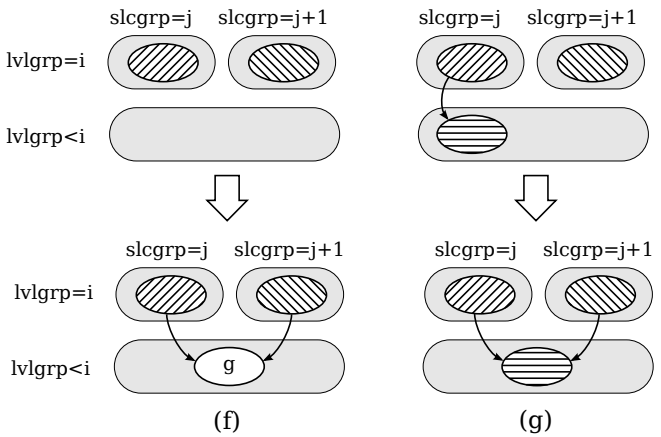



**Figure 4.5:** Different scenarios from *HD-UF* (Algorithm 6) for joining vertices of the global label graph $F_G$ associated with different component segments dependent on their *slcgrp*.

The possible combinations of these properties are depicted in Figure 4.4 and 4.5 and the instructions of *HD-UF* (Algorithm 6) in lines 19 to 35 are discussed in the following. In cases *(a)* to *(e)* in Figure 4.4, the vertices $e$ and $f$ are in the same *slcgrp*. In cases *(f)* and *(g)* shown in Figure 4.5 the *slcgrps* of vertex $e$ and $f$ are different.

(a) The vertices $e$ and $f$ and their root vertices $sr_0$ and $sr_1$ are in the same *lvlgrp*. Therefore, $sr_{min}$ is made the parent of $sr_{max}$.

(b) The vertices $e$ and $f$ are in the same *lvlgrp* as one sub-root and one root vertex of $e$ and $f$. Therefore, $sr_{min}$ becomes parent of $sr_{max}$.

(c) Both sub-roots of $e$ and $f$ are in the same *lvlgrp* as $e$ and $f$ and the parent vertices of both are in a lower *lvlgrp*. Therefore, $sr_{min}$ is made parent of $sr_{max}$ and a union operation is performed on $psr_{min}$ and $psr_{max}$.

(d) Sub-root vertex $sr_{min}$ is in the same *lvlgrp* as $psr_{max}$, which is lower than the *lvlgrp* of $sr_{max}$. As $psr_{min}$=$sr_{min}$, a union operation is performed on $psr_{min}$ and $psr_{max}$.

(e) The sub-root vertices of $e$ and $f$ are in different *lvlgrps*, therefore, $sr_{min}$ is made the parent of $sr_{max}$.

(f) The sub-root vertices of both $e$ and $f$, $sr_{min}$ and $s_{max}$, are in the same *lvlgrp*, but in different *slcgrps*. To join them, a new vertex $g$ with a lower *lvlgrp* is created by a *makeSet* operation and two union operations are issued to join $sr_{min}$ and $s_{max}$ root vertices with the new vertex $g$.

(g) The sub-root vertices of both $e$ and $f$, $sr_{min}$ and $s_{max}$, are in the same *lvlgrp*, but in different *slcgrps*. The parent of $sr_{min}$, $psr_{min}$, is in a lower *lvlgrp*. Therefore, $sr_{max}$ is made a child of $psr_{min}$ by issuing a union operation to the union-find instance that $sr_{max}$ is associated with.

The *union* operation of the *HD-UF* algorithm (Algorithm 6) recursively joins the sub-root vertices of $e$ and $f$ if they are in a lower *lvlgrp* for cases *(c)* and *(d)*, as shown in Figure 4.4. In the cases *(f)* and *(g)* the union operation is applied on vertices of different sub-graphs $F_{i,j}$, $F_{k,l}$ of the global label graph $F_G$, where $i \neq k$ or $j \neq l$. Each union-find instance only has access to the parents of its associated vertices. Therefore, the *makeSet* instruction and the union instruction on $sr_{max}$ must be forwarded to the associated union-find instances.

Since there is only a single *sub-root vertex* per connected component in each sub-graph $F_{i,j}$ of the global label graph $F_G$ if *union* from *HD-UF* (Algorithm 6) is applied, the union-find instructions are distributed among all union-find instances. Since *find* from *HD-UF* (Algorithm 6) always returns the sub-root vertex or the root vertex, several union-find instances can perform union-find operations on the same tree structure of the global label graph $F_G$ simultaneously. Each instance carries out operations on its associated vertices. This enables parallel union-find processing on all sub-graphs $F_{i,j}$ simultaneously by restricting the access to a vertex to its associated union-find instance. In this way, the race condition described in [100] is avoided.

## 4.3 Global Operations

The graphs in Figure 4.2 show an example of the data structures used by PCCA to associate every object pixel of input image $I$ with exactly one connected component. First, the object pixels of each image slice are associated with a vertex of their local label graph $F_{L,0},\ \ldots,\ F_{L,p-1}$. This labelling process is described in detail in Chapter 2. The *PCCA* algorithm deals with associating vertices of the local label graphs $F_{L,0},\ \ldots,\ F_{L,p-1}$ of the same connected component which span multiple slices created by multiple *SLCCA* instances (each processing one image slice) with a common forest structure in the global label graph $F_G$. The global operations presented in Pseudocode 8 (*global operations*) use a combination of *union-find* and *lookup* operations to build a forest structure in which a directed path from each vertex in the pixel graph $G_P$ to a root vertex in $F_{L,0},\ \ldots,\ F_{L,p-1}$ or to a root vertex in $F_G$ exists. This associates every object pixel of input $I$ with its connected component. Since *union*, *find* and *makeSet* operations are applied on the local label graphs $F_{L,0},\ \ldots,\ F_{L,p-1}$ and the global label graph $F_G$, the notation

**operation** $(\{vertices, lvlgrp, slcgrp\}, graph)$

is used. For instance, *union({e,f},$F_{L,5}$)* carries out a union operation on vertex $e$ and $f$ which both belong to the local label graph associated with image slice 5, $F_{L,5}$. The steps of the global operations (*global operations* are shown in Pseudocode 8) define the union-find operations for joining slice-components associated tree structures in different local label graphs $F_{L,0},\ \ldots,\ F_{L,p-1}$, which are connected in the pixel graph $G_P$ via one or multiple border edges from $E_b$.

A *global new label operation* (GNLO) is invoked if vertices $e$ and $f$ from the label graphs $F_{L,0},\ \ldots,\ F_{L,p-1}$ representing local labels are assigned to adjacent border positions of different neighbouring slices and both are not associated with a vertex of the global label graph $F_G$ via $LT$. The *GNLO* associates the root vertices of $e$ and $f$ from the local label graphs $F_{L,0},\ \ldots,\ F_{L,p-1}$ with a new vertex in the global label graph $F_G$, *newGL*, generated by a *makeSet* operation.
A *global merger operation, global to local,* ($\mathrm{GMO}_{GL}$) is invoked if the local labels associated with the vertices $e$ and $f$ are assigned to different image slices and $e$ is associated with a vertex $F_G$ via an arc in $LT$, and $f$ is not. In this case, an arc is added to $LT$ to associate the root vertex of $f$ with the same vertex in $F_G$.
A *global merger operation, global to global,* ($\mathrm{GMO}_{GG}$) is induced if the local labels associated with the vertices $e$ and $f$ are assigned to different image slices and both are associated with different vertices of $F_G$ via an arc in $LT$.
If the local labels associated with the vertices $e$ and $f$ are assigned to the same image slice and both are associated with different vertices of $F_G$ via an arc in $LT$, a *global merger operation within a slice* ($\mathrm{GMO}_{Slice}$) operation is induced. Both merger operations, $\mathrm{GMO}_{GG}$ and $\mathrm{GMO}_{Slice}$, carry out a union operation on the vertices in the global label graph $F_G$ which are associated with $e$ and $f$. A $\mathrm{GMO}_{Slice}$, additionally,

**Pseudocode 8:** *Global operations* — Union-find operations carried out by global operations

```
1  GNLO(vertex e, vertex f)
2      makeSet({newGL, lvlgrp, slcgrp}, F_G)
3      LT[find(e,F_{L,i})] := newGL
4      LT[find(f,F_{L,j})] := newGL
5  GMO_GL(vertex e, vertex f)
6      LT[find(f,F_{L,i})] := LT[find(e,F_{L,i})]
7  GMO_GG(vertex e, vertex f)
8      h:= LT[find(e,F_{L,i})]
9      k:= LT[find(f,F_{L,j})]
10     union({h,k},F_G)
11 GMO_Slice(vertex e, vertex f)
12     i:=find(e,F_{L,i}); h:= LT[find(i,F_{L,i})]
13     j:=find(f,F_{L,i}); k:= LT[find(j,F_{L,i})]
14     union({i,j},F_{L,i})
15     union({h,k},F_G)
```

carries out a union operation on the root vertices of $e$ and $f$ in $F_{L,0}, \ldots, F_{L,p-1}$. This reduces the number of tree structures in $F_{L,0}, \ldots, F_{L,p-1}$ associated with a single connected component to one per slice.

## 4.4 Partitioning of the PCCA Algorithm

The global operations described in Pseudocode 8 (*global operations*) cover the functionality to detect slice-components and associate them with their connected component. In this section, the *PCCA* algorithm is partitioned to match the distributed memory model and the architecture to execute instructions in parallel of state-of-the-art computing systems, such as multi-core CPUs or FPGAs. These optimisations include:

- *Parallelisation*:
  Distributing workload to several processing instances
- *Memory efficiency*:
  Reduction of the memory requirements by recycling the memory allocated by vertices of completed connected components
- *Memory distribution*:
  Usage of several small distributed memories instead of a single large one

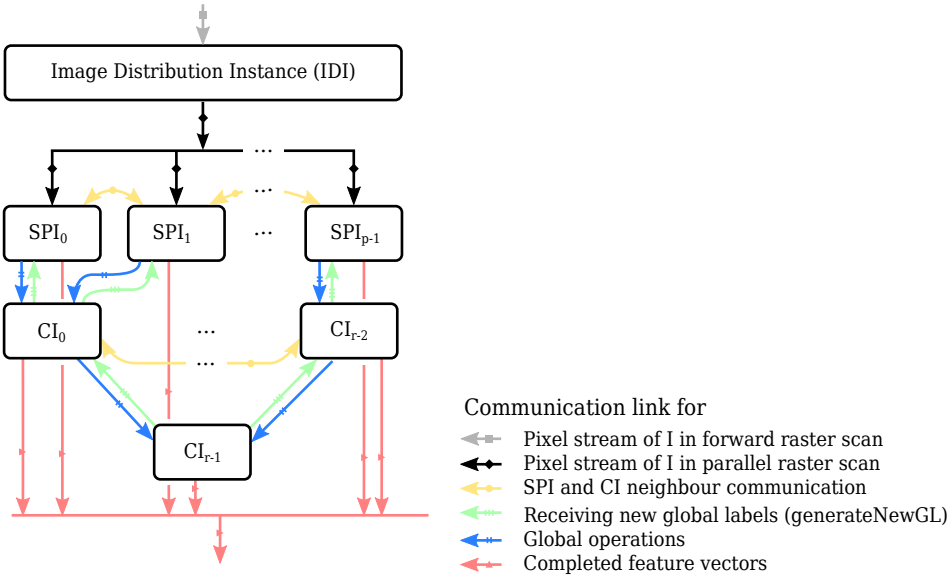



**Figure 4.6:** Interaction of constituents of *PCCA*.

The process of detecting connections of slice-components and the extraction of their feature vectors is described in Section 4.4.1. Section 4.4.2 explains how to combine the feature vectors of these slice-components efficiently to the feature vectors of their connected components. The following paragraph, *parallel processing and communication in PCCA*, introduces the constituents of PCCA and their interaction. This is followed by the paragraph *positions and labels in PCCA*, which introduces the naming conventions and the terminology used in this chapter for assigning labels to the *labelled image L*.

## Parallel Processing and Communication in PCCA

The *PCCA* algorithm (Algorithm 7) receives a binary pixel stream in forward raster scan order *I_RS* and outputs the extracted feature vectors of connected components in *I_RS* to *FV_Extr*. The pseudocode in Algorithm 7 shows the instances processing pixel data and control instructions simultaneously to carrying out *PCCA*. In fact, the instance in each line of Algorithm 7 processes input data independently of the instances described in the other lines of Algorithm 7. So, rather than a sequential order of execution, Algorithm 7 describes the communication of the individual instances processing data in parallel, forming *PCCA*. This communication is depicted in detail in Figure 4.6 to show the different types of communication between the instances introduced in the following.

The binary image received as pixel stream $I\_RS$ is divided into $p$ image slices by the *image distribution instance (IDI)* which generates $p$ binary image streams in parallel raster scan order, $I\_PRS_0, \ldots, I\_PRS_{p-1}$ (*parRasterScan*, see Equation 4.2). Each of these binary image streams is forwarded to one of the $p$ slice processing instances (*SPI*). Each *SPI* is an instance of *SLCCA* introduced in Chapter 2 extended by the ability to detect connected components crossing a slice border. The feature vectors of component segments belonging to the connected component are called adjacent feature vectors. To associate adjacent feature vectors with the same connected component, *coalescing instances (CI)* are introduced which keep a record of which feature vectors are adjacent by assigning the same global labels to the pixels of their component segments. Each *SPI* communicates to its neighbour *SPIs* to detect adjacent feature vectors. After detection, a global label is acquired by an *SPI* by calling the sub-routine *generateNewGL* (explained later in Pseudocode 16). Adjacent feature vectors are transferred from an SPI to a CI via global operations (*GOs*), as discussed in Section 4.4.1 and Section 4.4.2. As GOs are processed sequentially within a *CI* to accelerate processing, the workload is distributed among several CIs. These coalescing instances $CI_0, \ldots, CI_{r-1}$ are arranged in a tree, where each CI processes GOs (Pseudocode 15) and generates new global labels (Pseudocode 16), as discussed in detail in Section 4.4.2. All SPIs and CIs output feature vectors simultaneously; these are therefore concatenated to the vector $FV\_Extr$ (Algorithm 7, line 6).

**Algorithm 7:** *PCCA* — The parallel *SLCCA* algorithm.

**Input**: I_RS
**Output**: FV_Extr

// Image distribution instance (IDI), see Equation 4.2
1 **IDI**(Input→I_RS, Output→I_PRS)

// Slice processing instances $0, \ldots, p-1$
2 **SPI**$_0$(In→(I_PRS$_0$, CI$_{Prnt}$.generateNewGL(), ...),
Out→(toPrntCI$_{SPI,0}$, SPI_FV$_0$, ...))
...
3 **SPI**$_{p-1}$(In→(I_PRS$_{p-1}$, CI$_{Prnt}$.generateNewGL(), ...),
Out→(toPrntCI$_{SPI,p-1}$, SPI_FV$_{p-1}$, ...))

// Coalescing instances $0, \ldots, r-1$
4 **CI**$_0$:
// Pseudocode 15
processGO(In→(toPrntCI$_{chld}$, ...), Out→(toPrntCI$_{CI,0}$, FV_CI$_0$, ...) )
// Pseudocode 16
generateNewGL()
...
5 **CI**$_{r-1}$:
processGO(In→(toPrntCI$_{chld}$, ...), Out→(toPrntCI$_{CI,r-1}$, FV_CI$_{r-1}$, ...) )
generateNewGL()

// Concatenation of feature vectors extracted by SPI and CI
6 FV_Extr := (FV_SPI$_0$, ..., FV_SPI$_{p-1}$, FV_CI$_0$, ..., FV_CI$_{r-1}$)

## Positions and Labels in PCCA

This subsection introduces the terminology used to describe labels and positions in the following. A *local label* has a one-to-one relation to vertex from one of the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$. Local labels are assigned to the labelled image $L$. The local labels associated with image slice $i$ are in columns $i \times W_S$ to $(i+1) \times W_S - 1$ of $L$. The local label $L_i[(a,b)]$ refers to the local label in slice $i$ of $L$ located in column $a$ and row $b$.

$$L_i[(a,b)] = L[(a + i \times W_S, b)]. \tag{4.9}$$

A *global label* corresponds to a vertex in global label graph $F_G$. The relation of global and local labels is stored in the *link table LT*. The neighbourhood $\eta_L$ for selecting the local label of the current position $X = (x, y)$ is different depending on which column of the image slice is processed. In the first column of an image slice, positions $B = (x, y-1)$ and $C = (x+1, y-1)$ of the current slice are considered to select the local label of the current pixel. In the last column positions $A = (x-1, y-1)$,

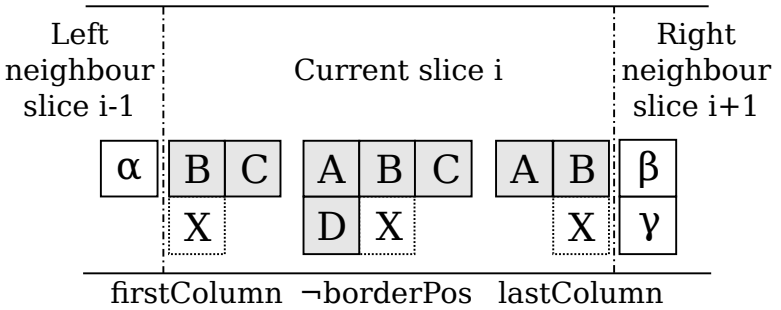



**Figure 4.7:** Neighbourhood positions considered in the first column, last column and in between.

$B = (x, y-1)$ and $D = (x-1, y)$ are considered. $L_A, \ldots, L_D$ are the labels assigned to $L[A], \ldots L[D]$, as shown in Figure 4.7. Since the same local label is assigned to the labelled image $L$ at position $A$ and $D$, $L[A]$ and $L[D]$, if they are object pixels [67], this label is referred to as $L_{AorD}$. The global label associated with $L_{AorD}$ by an arc in the link table $LT$ is $G_{AorD}$. The local label $L_B$ is associated via an arc in the $LT$ with $G_B$, and the local label $L_C$ is associated via an arc in the $LT$ with $G_C$.

A position in *imagePos* (see Section 2.1), which is either in the first or last column of an image slice, is a border position,

$$borderPos = firstColumn \vee lastColumn. \tag{4.10}$$

If the current pixel is not at a border, the positions $A$ through $D$ are considered for $\eta_L$.

$$\eta_L \coloneqq \begin{cases} \{B, C\}, & firstColumn, \\ \{A, B, D\}, & lastColumn, \\ \{A, B, C, D\}, & \neg borderPos. \end{cases} \tag{4.11}$$

The set $L_\eta$ contains the local labels in the neighbourhood $\eta_L$ which are not background label 0.

$$L_\eta \coloneqq \{L[i] : i \in \eta_L, L[i] \neq 0\}. \tag{4.12}$$

The local label in $L_\eta$ assigned first to $L$ in a raster scan[3] is referred to as $L_X$ [65], and is assigned to $L$ as the provisional label of the current position:

$$L[(x, y)] \coloneqq L_X. \tag{4.13}$$

[3]Section 2.3.5 deals with the method to determine the (local) label assigned first in raster scan.

An entry in the $LT$ represents an arc from a vertex in one of the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ to a vertex in the global label graph $F_G$. The link table $LT$ is a 1-D array with an entry for each local label associated with a vertex in one of the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$. Each of these entries contains the global label associated with a vertex of $F_G$. If a vertex in $F_{L,0}, \ldots, F_{L,p-1}$ is not associated with a vertex in the global label graph $F_G$, the corresponding entry in $LT$ is zero. The global labels in the neighbourhood $\eta_L$, $G_\eta$, are the global labels associated with the local labels in $L_\eta$.

$$G_\eta \coloneqq \{LT[i] : i \in L_\eta, LT[i] \neq 0\}. \tag{4.14}$$

Equation 4.15 shows that the minimum label in $G_\eta$ is associated with $L_X$ by extending the *local operations* (new label, label copy and merger operation) defined in Section 2.2. The next available global label *newGL* is associated with $L_X$ when a global new label pattern (*GNLPat*) is detected, which is explained in Section 4.4.1. If there are no global labels in the neighbourhood of the current pixel $L_X$ ($G_\eta = \emptyset$) then the link table entry of $L_X$ ($LT[L_X]$) is set to 0. Otherwise, the minimum of the global label $G_\eta$ is associated with $L_X$.

$$LT[L_X] \coloneqq \begin{cases} newGL, & GNLPat, \\ 0, & G_\eta = \emptyset, \\ \min\{G_\eta\}, & otherwise. \end{cases} \tag{4.15}$$

The reference to a global label is always available in the link table $LT$. The operation from Equation 4.15, therefore, propagate global labels in raster scan direction along, together with the associated local labels.

To select the global label of a border pixel, the global labels associated with border pixels of the neighbour slices are considered, too. The positions and labels of border pixels are denoted as follows. The position to the top left of a left border pixel of the current image row $y$ is denoted as $\alpha$. The position below $\alpha$ is not considered for labelling as it succeeds the current position $X = (x, y)$ in parallel raster scan. The positions to the right and top right of the right border pixel of the current image row $y$ are denoted as $\beta$ and $\gamma$. The global labels associated with these positions are called $G_\alpha$, $G_\beta$ and $G_\gamma$. $G_\beta$ and $G_\gamma$ have the same global label if their associated pixels are not background, therefore, their label is referred to as $G_{\beta or \gamma}$.

This makes the positions in the neighbourhood of global labels $\eta_G$ relevant for selecting a global label for the current pixel.

$$\eta_G \coloneqq \begin{cases} \{\alpha, B, C\}, & firstColumn, \\ \{A, B, D, \beta, \gamma\}, & lastColumn, \\ \{A, B, C, D\}, & \neg borderPos. \end{cases} \tag{4.16}$$

Figure 4.7 summarises all the positions relevant for local and global labelling depending on the position in the image. Those for selecting a local label are marked in grey. For global labelling the positions marked white or grey positions are relevant.

### 4.4.1 Slice Processing Instance

A *slice processing instance* (*SPI*) is an instance of *SLCCA* extended with the ability to issue global operations. It extracts feature vectors of all connected components in an image slice and detects connected components spanning slice borders.

An *SPI* extracts feature vectors from image slice $i$ by associating all vertices of a connected component in $G_{P,i}$ with the same tree structure in local label graph $F_{L,i}$. In the following, the current $\text{SPI}_i$ processes image slice $i$. Its data structures are identified by subscript $i$. The data structure of neighbour slices are identified by subscripts $i-1/i+1$. To combine the slice-components of neighbour slices, an SPI identifies *global operations (GO)* from the pixel patterns in its image slice and the border pixels of its neighbours' image slices. These *GOs* correspond to the union-find instructions from *HD-UF* (Algorithm 6). Their instructions are carried out on SPIs and *coalescing instances (CIs)*. Each *SPI* carries out instructions on its own data structures and issues *GOs* to its associated *CI* to induce operations on the global label graph $F_G$. Each *SPI* is associated with exactly one *CI*, in the following referred to as the *SPI's parent CI*. The operations that the *CIs* carry out are introduced in Section 4.4.2. There are two sub-categories of global operations: *global label operations (GLOs)* and *global combination operations (GCOs)*. *GLOs* deal with associating local labels of slice-components with a global label. In addition, *GLOs* associate all vertices of global labels assigned to slice-components with the tree structure in the global label graph $F_G$. *GCOs* deal with combining the feature vectors of slice-components (from different image slices) to a single feature vector for each connected component. The instructions of *GOs* carried out by the SPIs are described in the following. Instructions of *GOs* carried out by CIs are discussed in Section 4.4.2. Since *GO* are data dependent, they contain the row number of their detection as an arbitration tag to establish their order in the CI. The necessity for these arbitration tags is discussed in Section 4.4.2.

Global Operations in the SPIs

The image patterns which induce *global operations* depend on local labels in $L$ and the global labels which they are associated with. The arc from the vertex of a local label in slice $i$ to a vertex of a global label is stored in the link table $LT_i$. The link table of the SPI processing the left neighbour slice is $LT_{i-1}$. The link table of the SPI processing the right neighbour slice is $LT_{i+1}$.

In Equation 4.17 to Equation 4.22, for convenience, the local labels and global labels are mapped to Boolean variables. The Boolean value of local or global labels which is equal 0, is *False* or otherwise *True*. To improve the readability, these Boolean variables use the same names as their corresponding global or local labels.

A *global new label pattern* (GNLPat) is detected for two adjacent object pixels belonging to different image slices whose local labels are both not associated with a global label.

$$GNLPat \coloneqq borderPos \wedge L_X \wedge \neg G_X \wedge (L_\alpha \wedge \neg G_\alpha \vee L_{\beta or \gamma} \wedge \neg G_{\beta or \gamma}). \quad (4.17)$$

The instructions carried out in an SPI when a *GNLPat* is detected are shown in Pseudocode 9 (*GLNO in SPI*). To induce the parent CI to create the vertex associated with global label *newGL* in the global label graph $F_G$, a *GNLO* is sent to the *CI*. The new global label is requested from the parent CI. Details of the instructions within the CI to generate a global label are given in Section 4.4.2 (Pseudocode 16, *generateNewGL*). Sending a *GNLO* containing the global label *newGL* and the arbitration tag *rowNo* to the parent *CI* is denoted as *toPrntCI(GNLO(newGL,rowNo))* in Pseudocode 9 (*GLNO in SPI*) and in the following text. Both the local label assigned to the current position and the local label assigned to a neighbour slice are joined with *newGL* by updating the link tables of the current and the neighbour SPI. If a *GNLPat* is detected in the first column of a slice $i$, $LT_{i-1}$ is updated; if detected in the last column $LT_{i+1}$ is updated. Figure 4.8 shows an example of an image containing a *GNLPat*, the tree structures in the local label graphs $F_{L,i}$, $F_{L,i-1}$ and the global label graph $F_G$ before and after the *GNLO* is executed.

If the current position is not at a border ($\neg borderPos$), a *global merger pattern within a slice* ($GMPat_{Slice}$) is identified for two different local labels in the neighbourhood $L_\eta$, and two different global labels in the neighbourhood $G_\eta$. This requires $L_{AorD} \neq L_C$ and $G_{AorD} \neq G_C$, as described in Equation 4.18.

$$\begin{aligned} GMPat_{Slice} \coloneqq & \neg borderPos \wedge L_x \wedge L_{AorD} \wedge L[C] \\ & \wedge L_{AorD} \neq L_C \wedge G_{AorD} \wedge G_C \wedge G_{AorD} \neq G_C. \end{aligned} \quad (4.18)$$

The instructions that are carried out in an SPI when a $GMPat_{Slice}$ is detected are shown in Pseudocode 10 ($GMO_{Slice}$ *in SPI*). The local label assigned to the current pixel $L_X$ is associated with the minimum global label by updating the link table. In addition, a $GMO_{Slice}$ to join the vertices of the global labels in $F_G$ is sent to

**Pseudocode 9:** *GLNO in SPI* — Instructions of GNLO carried out in an SPI

```
1 if GNLPat then
      // A new global label, newGL, is requested from the
         associated CI, see Pseudocode 16
2     newGL := generateNewGL()
3     LT_i[L_X] := newGL
4     if firstColumn then
5         LT_{i-1}[L_α] := newGL
6     else if lastColumn then
7         LT_{i+1}[L_{βorγ}] := newGL
8     toPrntCI(GNLO(newGL,rowNo))
```

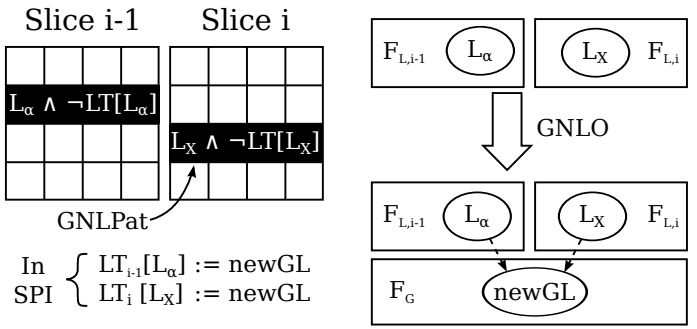



**Figure 4.8:** The global new label pattern detected in the first column of the current image slice $i$ induces a *global new label operation*. This associates the vertices of the component segments labelled $L_\alpha$ and $L_X$ with the vertex of global label *newGL*.

the parent CI. In *SLCCA*, using the minimum local label is not sufficient to obtain the current local label when local labels are recycled (see Section 2.3.5). In *PCCA*, assigning the minimum global label to the current pixel is sufficient as *QuickUnion with path compression* (Algorithm 3) is used by the *CI*, as the find operation of *QuickUnion with path compression* always obtains the root vertex.

Figure 4.9 shows an example image in which $L_C$ precedes $L_A$ in raster scan order ($L_C \prec L_A$) and $G_A < G_C$. Therefore, the vertex of $L_C$ becomes the parent of the vertex of $L_A$. By updating $LT$, $L_C$ is associated with the minimum global label $G_A$.

A *global merger pattern from a global to a local label* ($GMPat_{GL}$) is detected for two adjacent object pixel from different slices of the labelled image $L$ which are assigned local labels. In addition, a $GMPat_{GL}$ requires that exactly one of these

**Pseudocode 10:** *$GMO_{Slice}$ in SPI* — Instructions of $\text{GMO}_{Slice}$ carried out in an SPI

```
1 if GMPat_Slice then
2     parent := min(G_AorD, G_C)
3     child := max(G_AorD, G_C)
4     LT_i[L_X] := parent
5     toPrntCI(GMO_Slice(parent,child,rowNo))
```

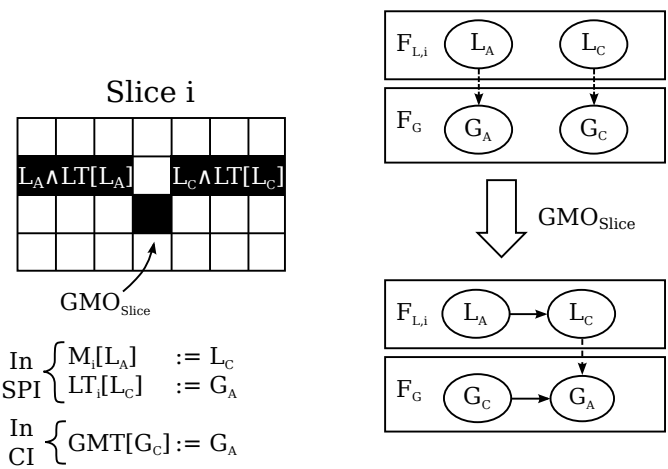



**Figure 4.9:** The $GMPat_{Slice}$ detected in image slice $i$ induces a $GMO_{Slice}$. This makes $L_A$ a child of $L_C$, and $G_C$ a child of $G_A$. The arcs in $LT$ are removed and a new arc from $L_C$ to $G_A$ is added.

two local labels is associated with a global label. The condition $GL_{left}$ of Equation 4.19 detects merger patterns in the first column of an image slice, where either a global label is associated with the current ($G_X$) or with the adjacent pixel in the left neighbour slice ($G_\alpha$). Condition $GL_{right}$ of Equation 4.19 detects merger patterns in the last column of an image slice, where either a global label is associated with the current ($G_X$) or with the adjacent pixel in the right neighbour slice ($G_{\beta or \gamma}$). A $GMPat_{GL}$ is, therefore, detected when $GL_{right}$ or $GL_{left}$ hold.

$$\begin{aligned} GL_{left} &:= firstColumn \wedge ((G_X \wedge L_\alpha) \oplus G_\alpha), \\ GL_{right} &:= lastColumn \wedge ((G_X \wedge L_{\beta or \gamma}) \oplus G_{\beta or \gamma}), \\ GMPat_{GL} &:= L_X \wedge (GL_{left} \vee GL_{right}). \end{aligned} \tag{4.19}$$

The instructions carried out in an SPI when a $GMPat_{GL}$ is detected are shown in Pseudocode 11 (*$GMO_{GL}$ in SPI*) and explained in the following.

If the local label $L_X$ is joined with the global label $G_X$, the local label in the neighbour image slice ($L_\alpha$ or $L_{\beta or \gamma}$) is joined with $G_X$ also. To join $L_\alpha$ with $G_X$, the link table $LT_{i-1}$ of the SPI processing the left neighbour slice is updated. The process of joining $L_{\beta or \gamma}$ with $G_X$ updates the link table $LT_{i+1}$ of the SPI processing the right neighbour slice. If the vertex of $L_\alpha$ is joined with a vertex $G_\alpha$ in the global label graph $F_G$, the local label of $L_X$ is joined with the global label $G_\alpha$ by updating the link table of the current SPI, $LT_i$. Analogous to $L_\alpha$, the following holds true for $L_{\beta or \gamma}$: If the vertex of $L_{\beta or \gamma}$ is joined with a vertex $G_{\beta or \gamma}$ in the global label graph $F_G$, the local label of $L_X$ is joined with the global label $G_{\beta or \gamma}$ by updating the link table of the current SPI, $LT_i$. The parent CI is informed about this event by sending a global merger operation $GMO_{GL}$.

**Pseudocode 11:** *$GMO_{GL}$ in SPI* — Instructions of $\mathrm{GMO}_{GL}$ carried out in an SPI

```
1  if GMPat_GL then
2      if G_X then
3          if firstColumn then
4              LT_{i-1}[L_α] := G_X
5          else if lastColumn then
6              LT_{i+1}[L_βorγ] := G_X
7      else
8          if firstColumn then
9              G_X := LT_{i-1}[L_α]
10         else if lastColumn then
11             G_X := LT_{i+1}[L_βorγ]
12         LT_i[L_X] := G_X
13     toPrntCI(GMO_GL(G_X, rowNo))
```

In Figure 4.10, an example of an image with a $\mathrm{GMPat}_{GL}$ is shown. The digraphs on the right show the tree structures in $F_{L,i}$, $F_{L,i-1}$ and the global label graph $F_G$ before and after carrying out $\mathrm{GMO}_{GL}$.

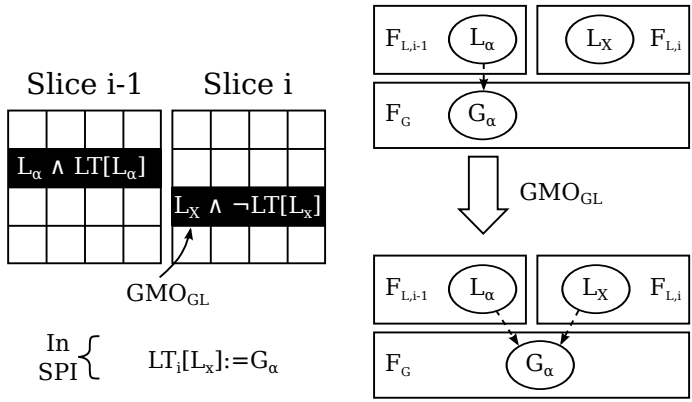



**Figure 4.10:** The $GMPat_{GL}$ detected in the first column of the current image slice $i$ induces a $GMO_{GL}$. This associates the vertices of the component segments labelled $L_X$ with $G_\alpha$.

A *global merger pattern from a global label to a global label* ($GMPat_{GG}$) is detected for two adjacent local labels assigned to different slices of the labelled image $L$, where both local labels are associated with different global labels, as shown in Equation 4.20.

$$GMPat_{GG} \coloneqq borderPos \wedge G_X \wedge (G_\alpha \vee G_{\beta or \gamma}) \wedge (G_X \neq G_\alpha \vee G_X \neq G_{\beta or \gamma}). \quad (4.20)$$

The instructions carried out in an SPI when a $GMPat_{GG}$ is detected are shown in Pseudocode 12 ($\text{GMO}_{GG}$ in SPI). To induce the parent CI to associate the vertices of both global labels in the global label graph $F_G$, a $GMO_{GG}$ and both global labels are sent to the parent CI.

**Pseudocode 12:** *$GMO_{GG}$ in SPI* — Instructions of $\text{GMO}_{GG}$ carried out in an SPI.

```
1 if GMPat_GG then
2     parent := min{G_η}
3     child := max{G_η}
4     toPrntCI(GMO_GG(parent, child, rowNo))
```

In Figure 4.11, an example of an image with a $\text{GMPat}_{GG}$ is shown. The digraphs on the right show the tree structures in local label graphs $F_{L,i}$, $F_{L,i-1}$ and the global label graph $F_G$ before and after carrying $\text{GMO}_{GG}$. In the example shown, $G_X$ is assumed to be smaller than $G_\alpha$. Therefore, $G_\alpha$ becomes a child of $G_X$.

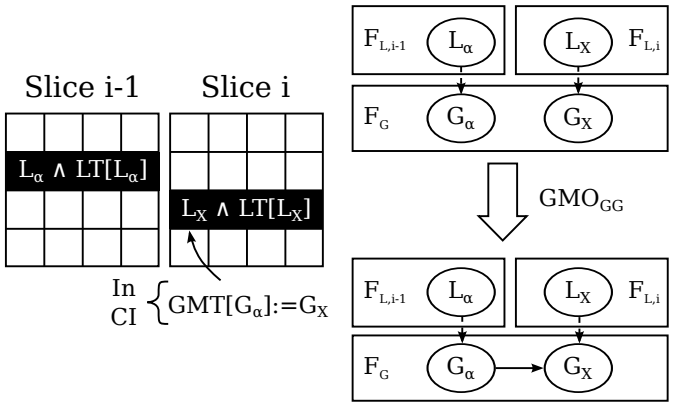



**Figure 4.11:** The $GMPat_{GG}$ detected in the first column of the current image slice $i$ induces a $GMO_{GG}$. In this example, $G_X < G_\alpha$, therefore, vertex $G_\alpha$ becomes a child of $G_X$.

Simultaneous Global Operations

Due to the simultaneous processing of $p$ image slices, multiple simultaneous global patterns in different *SPIs* can be detected, that cannot be processed independently. Therefore, there is a maximum of $p-1$ simultaneous global patterns in an image row, e.g. a connected component which spans all $p$ image slices. Simultaneous global patterns $simGP_i$ in two neighbour slices $i$ and $i-1$ are detected if both the Boolean conditions $sim_1$ and $sim_2$ from Equation 4.22 hold true.

$$simGP_i \coloneqq sim_1 \wedge sim_2. \tag{4.21}$$

$$\begin{aligned} sim_1 &\coloneqq (GNLPat_i \vee GMPat_{GL,i}) \wedge (GNLPat_{i-1} \vee GMPat_{GL,i-1}), \\ sim_2 &\coloneqq (L_i[0,y] = L_i[W-1,y-1]) \vee (L_i[0,y-1] = L_i[W-1,y-1]). \end{aligned} \tag{4.22}$$

The condition $sim_1$ holds true if in both slices $i$ and $i-1$ either an *GNLPat* or a $GMPat_{GL}$ is detected. The condition $sim_2$ holds true if the local label assigned to the labelled image slice $L_i$ at the left border is equal to the local label assigned to the right border.

The image slices are scanned in parallel raster scan (see Equation 4.1 and Equation 4.2). Simultaneously detected global operations at positions $simPos(x,y)$ are, therefore, based on the tree structures created from processing the previous positions in parallel raster scan order. The previous positions are $simPos(x-1,y)$, or $simPos(W-1,y-1)$ if the current positions are $simPos(0,y)$. The GLOs carried

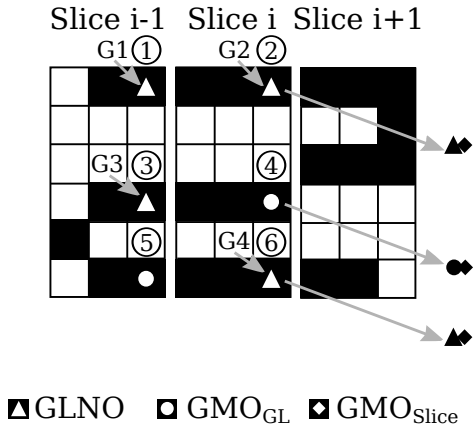



**Figure 4.12:** The global patterns in image slice $i-1$ and $i$ are detected simultaneously. Since SPI $i-1$ and $i$ only have a knowledge of the direct neighbour slices, image segments are associated with multiple global labels. This is detected and adapted by inducing a GMO$_{Slice}$ operation.

out by SPI$_{i-1}$ and SPI$_i$, therefore, join one local label with two different global labels by updating the link tables $LT_{i-1}$ and $LT_i$. This is resolved by associating $L_X$ of the current SPI$_i$, $L_{X,i}$, with the smaller global label of $G_{X,i}$ and $G_{X,i-1}$, as shown in Pseudocode 13 (*simultaneous global operations*). To join the vertices of $G_{X,i}$ and $G_{X,i-1}$ in the global label graph $F_G$ and to undo the double association of $L_{X,i}$, a GMO$_{Slice}$ is sent to the parent CI.

**Pseudocode 13:** *Simultaneous global operations* — Instructions carried out in an SPI if two simultaneous global merger patters are detected

```
1 if simGP then
2     parent := min(G_{X,i}, G_{X,i-1})
3     child := max(G_{X,i}, G_{X,i-1})
4     LT_i[L_X, i] := parent
5     toPrntCI(GMO_Slice(parent, child, rowNo))
```

Figure 4.12 shows an example of an image in which several global merger patterns are detected simultaneously. The global new label patterns ($GNLPat$) at positions ① and ② induce global new label operations creating two vertices in the global label graph $F_G$ labelled $G1$ and $G2$, as shown in Figure 4.13(a). The slice-components of the common vertex of local label graph $F_{L,i}$ are associated with both vertices, $G1$ and $G2$. At position ② a *simGP* is detected and as shown in Figure 4.13(a) (by executing *simultaneous global operations* Pseudocode 13) an arc from $G2$ to $G1$

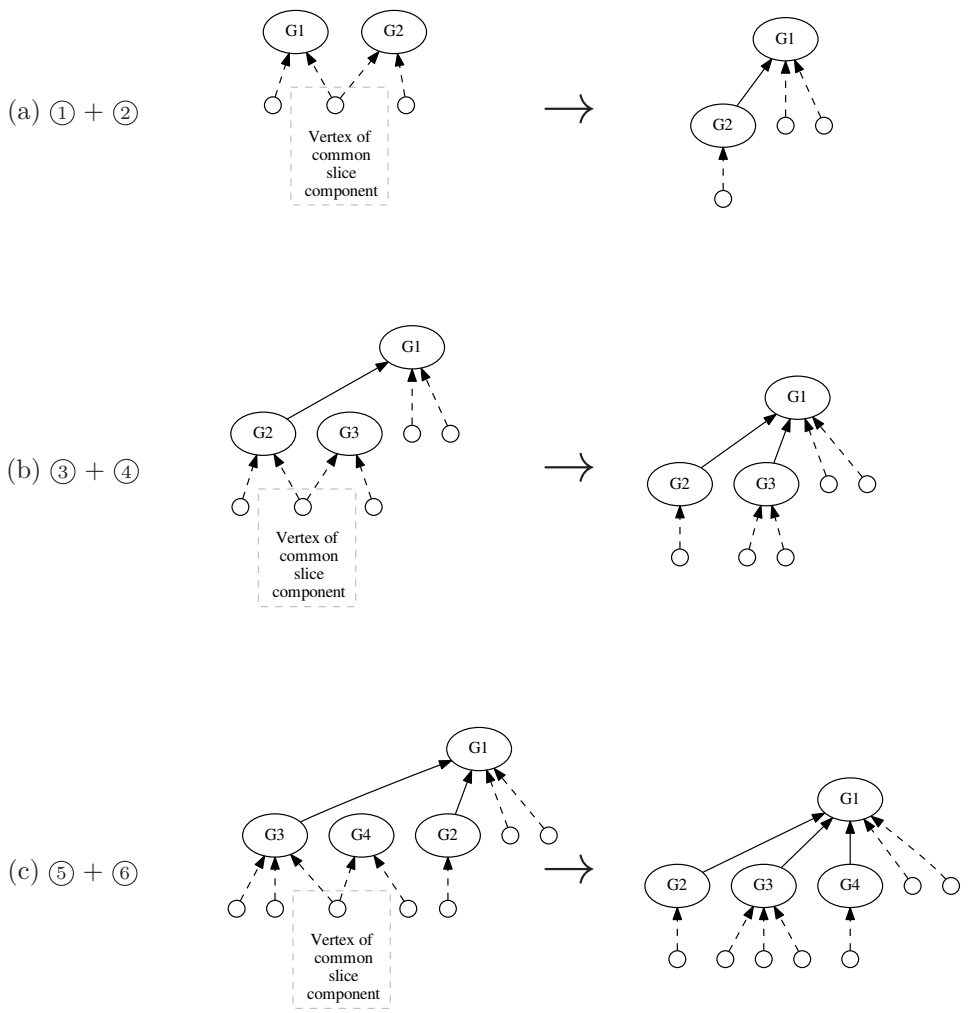



**Figure 4.13:** These images show the global label graph $F_G$ and the local label graphs $F_{L,i-1}$, $F_{L,i}$ and $F_{L,i+1}$ after the global operations induced by the patterns at positions ① to ⑥ from Figure 4.12. The large vertices $G1$ to $G4$ are from the global label graph $F_G$. The small vertices are of $F_{L,i-1}$, $F_{L,i}$ and $F_{L,i+1}$. Dashed arcs are from $LT$, solid ones from the global label graph $F_G$.

is created in $LT$. This is depicted in Figure 4.13(a). Since $G1$ and $G2$ belong to the same connected component, a global merger operation, $GMO_{Slice}$, is invoked to join $G1$ and $G2$ in the CI. The global label operation induced by the *GNLPat* at position ③ creates global label $G3$. The slice-component of the common vertex of local label graph $F_{L,i}$ is associated with the slice-components of the vertices labelled $G2$ and $G3$, as shown in Figure 4.13(b). Since a *simGP* is detected at position ④, a global merger operation is issued ($\text{GMO}_{Slice}$) to join the vertices in the global label graph $F_G$ labelled $G2$ and $G3$. The patterns at positions ⑤ and ⑥ are processed analogously which results in the tree structures shown in Figure 4.13(c).

GCOs in the Slice Processing Instance

*Global combination operations (GCOs)* involve combining the feature vectors of slice-components to a single feature vector for each connected component. A *GCO* is issued to the CI by an SPI for each completed[4] slice-component. A slice-component is a connected component which spans at least two image slices. Each *GCO* consists of the feature vector of its slice-component, the global label associated with its slice-component and an arbitration tag. The row number that a *GCO* is detected in is used as an arbitration tag, as discussed in Section 4.4.2.

[4]The property *completed* is defined in Section 2.3.5, Equation 2.28.

### 4.4.2 Coalescing Instance

A *coalescing instance (CI)* executes global operations (Pseudocode 8) which carry out union-find operations on the global label graph $F_G$ and combines feature vectors of slice-components to feature vectors of connected components.

When more than one CI is used, they are arranged in a tree. There is one root CI of *lvlgrp* 0. An SPI detects global operations and transfers them to the CI. Coalescing instances issue global labels of *lvlgrp*=$i$ to SPIs when a *global new label operation* is detected at an image border of level $i$. Each CI with *lvlgrp* $i$ ($i > 0$) is connected to one CI with *lvlgrp* $i - 1$ which is called its parent CI. CIs with the same *lvlgrp* with the same parent CI are referred to as sibling CIs.

All CI instances simultaneously receive global operations (GOs) to combine feature vectors of slice components issued by child SPIs, child CIs or sibling CIs. However, the GOs must be carried out in the same order that the associated image patterns appear in the original input image. The two types of GOs, GLOs and GCOs, are issued simultaneously, and therefore are buffered in separate queues referred to as *GLOq* and *GCOq*. GCOs received at the input of the CI contain feature vectors of slice-components. *Global label operations (GLOs)* contain union-find instructions issued by SPIs or CIs. The union-find instructions contained in GLOs describe which feature vectors to combine. In the following, the data structures to store the global label graph $F_G$ and the feature vectors are introduced. The operations on these data structures and necessary control information are explained in detail. A mechanism to establish the order of GOs is introduced.

#### Data Dependencies of Global Operations

All of the $p$ SPIs issue GOs simultaneously. In a CI of the highest *lvlgrp* this requires one *GLOq* and one *GCOq* for each connected SPI. CIs of a lower *lvlgrp* receive GOs from their child CIs, i.e. require one *GLOq* and one *GCOq* for each child CI. In this parallel en-queuing process, the data dependencies between the GOs resulting from the order of the raster scan are not considered. Figure 4.14 illustrates an example showing the deviation of the order of GOs in queues and their data dependencies due to the round-robin policy employed as explained in the following.

The GOs in Figure 4.14 associated with the black slice-components are already processed. The *GOs* associated with these black slice-components are, therefore, not shown in the queues in Figure 4.14. The hatched slice-component (blue and red in Figure 4.14) in slice 0 induces four GLOs and two GCOs, which are buffered in the queues $GLO_{q0}$ and $GCO_{q0}$. The hatched slice-component in slice 1 induces one GCO buffered in $GCO_{q1}$. The GCOs at the head of $GCO_{q0}$ and $GCO_{q1}$ are dependent on the GLOs in $GLO_{q0}$, as indicated by dashed arrows. The arbitration tags are assigned in the SPI and correspond to the row number in which a GO is detected. To meet the data dependencies, the GOs are read according to increasing arbitration

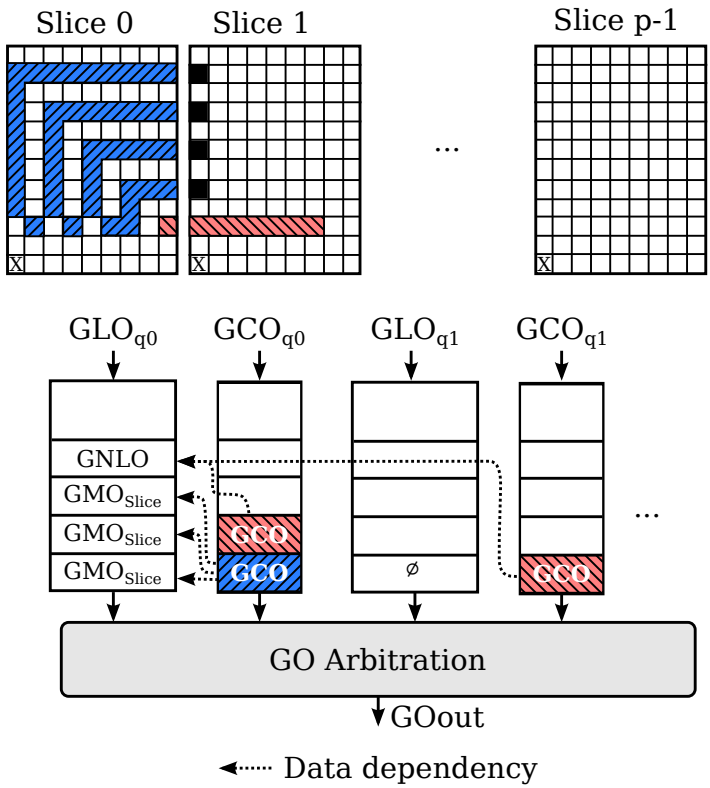



**Figure 4.14:** Example showing the deviation of the order the GOs are stored in queues to with regards to their data dependencies.

tag number from the queues. Pseudocode 14 (*read order mechanism*) presents a mechanism enabling an efficient realisation to establish the arbitration tag read order as hardware architecture. It exploits the property that GOs are en-queued in ascending order according to their arbitration tag and that these arbitration tags are continuous. The row number that a GO is issued in is used as its arbitration tag. $GO_{Sync}$ are GOs consisting of nothing but an arbitration tag. They are used to synchronise CIs for image rows without global patterns issued by SPIs or CIs. Therefore, the arbitration tags of the GOs in each of the queues are ascending and continuous, as the image is processed in raster scan order and $GO_{Sync}$ operations are issued.

Pseudocode 14 (*read order mechanism*) describes the ordering process of the GOs in the *GLOqs* and *GCOqs*. Its input vector *GOq* is the concatenation of the global operations at the head of *GLOqs* and *GCOqs* associated with the current CI. The GOs at *GOq* are ordered according to the data dependencies by a continuous comparison of their arbitration tags with the read barrier, *rdBarrier*. Initially,

**Pseudocode 14:** *Read order mechanism* — Mechanism for a hardware-efficient detection of the read order of the GO queues.

```
   Parameter: j = # children + # siblings
   Input: GOq(0 to 2 × j − 1)
   Output: GOout
 1 block := (False, ..., False)
 2 rdBarrier := 0
 3 while True do
 4     for i in 0 to 2 × j − 1 do
 5         if GOq_i.arbtag > rdBarrier then
 6             block_i := True
 7         else
 8             GOout := GOq_i.pop
 9     if block = (True, ..., True) then
10         rdBarrier++
11         block := (False, ..., False)
12         GOout :=(SYNC,rdBarrier)
```

*rdBarrier* is reset to 0. This allows reading all GOs of the queues with arbitration tag 0. If the arbitration tag of the GO at the head of a queue $i$ is greater than *rdBarrier* the read access to this queue is blocked by setting the $blocked_i$ flag. As soon as the read access to all queues is blocked, the *rdBarrier* is incremented by one and access to all queues is again granted by resetting the vector of *blocked* flags. Additionally, a $GO_{Sync}$ is issued to signal the parent CI that the *rdBarrier* is updated, which extends this mechanism to multiple CI levels. All GOs at the head of non-blocked queues are read in a round-robin manner and output to *GOout*. This results in a stream of GOs with ascending arbitration tags and complies with the data dependencies the GOs have due to the raster scan processing of the original input image. The merits of the method used in Pseudocode 14 (*read order mechanism*) with respect to a realisation in hardware are:

- An immediate decision without reading the heads of the queues is possible for outputting a global operation to *GOout*.
- The checks to block read access of queues are carried out simultaneously for all queues.

The hardware architecture for GO arbitration is discussed in Section 5.3.1.

Processing of Global Operations in a Multi-CI System

This paragraph describes the detailed realisation of a coalescing instance for a system which uses multiple CIs. The principles described in this sub-section lay the foundation of the *CI* also used for the special case of systems using only a single *CI*, as described in the following sub-section.

The realisation of a coalescing instance requires an efficient data structure to store the global label graph $F_G$. The tree structures of the global label graph $F_G$ are stored in the *global merger table (GMT)* which is realised as a 1-D array. An arc joining the vertex of global label $G_1$ with the vertex of global label $G_2$, $(G_1, G_2) \in E(F_G)$, is stored in the $GMT$ by updating $GMT[G_2.index]$ with $G_1$. The index of global labels is defined in Section 4.2. For storing feature vectors, the *global data table (GDT)* is introduced which is realised as a 1-D array index by the associated global label. All CIs have a common name space for global labels. Therefore, there can be arcs which join vertices of global labels which are associated with different CIs. To detect completed connected components and to recycle global labels, the number of associated child vertices in the local label graphs $F_{L,0}, \ldots, F_{L,p-1}$ and the global label graph $F_G$ is used, as explained in the following. This method requires the in-degree for each vertex of a global label to be stored.

**Definition 27.** *In-degree of a global label:* The in-degree of a vertex $v_G$ of a global label $G$ is the number of arcs in $E(F_G)$ and $LT$ pointing to $v_G$. Global labels whose vertices point to $v_G$ are in $E(F_G)$. Local labels whose vertices point to $v_G$ are in $LT$.

The 1-D array *INDEG* is the data structure storing the *in-degree* of a global label $G$ at *INDEG[g.index]*. A connected component is detected as *completed* if the in-degree of the global label is zero and the global label is a root vertex in $F_G$.

Each CI in a multi-CI system has its own *GDT*, *GMT* and *INCNT* data structure and is assigned a *lvlgrp* and a *slcgrp*. The instructions in Pseudocode 15 (*CI instructions*) are discussed in the following in a top-down approach. Each operation which requires more than one access to a data structure, or is iterative is expressed as a sub-routine. The instructions of Pseudocode 15 create the tree structures in the global label graph $F_G$ which associate each slice-component of the input image with its connected component. This is achieved by carrying out the global instructions issued by an SPI or child/sibling CI on the data structures *GMT*, *GDT* and *INDEG*, as explained in the following.

Each *GO* contains either one or two global labels (*lbl0* and *lbl1*), as introduced in Section 4.4.1. *GCOs* additionally contain a feature vector *fv*. If neither *lbl0* nor *lbl1* have the same *lvlgrp* as the receiving CI, the GO is forwarded to the parent CI. For processing a GCO, the GCO's global label *lbl0* and its feature vector *fv* is used. If the *GDT* contains no feature vector for global label *lbl0*, the feature vector *fv*, contained in a *GCO*, is copied to *GDT[lbl0.index]*. If the *GDT* already

**Pseudocode 15:** *CI instructions* — Instructions induced by *global operations (GOs)* carried out by the coalescing instance

```
1  processGO (GO)
       // Processing GCO
2      if GO.lbl0.lvlgrp ≠ CI_lvlgrp
3      ∧ GO.lbl1.lvlgrp ≠ CI_lvlgrp then
           // Forward GO to parent CI
4          toPrntCI(GO)
5      else if GO.type = GCO then
6          fv := GO.fv
7          label := GO.lbl0
8          GDT[l.index] := GDT[l.index] ∘ fv
9          decInDeg(label)
       // Processing GLOs
10     else if GO.type = GNLO then
11         l := GO.lbl0.index
12         GMT[l] := GO.lbl0
13         INDEG[l] := 2
14     else if GO.type = GO_Inc then
15         incInDeg(GO.lbl0)
16     else if GO.type = GO_Dec then
17         decInDeg(GO.lbl0)
18     else if GO.type = GMO_GL then
19         incInDeg(GO.lbl0)
20     else if GO.type = GMO_GG then
21         globalJoin(GO.lbl0, GO.lbl1)
22     else if GO.type = GMO_Slice then
23         globalJoin(GO.lbl0, GO.lbl1)
24         decInDeg(max(GO.lbl0, GO.lbl1))
```

contains a feature vector for global label *lbl0*, feature vector *fv* is combined with and stored to *GDT[lbl0.index]*. The ∘-operator is used, as defined in Section 2.3.3, as a combination operator dependent on the extracted feature vectors. As the feature vector of a GCO is associated with a slice-component, the in-degree of the associated global label is decremented by one when a GCO is processed by a CI. Decrementing *INDEG* is expressed as a sub-routine *decInDeg* which handles the special case when the in-degree becomes zero. Further details regarding *decInDeg* (Pseudocode 18) are discussed later in this section.

A global new label operation (*GNLO*) makes global label *lbl0* a root by updating the $GMT[lbl0.index]$ with global label *lbl*0. Since every new global label is associated with two slice-components (see Section 4.3), $INDEG[lbl0.index]$ is initialised with two. CIs issue $\mathrm{GO}_{Inc}$ operations to increment the in-degree of global labels associated with their parent or sibling CI by calling sub-routine *incInDeg* (Pseudocode 17). CIs issue $\mathrm{GO}_{Dec}$ operations to decrement the in-degree of global labels associated with their parent or sibling CI by calling sub-routine *decInDeg* (Pseudocode 18). A $GMO_{GL}$ joins another slice-component with the associated global label *lbl0*. This is required to increment the in-degree by one, handled by sub-routine *incInDeg* (Pseudocode 17). $GMO_{GG}$ and $GMO_{Slice}$ join the sub-root vertices of their associated global labels (*lbl0* and *lbl1* used in the CI instructions in Pseudocode 15). Sub-routine *globalJoin* realises the *union* (as in *HD-UF* shown in Algorithm 6) operation distinguishing the *slcgrp* and *lvlgrp* of the global labels. A $GMO_{Slice}$ is detected when two global labels are joined which are both associated with a common vertex in local label graph $F_{L,i}$. Since SPIs always associate vertices in $F_{L,i}$ with the smaller global label via *LT* (see Pseudocode 10, line 4), the in-degree of the larger global label is decremented by one.

Each CI issues a global label to the associated SPI upon request. A CI is associated with the SPIs processing image slices whose borders have the same level as the CI. Global labels are generated and recycled by the *global label management (GLM) operations* (Pseudocode 16) within a CI. The sub-routine *generateNewGL* (Pseudocode 16) combines the *slcgrp* of the CI, the *lvlgrp* of the CI and the index at the head of its *global reuse queue (GR)* to a global label, which is passed on to an SPI. The *GR* is the queue which contains one index for each global label of its CI. The *global reuse queue (GR)* is initialised with one index for each global label of its CI. Global labels of completed slice-components are recycled to *GR* by calling sub-routine *recycleGL* (Pseudocode 16) from *decInDeg* (Pseudocode 18, line 16).

**Pseudocode 16:** *GLM operations* — Operations to generate and recycle global labels (*GL*).

```
1 generateNewGL()
2     G.slcgrp := CI.slcrp
3     G.lvlgrp := CI.lvlgrp
4     G.index := GR.pop()
5     return(G)
6 recycleGL(GL G)
7     GR.push(G)
```

Sub-routine *incInDeg*, presented in Pseudocode 17, increments the in-degree of a global label by one. If the global label *lbl* has the same *lvlgrp* as the CI instance calling *incInDeg*, the in-degree is increased by incrementing *INDEG[lbl.index]* by one. If the *lvlgrp* of *lbl* is smaller, a $GMO_{Inc}$ is issued to the parent CI.

**Pseudocode 17:** *In-degree increment* — Sub-routine for incrementing the in-degree

```
1 incInDeg (lbl)
2     if GMT[lbl.index].lvlgrp < CI_lvlgrp then
3         toPrntCI(GO_inc(lbl));
4     else
5         INDEG[lbl.index]++;
```

The sub-routine *decInDeg* shown in Pseudocode 18 decrements the in-degree of global label *lbl* by one. If the in-degree of a global label is zero, the slice-component or connected component that *lbl* is associated with, is detected as completed. The memory in *GMT*, *GDT* and *INDEG* is, therefore, not required anymore and reused by recycling *lbl* to the *GR*. If *lbl* is the global label of a root vertex, no further processing for the extraction of its feature vector is required. It is, therefore, output (line 7 in Pseudocode 18) and its *GDT* entry is cleared. If *lbl* is not the global label of a root vertex, its feature vector is combined with the feature vector of its parent and its *GDT* entry is cleared. For the case that *lbl* is a sub-root, it is combined with its parent by issuing a *GCO* to the parent CI containing *lbl*'s feature vector, *GDT*[*lbl.index*], and the parent label of its global label. If *lbl* is not a sub-root it is combined with its parent's feature vector by the current CI.

The sub-routine *globalJoin* presented in Pseudocode 19 (*global join*) joins the sub-root vertices of two global labels, $l_0$ and $l_1$, by updating the data structures *GDT*, *GMT* and *INDEG*. It extends the *union* operation from *HD-UF* (Algorithm 6) by providing the specific instructions on the data structure associated with its CI. The recursion (as in cases (c), (d) and (f)) of *union* in *HD-UF* (Algorithm 6) are realised by issuing GOs to sibling or parent CIs. In the following, the different operations of *HD-UF* (Algorithm 6) to process cases (a) to (g) are discussed using the terminology introduced in Section 4.2. For *case (a)* and *(b)*, both global labels $l_0$ and $l_1$ have the same *lvlgrp*. The global labels of the associated sub-roots $sr_{min}$ and $sr_{max}$ are joined by updating GMT[$\mathrm{sr}_{max}$] with $sr_{min}$ and by incrementing the in-degree of $sr_{min}$ by one. To process case *(e)* the same operations as in *(a)* and *(b)* are used, however the *incInDeg* subroutine issues a $\mathrm{GMO}_{Inc}$ to the parent CI which is required, because $sr_{min}$ and $sr_{max}$ are in different *lvlgrps*. For processing case *(c)*, the parent vertices of the sub-roots of $l_0$ and $l_1$, GMT[$\mathrm{sr}_{min}$] and GMT[$\mathrm{sr}_{max}$] respectively, are joined by issuing a $GMO_{GG}$ to the parent CI. This corresponds to the recursive call of *union* in *HD-UF* (Algorithm 6). The sub-root vertices of $l_0$ and $l_1$ are joined by updating GMT[$\mathrm{sr}_{max}$] with $\mathrm{sr}_{min}$ and incrementing the in-degree of $sr_{min}$ by one. It is also requires to reduce the in-degree of GMT[$\mathrm{sr}_{max}$] by one, which is realised by issuing a $\mathrm{GMO}_{Dec}$ to the parent CI. For processing case *(d)*, the parent vertices of the sub-roots of $l_0$ and $l_1$, GMT[$\mathrm{sr}_{min}$] and GMT[$\mathrm{sr}_{max}$] respectively, are joined by issuing a $GMO_{GG}$ to the parent CI. In cases *(f)* and *(g)*, $sr_{min}$ and $sr_{max}$ have

**Pseudocode 18:** *In-degree decrement* — Sub-routine for decrementing the in-degree *INDEG* iteratively.

```
1  decInDeg (lbl)
2      INDEG[l] -- // Decrementing in-degree
3      if INDEG[lbl.index] = 0 then
4          l := lbl.index
5          p := GMT[l.index]
6          if p = l then
               // Completed connected component is detected
7              output(GDT[l])
8              GDT[l] := ∅
           // Combination with parent FV
9          else if p.lvlgrp < CI_lvlgrp then
               // lbl is sub-root
10             toPrntCI(GCO(p, GDT[l]))
11             GDT[l] := ∅
12         else
13             decInDeg(p)
14             GDT[p.index] := GDT[p.index] ∘ GDT[l]
15             GDT[l] := ∅
           // Recycling of lbl by using Pseudocode 16
16         recycleGL(lbl)
```

the same *lvlgrp* but different *slcgrps*, i.e. are associated with sibling CIs. In case *(f)* the vertex associated with $sr_{min}$ does not have a parent vertex associated with a lower *lvlgrp*. As the parent vertex associated with global label $sr_{max}$ is stored in the GMT of the sibling CI, it is not possible to determine whether $sr_{max}$ is associated with a global label which has a lower *lvlgrp*. Therefore, a new global label *newGL* is requested from the parent CI and associated with $sr_{min}$ by updating GMT[$sr_{min}$] with *newGL*. To associate $sr_{max}$ with *newGL*, a $GMO_{GG}$ is issued to the sibling CI. In case *(g)*, the parent vertex of $sr_{min}$ is associated with a global label of a lower *lvlgrp*: $psr_{min}$. To associate $sr_{max}$ with $psr_{min}$, a $GMO_{GG}$ is issued to the sibling CI.

**Pseudocode 19:** *Global join* — Global join operation when multiple CIs are used

```
1  globalJoin(l_0, l_1)
2      sr_0 := find(l_0,GMT); psr_0 := GMT[sr_0]
3      sr_1 := find(l_1,GMT); psr_1 := GMT[sr_1]
4      if sr_0 ≠ sr_1 then
5          if psr_0.lvlgrp≤psr_1.lvlgrp then
6              sr_min := sr_0; sr_max := sr_1
7          else sr_min := sr_1; sr_max := sr_0
           // Lvlgrp of both sr = CI_lvlgrp
8          if sr_0.lvlgrp = sr_1.lvlgrp = CI_lvlgrp then
9              if sr_min.slcgrp ≠ sr_max.slcgrp then
10                 if psr_min.lvlgrp = sr_max.lvlgrp then
                       // Case (f)
11                     newGL := toPrntCI(GNLO)
12                     GMT[sr_0] := newGL
13                     toNghCI(GMO_GG(sr_1,newGL))
14                 else // Case (g)
15                     toNghCI(GMO_GG(sr_1,psr_1))
16             else if (psr_0.lvlgrp ≠ sr_0.lvlgrp) ∧ (psr_1.lvlgrp ≠ sr_1.lvlgrp) then
                   // Case (c)
17                 toPrntCI(GMO_GG(GMT[sr_min],GMT[sr_max]))
18                 toPrntCI(GMO_Dec(GMT[sr_max]))
19                 GMT[sr_max] := sr_min
20                 incInDeg(sr_min)
21             else // Cases (a)&(b)
22                 GMT[sr_max] := sr_min
23                 incInDeg(sr_min)
24         else
25             if psr_max = sr_min then
                   // Case (e)
26                 GMT[sr_max] := sr_min
27                 incInDeg(sr_min)
28             else // Case (d)
29                 toPrntCI(GMO_GG(GMT[sr_min],GMT[sr_max]))
```

Processing of Global Operations in a Single CI System

In the special case that a single coalescing instance (CI) processes all the global operations issued by the SPIs, all global labels have the same *slcgrp* as the CI, and the same *lvlgrp* as the CI. This reduces the operation *globalJoin* to the case shown in Figure 4.4*(a)*. The use of *globalJoinSingle* shown in Pseudocode 20 instead of *globalJoin* is, therefore, sufficient.

A single CI carries global operations one after the other, due to their sequential data dependencies (Section 4.4.2). As the maximum number of global operations increases with the number of SPIs, the maximum performance of a single CI limits throughput of a parallel SLCCA instance, as shown for an FPGA hardware architecture in [64].

On the one hand, a single joining operation of two global labels requires fewer instructions when using a single CI, as shown in Pseudocode 20 (*single CI global join*). On the other hand, a single CI becomes a processing bottle-neck for a large number of SPIs, hence limiting the scalability with the number of SPIs. The question of whether a single CI or multiple CIs are more efficient is implementation-specific and discussed in Section 5.5 by comparing the resources required for hardware architectures dependent on the image size and processing throughput.

**Pseudocode 20:** *Single CI global join*

```
1 globalJoinSingle(l_0, l_1)
2     sr_0 := find(l_0,GMT)
3     sr_1 := find(l_1,GMT)
4     if sr_0 ≠ sr_1 then
5         sr_min := min(sr_0,sr_1)
6         sr_max := max(sr_0,sr_1)
7         GMT[sr_max] := sr_min
8         incInDeg(sr_min)
```

## 4.5 Summary and Contributions of the PCCA Algorithm to the State of the Art

The parallel *SLCCA* algorithm, *PCCA*, which was introduced and described in detail in this chapter is an improvement to the state of the art on several levels, as pointed out in the following.

- **Improved scalability by parallel feature vector combination:** In the parallel CCA architecture in [71] (the predecessor of *PCCA*), the limitation in terms of scalability results from a single coalescing instance which carries out union-find operations and combines feature vectors. The *PCCA* algorithm eliminates this bottleneck by introducing multiple communicating coalescing instances.
- **Distributed union-find algorithm:** A hierarchical distributed union-find (*HD-UF*) algorithm is introduced to process multiple global operations in parallel. Only the usage of *HD-UF* allows the aforementioned improvement of distributing operations from one to multiple coalescing instances. Chapter 5 shows that the throughput of the *PCCA* hardware architecture increases with the number of image slices processed in parallel. The hierarchical distributed union-find algorithm, therefore, contributes significantly to the increased throughput achieved by the *PCCA* hardware architecture presented in Chapter 5.
- **On-the-fly processing of the binary pixel stream:** The pixel data of the binary image stream is processed on the fly as it is received. A frame buffer to store an entire input image is, therefore, not required. This is especially important to realise a resource-efficient *PCCA* hardware architecture, as storing large images (e.g. *UHD8k*) in on-chip memory requires a significant amount of modern processing devices.

These properties of the *PCCA* algorithm enable the use of spatial parallelism and temporal parallelism on modern processing devices, such as FPGAs and multi-core GPPs, to accelerate processing. The improvements at the algorithmic level of *PCCA* are a necessity for the design of the efficient *PCCA* hardware architecture presented in the following chapter which achieves approximately double the throughput as the architecture in [71].

# 5 Hardware Architecture of Parallel SLCCA

This chapter describes the design of the dedicated hardware architecture carrying out the parallel *Single-Lookup Connected Components Analysis (PCCA)*. The architecture proposed in this chapter is customised for (but not limited to) a realisation as a hardware architecture on an FPGA. The hardware sub-units of the PCCA hardware architecture are discussed in Sections 5.1 to 5.3 in detail, followed by a discussion of the resource sharing of sub-components of the PCCA architecture in Section 5.4 and the experimental results and required hardware resources in Section 5.5.

The hardware architecture of a *slice processing instance* (SPI) is denoted as a *slice processing unit (SPU)*. The hardware architecture of a *coalescing instance (CI)* is denoted as a *coalescing unit (CU)* in the following. Figure 5.1 shows the block diagram of the *PCCA* hardware architecture with $p$ SPUs and $q$ levels of CUs arranged in a tree. The following sections evolve each component from the algorithmic description, as given in Chapter 4, to an architecture at register-transfer level.

## 5.1 Image Distribution Unit

The *image distribution unit (IDU)* separates the incoming pixel stream into $p$ pixel streams in order to generate $p$ vertical image slices. This effectively leads to the separation of $G_P$ into the sub-graphs $G_{P,0}, \ldots, G_{P,p-1}$ described in Section 4.2.

Figure 5.2 shows the architecture of the *IDU* at register-transfer level. At the input interface $W_{IDU}$ binary pixels of the input stream are received in parallel, which are consecutive in raster scan order. For the implementation, $W_{IDU} = 64$ was chosen, as shown in Figure 5.2. This value, however, can vary and depends on the bandwidth of the input stream. As the incoming pixel data are received in raster scan order, they are buffered by $p$ first-in-first-out (FIFO) memories called IDU-FIFOs. There is one IDU-FIFO to buffer the pixel data received for each image slice. The IDU-FIFO associated with image slice $i$ is referred to as *IDU-FIFO*$_i$. The $W$ pixels received for each image row are distributed to the IDU-FIFOs by writing pixel $0, \ldots, \lceil W/p \rceil - 1$ of each row to IDU-FIFO$_0$, followed by writing pixel $\lceil W/p \rceil, \ldots, 2 \times \lceil W/p \rceil - 1$ to IDU-FIFO$_1$, and so on. Pixel data of image row $i + 1$ are written to the IDU-FIFOs

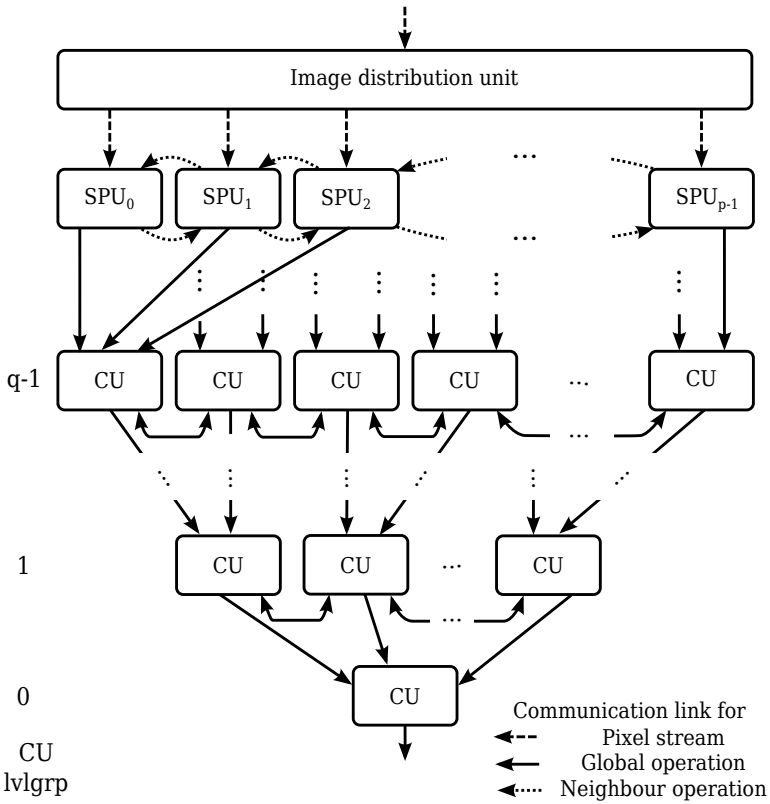



**Figure 5.1:** This figure shows the block diagram of the *PCCA* hardware architecture consisting of one *image distribution unit (IDU)*, $p$ *slice processing units (SPUs)* and $q$ levels of *coalescing units (CUs) arranged in a tree*. The arrows show the communication links.

while the pixel data of image row $i$ is read. Each IDU-FIFO, therefore, requires the capacity to store the pixel data for two image rows of its associated image slice, i.e. $2 \times \lceil \frac{W}{p} \rceil$ pixels. The pixel data at the outputs of the IDU-FIFOs are applied pixel by pixel to the output of the *IDU* after the first image row is received. By reading one pixel in each clock cycle it is ensured that the $p$ slice streams created are synchronous, i.e. the pixels in the i$^{th}$ columns of each image slice are output simultaneously.

The inputs and outputs of IDU-FIFOs are operated with independent and different frequencies: $f_{pix}$ at the input and $f_{CCA}$ at the output. To ensure that none of the IDU-FIFOs overflows, the frequency ratio of $f_{pix}$ and $f_{CCA}$ must be

$$\frac{f_{CCA}}{f_{pix}} \geq \frac{W_{IDU}}{p} \times \frac{6}{5}. \tag{5.1}$$

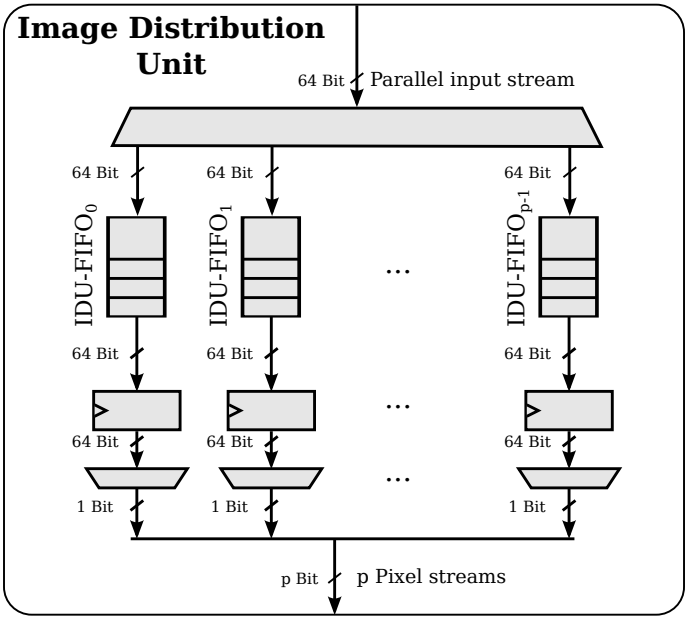



**Figure 5.2:** Data-path of architecture of the *image distribution unit* at register-transfer level for $W_{IDU} = 64$.

## 5.2 Slice Processing Unit

Each of the pixel streams generated by the *IDU* is processed by a *slice processing unit (SPU)*, which extracts the feature vectors of slice components and connected components within its image slice. An *SPU* which processes the pixel stream of image slice $i$ associates all vertices of a connected component in $G_{P,i}$ with the same tree structure in the label graph $F_{L,i}$ (see blue arcs in Figure 4.2) by carrying out union-find instructions on the label graph $F_{L,i}$ and extracting the feature vectors of the associated connected components.

In Figure 5.3, the block diagram of architecture of the *SPU* is shown as extending the architecture from Figure 3.1 in Section 3.1 by the ability to communicate to other SPUs (processing a neighbour slice) and being able to detect and issue global operations. In the following, only the units of the SPU added to or altered by the architecture from Section 3.1 which processes an entire image, are discussed. The black arrows in Figure 5.3 are the interfaces and connections added due to multi-slice processing. The grey arrows are connections that already exist in Figure 3.1. The widths of control signals in Figure 5.3 and the following figures of this section are given in *bits*. Data signals are illustrated as abstract types, such as *GO* (global operations, introduced in Section 4.3), as their widths depend on the image size.

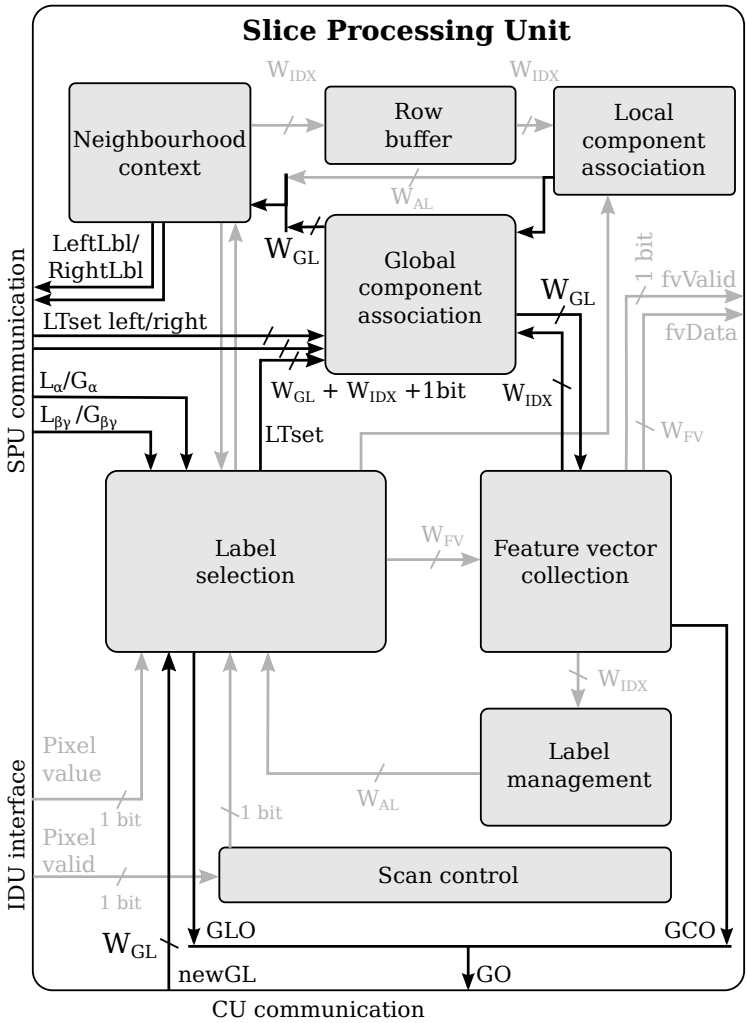



**Figure 5.3:** This figure shows the architecture of the *slice processing unit* with the capability of detecting and executing *global operations*. The black arrows indicate the internal and external connections added to the architecture from Figure 3.1 to enable multi-slice processing.

### 5.2.1 Local and Global Component Association Units

The *local component association* unit maintains the tree structures of the label graph $F_{L,i}$ for its associated image slice with the identifier $i$ in a *local merger table* $M_i$ which is realised as an 1-D array indexed by the local label. The parent of a local label $v$ is stored in $M_i[v]$. Since there are no arcs between local labels of different slices, each SPU maintains its own name space for local labels starting at 1.

The *global component association* unit joins vertices from $F_{L,i}$ with their parent vertices in the global label graph $F_G$ (introduced in Figure 4.2). These connections are established by the *link table LT*, which is realised as a 1-D array associating a local label $l$ with its global label $g$ by updating $LT$ with $LT[l] \coloneqq g$. The link table $LT$ is read for each local label in the labelled image $L$ to determine the global label associated with each local label from $L$. When a slice component is completed, the global label of its associated feature vector is also determined by a lookup in the link table $LT$. The link table $LT$ is adapted when a *GLO* is detected by the *label selection* unit of the current SPU or a *GLO* is detected in one of the neighbour SPUs, e.g. the write accesses to $LT_{i-1}$ or $LT_{i+1}$ in Algorithm 9 or 11. As part of the PCCA hardware architecture, the link table $LT_i$ in the *global component association* unit receives a local label at its input in every clock cycle. The corresponding global label is output in the following clock cycle. The $LT$ is realised as true-dual port [133] block-RAM (BRAM). One port is used to lookup the global label of the continuous stream of local labels at its input received by the *local component association* unit. The second port of the $LT$ is shared between the *label selection* unit, the neighbour SPUs and the *feature vector collection* unit. It is used as a write port when *GLOs* are detected and as a read port when *GCOs* are detected.

### 5.2.2 Feature Vector Collection Unit

Local feature vectors define the properties of slice components or connected components relative to the coordinates within the processed image slice. Global feature vectors define the properties of slice components or connected components relative to the coordinates of the entire input image. The *feature vector collection (FVC)* unit outputs the global feature vector for each connected component or slice component in its processed image slice. To save memory resources, however, the *FVC* unit stores and updates local feature vectors associated with each local label $l$ in $L$ (labelled image) in the *data table (DT)* at $DT[l.index]$ , as explained in more detail in Section 3.1.3. Local feature vectors are transferred to global feature vectors when the associated connected component of a slice component is detected as completed.

The *FVC* unit uses the index of a completed feature vector to check whether the feature vector is associated with a global label. If the completed feature vector is associated with a global label it belongs to a slice component. Therefore, a *GCO* containing the global feature vector and the global label is issued and passed on to

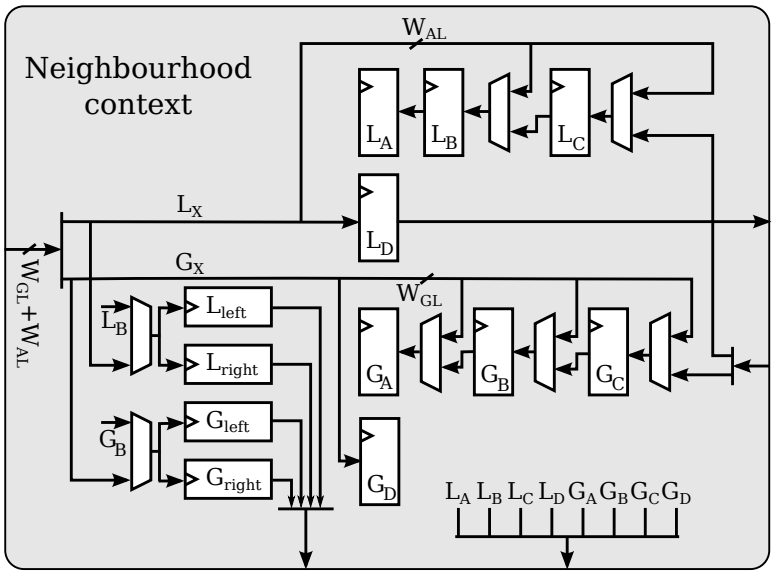



**Figure 5.4:** Hardware architecture of *neighbourhood context* at the register-transfer level.

the *CU*. If a completed feature vector in $DT$ is not associated with a global label, its global feature vector is output to *fvData*. The parts of the local feature vectors related to image positions are updated after being read out of $DT$. The exact way to transform local to global feature vectors depends on the extracted features. For bounding boxes, a global feature vector $FV_G$ is derived from a local feature vector $FV_L$ by adding the offset of the processed image slice. This offset is calculated within the SPU as the product of the slice identifier $slice_{id}$ and the slice width $W_s$:

$$FV_G := \begin{pmatrix} FV_L.x_{min} \\ FV_L.y_{min} \\ FV_L.x_{max} \\ FV_L.y_{max} \end{pmatrix} + \begin{pmatrix} slice_{id} \times W_S \\ 0 \\ slice_{id} \times W_S \\ 0 \end{pmatrix}. \tag{5.2}$$

SPUs only process image slices of level $q-1$, i.e. $slice_{id}$ are the slice identifiers of level $q-1$.

### 5.2.3 Neighbourhood Context Unit

The *neighbourhood context* unit provides the local and global label in the pixel neighbourhood $\eta_L$ to the *label selection* unit. Figure 5.4 shows its hardware architecture at the register-transfer level. Parallel access to local and global labels in the

current pixel's neighbourhood is achieved by storing $L_A, \ldots, L_D$ and $G_A, \ldots, G_D$ in registers. The local and global labels of $L_C$ and $G_C$ are received by the *local component association* and *global component association* units shown in Figure 5.3. The registers are connected to provide the neighbourhood labels for position $(x+1, y)$ in the next cycle by assigning the labels

$$\begin{aligned} L_A &:= L_B^-, \\ L_B &:= L_C^-, \\ L_C &:= M[L[x+2][y-1]], \\ L_D &:= L_X^-, \\ G_A &:= G_B^-, \\ G_B &:= G_C^-, \\ G_C &:= LT[L[x+2][y-1]], \\ G_D &:= G_X^-, \end{aligned} \tag{5.3}$$

where the labels with superscript $-$ are the labels in the neighbourhood of the previous pixel. To keep the labels in the neighbourhood consistent when carrying out merger operations, multiplexers are added to assign the local/global label assigned to the current pixel to a register. When a propagating (local) merger pattern is detected, $L_B$ is updated with $L_X$ and $G_B$ is updated with $G_X$, as defined in Section 2.20. If a local merger pattern was detected for the previous pixel and the current pixel is an object pixel whose $L_C$ is an object pixel ($I[x+2, y-1] = 1$), then $L_C$ is updated with $L_X$ instead of the output of the *local label association* units. The same holds true for the global label $G_C$: if a local merger pattern was detected in the previous pixel and the current pixel is an object pixel whose $L_C$ is an object pixel ($I[x+2, y-1] = 1$), then $G_C$ is updated with $G_X$ instead of the output of the *global label association* units. If the link table $LT$ is updated to join local label $l$ with a global label $g$, all global labels where the local label is equal to $l$, are updated with $g$. Since updates of $LT$ are initiated by the current or a neighbour *SPU*, multiplexers are required for $G_A$, $G_B$ and $G_C$.

The local and global label at the left and right border of the current image row are provided to the neighbour *SPUs* by storing the value of $L_X/G_X$ in the registers $L_{left}/G_{left}$ and $L_{right}/G_{right}$ in the first/last column of each row. If the left-most/right-most pixel of the current row is not an object pixel, the label as-

signed to the pixel above is used. In this way diagonal connection are considered, as well.

$$\begin{aligned} L_{left} &:= \begin{cases} L[0,y], & I[0,y]=1 \\ L[0,y-1], & otherwise, \end{cases} \\ G_{left} &:= LT[L_{left}], \\ L_{right} &:= \begin{cases} L[W-1,y], & I[W-1,y]=1 \\ L[W-1,y-1], & otherwise, \end{cases} \\ G_{right} &:= LT[L_{right}]. \end{aligned} \tag{5.4}$$

The registers $L_{left}$ and $G_{left}$ are provided to the left neighbour SPU and used as $L_{\beta\gamma}$ and $G_{\beta\gamma}$ to select the label of border pixels and to detect global merger patterns. The registers $L_{right}$ and $G_{right}$ are provided to the right neighbour SPU and used as $L_{\alpha}$ and $G_{\alpha}$ to select the label of border pixels and to detect global merger patterns.

### 5.2.4 Label Selection Unit

The *label selection* unit uses the registers in the *neighbourhood context* unit of the current SPU and the two neighbour SPUs to select $L_X$ and $G_X$, as described in Section 4.4.1. It issues global label operations (*GLOs*) and acquires a new global label (*newGL*) via a direct connection to the associated CU.

## 5.3 Coalescing Unit

The *coalescing unit (CU)* receives GOs from its child unit or sibling unit to combine the feature vectors of slice components with their connected component. A child unit is either an SPU or a CU, a sibling unit is another CU with the same *lvlgrp* which is required to combine feature vectors of image slices of the same level (see page 121 for the definition of *lvlgrp*). Distributing GOs to multiple CUs allows parallel processing. The GOs associated with the same CU, however, need to be processed sequentially. Figure 5.5 shows a block diagram of the CU architecture. GOs received by child units or sibling units are buffered in queues: GLOs in *GLO queues* (GLOQ) and GCOs in GCO queues (GCOQ). For each of the $k$ connections to a child unit or sibling unit, one *GLOQ* and one *GCOQ* is required. The *GO arbitration* is the sub-unit of the CU which evaluates the arbitration tags of the GOs at the head of the *GLOQs* and *GCOQs* to comply with the data dependencies from the input image. The *global label management (GLM)* unit issues global labels (GLABs) to child SPUs or child CUs and stores global labels recycled by the *GO processor*. A major task of the CU is to carry out *find* operations on the *GMT* (Global Merger Table). Since every lookup of a *find* operation is dependent on the preceding lookup operation, the latency of the *GMT* determines the throughput of the CU. Therefore, the *GMT* is realised as on-chip block-RAM (BRAM), since the result of a read operation is available in the next clock cycle [133]. These data dependencies of find operations make the use of instruction level parallelism techniques such as pipelining [52] inefficient or impossible. This is a main motivation for the design of the *GO processor* which is the main processing unit of the CU discussed in 5.3.2. The *GDT* and *INDEG* (introduced in Section 12) are mapped to BRAMs, too, as their read latency is also crucial to the throughput of a CU. Storing the data of *GMT*, *GDT* or *INDEG* in off-chip memory would require a connection via a (single) memory controller. As there are several instances of the CU on the same (FPGA) chip, the memory controller's bandwidth would need to be shared among the CUs. When implementing the data structures as on-chip BRAMs, the required bandwidth is distributed to several memories each of which has a guaranteed bandwidth and latency. The sum of the bandwidth of the used on-chip BRAMs combined is significantly higher than the bandwidth of an external memory controller [129, 133].

### 5.3.1 Arbitration of Global Operations

The architecture shown in Figure 5.6 carries out Algorithm 14 to sort the GOs in the GLO queues and GCO queues to be consistent with the data dependencies of the original input image. The input vector *GOQs* is a concatenation of the outputs of $k$ queues. These are either GLO or GCO queues from child SPUs, child CUs or from sibling CUs. A multiplexer controlled by the one-hot encoded bit-vector

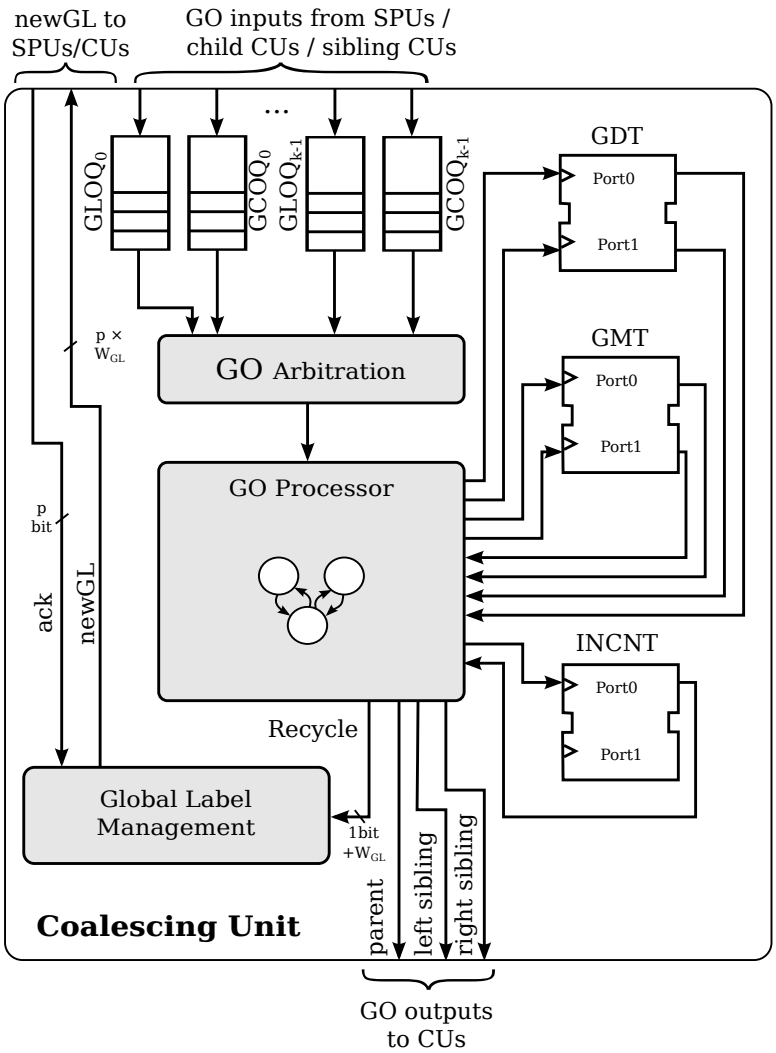



**Figure 5.5:** Block diagram of the architecture of the *coalescing unit* showing all sub-units and their connections.

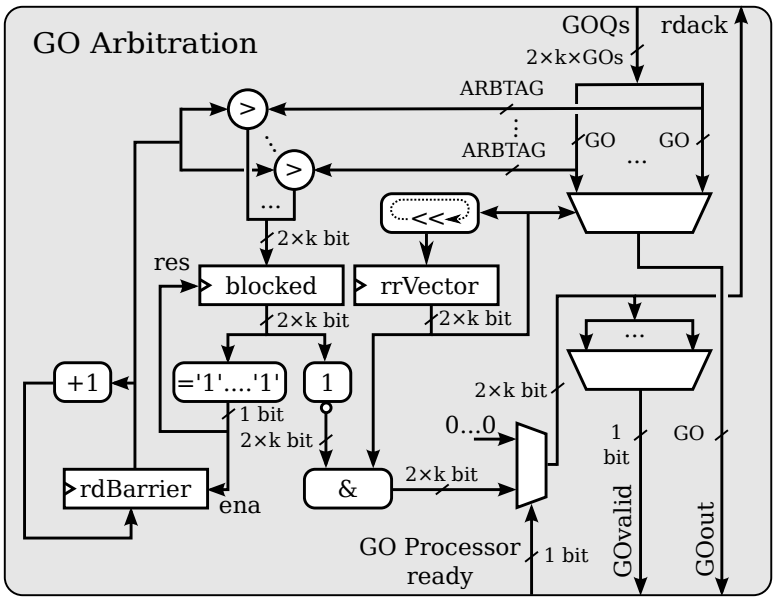



**Figure 5.6:** Architecture of the *global operations (GO) arbitration* at the register-transfer level to sort the GOs of $k$ queues connected to the input.

stored in the circular shift register *rrVector* forwards the GOs at input *GOQs* to the output *GOout*, one at a time. The circular $2 \times k$-bit shift-register *rrVector* is initialised with *0x1*. To forward the data at the input *GOQs* sequentially, the one-hot encoded value in *rrVector* is continuously shifted and serves as an access token to the output *GOout*. This effectively creates a round-robin arbiter. *GOvalid* serves as the valid-flag for output *GOout*. The output vector *rdack* are the signals to request the subsequent GOs in each queue attached to the input *GOQs*. To sort GOs as described in Algorithm 14, the generation of *GOvalid* and *rdack* is of major importance and is discussed in the following. At first, the arbitration tag (ARBTAG) of all $2 \times k$ GOs at the input are compared to the value of the read barrier stored in register *rdBarrier*. The ARBTAGs are assigned the row numbers, a GO is detected in, as explained in Section 4.4.1. If the arbitration tag of the GO is larger than *rdBarrier*, the access to its queue is blocked by asserting the corresponding flag in the *blocked* register to *'1'*. If all the flags stored in register *blocked* are *'1'*, all arbitration tags at *GOQs* are larger than *rdBarrier*, therefore the value in register *rdBarrier* is incremented by one and all the flags in register *blocked* are reset to *'0'*. This mechanism ensures that all GOs with arbitration tag $i$ are processed before one GO with arbitration tag $i + 1$ is processed.

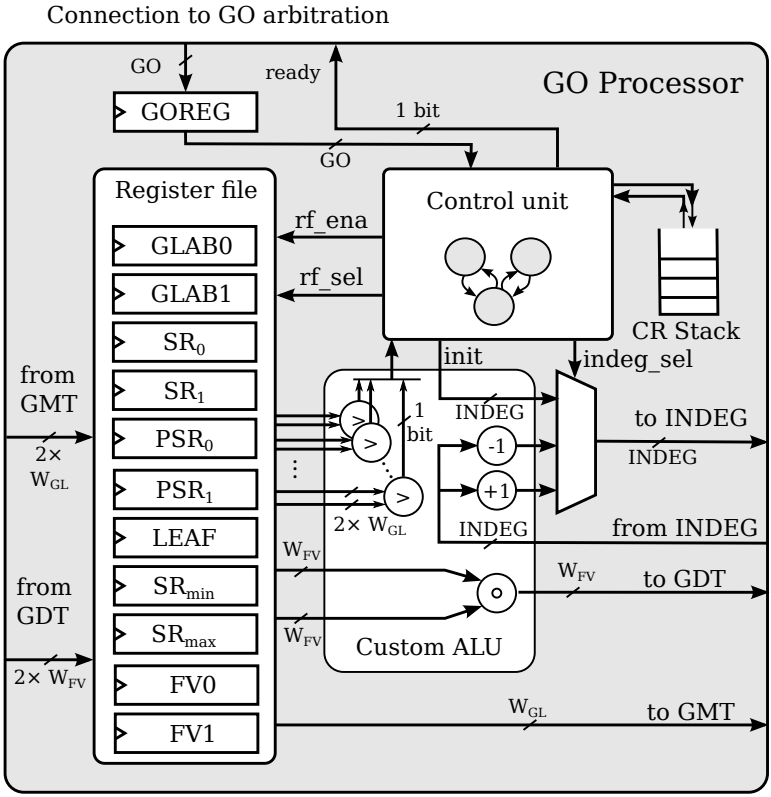



**Figure 5.7:** Architecture of the *global operations (GO) processor* at the register-transfer level.

### 5.3.2 GO Processor Unit

The *global operation (GO) processor* unit is a light-weight application-specific processor employed to carry out *global operations (GOs)*. Its architecture is based on the *von Neumann architecture* [95] reduced to only the components and instructions necessary for processing GOs. Figure 5.7 shows the architecture at the register-transfer level. The components introduced in the following are related to *von Neumann's* terminology [95]. The *GOREG* register is the equivalent of the *instruction register (IR)* which is used to store a global operation while it is being processed. As the *GO processor* processes the global operations stored in the *GLOQs* and *GCOQs* sequentially, the equivalent of a program counter *(PC)* is not necessary. The control unit decodes the GO in *GOREG* and carries out micro-programs based on Pseudocode 15 to Pseudocode 19 which are realised as hard-wired finite state machines (FSM). The vector of the current state of this FSM is, therefore, the equivalent of a micro-program counter *(MPC)*. When entering a sub-routine, such as *incInDeg* (Algorithm 17), the state to return to is pushed onto the call-return stack *(CR stack)*.

Since the maximum number of nested sub-routine calls for processing GOs is four, the *(CR stack)* has a depth of four, as well. A general algorithmic logic unit (ALU) is reduced to a *parallel custom ALU* carrying out four operations: Addition by one, subtraction by one, comparison of global labels and feature vector (FV) combination. Addition and subtraction are required to increase and decrease *INDEG* by one. The comparison operations are used to determine child vertex and parent vertex when two global labels are joined (see Pseudocode 19). The operations to combine feature vectors is dependent on the features to extract, and is realised as a dedicated combinational logic block. It is marked by "$\circ$" in the *parallel custom ALU* in Figure 5.7. To extract the area of connected components, the $\circ$-operator is implemented as an addition and to extract the bounding box the $\circ$-operator is implemented as a combination of minimum and maximum operation, as described in 3.1.3. All of the operations in the *parallel custom ALU* are carried out simultaneously in parallel.

The *register file* contains two sets of registers: one for global labels and one for feature vectors. The registers *GLAB0, ..., SR*$_{max}$ are used to store global labels when carrying out *find* operations on *GMT*. All of them can be used to address the *GMT* or the *GDT* or be used as input data to the *GMT*. The registers *FV0* and *FV1* store feature vectors. Both can be either filled with the output of *GDT* or with the feature vector of a combination operation.

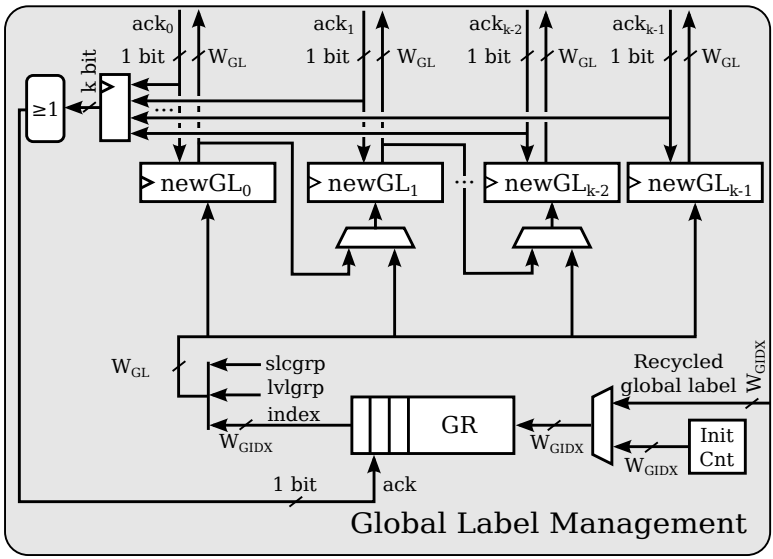



**Figure 5.8:** Architecture of the *global label management (GLM)* unit at the register-transfer level.

### 5.3.3 Global Label Management Unit

The *global label management (GLM)* unit provides global labels to its child SPUs and child CUs. Figure 5.8 shows its architecture at the register-transfer level. The global labels associated with a CU consist of the CU's *lvlgrp*, the CU's *slcgrp* and the *index* to address one entry in each of the CU's data structures. The *global reused queue* *GR* contains the indices of all unused global labels, which corresponds to all unused entries of the CU's data structures. Initially, *GR* is filled with continuous indices by the counter *init cnt*. As child units and sibling units may request global labels simultaneously, global label are provided via the registers $newGL_0$ to $newGL_{k-1}$. There is one *newGL* register for each of the $k$ child and sibling units. When child or sibling unit $i$ reads from $newGL_i$, it acknowledges this by asserting a *'1'* to $ack_i$, which is the signal to overwrite $newGL_i$ with the global label at the head of the *GR* queue. Simultaneous acknowledgements are buffered, as the refilling of $newGL_i$ registers is carried out sequentially. This sequential refilling is possible because SPUs can at most request a new global label in the first and last column of their image slice. Since processing one pixel in the SPU corresponds to one clock cycle, the *GLM* as described here requires the width $W_S$ of an image slice to be greater than or equal to the number of sibling and child units $k$ of the CU, $W_S \geq k$.

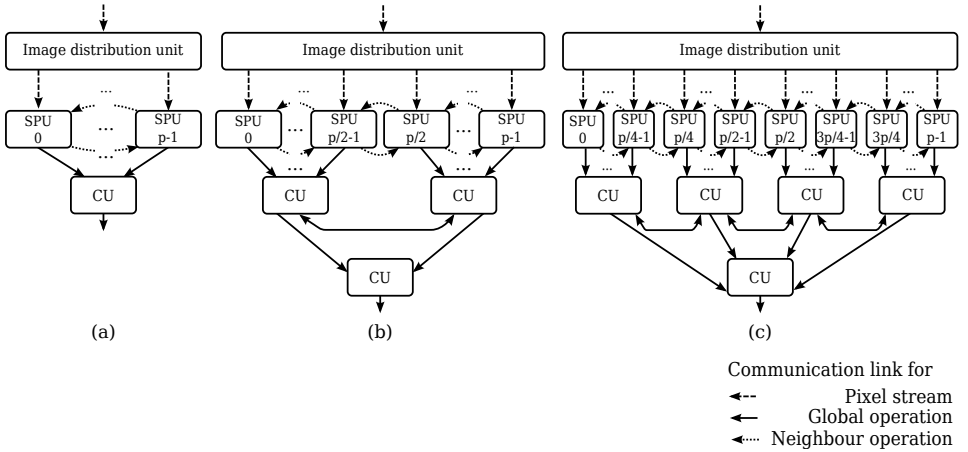



**Figure 5.9:** *PCCA* architecture utilising different numbers of CUs: in (a) a single CU is used, in (b) a tree of CUs consisting of 3 CUs is used and in (c) a tree of CUs consisting of 3 CUs is used.

## 5.4 Resource Sharing within the PCCA Architecture

"The problem of sharing a set of limited resources between users (customers) in an optimal way is fundamental" ([93]). *Resource sharing* in hardware architectures reuses functional units by time-multiplexing [15]. This can be implemented in a simple form by adding multiplexers at the inputs and outputs of the shared resources [107], or in a more advanced form by rearranging and connecting several functional units [15]. In the *PCCA* architecture, a small number of coalescing units process the global operations issued by multiple slice processing units (SPUs). This is implemented by the multiplexing and arbitration structures within the coalescing unit, described in Sections 4.4.2 and 5.3.

Figure 5.1 on page 156 shows the arrangement of multiple coalescing units in the *PCCA* architecture for scalability. In this section, without loss of generality, the FPGA implementations of the three PCCA variations shown in Figure 5.9 are discussed and evaluated:

- PCCA architecture consisting of $p$ SPUs connected to a single CU (Figure 5.9(a)).
- PCCA architecture consisting of $p$ SPUs connected to two CUs of *lvlgrp* 1 and one root CU of *lvlgrp* 0 (Figure 5.9(b)).
- PCCA architecture consisting of $p$ SPUs connected to four CUs of *lvlgrp* 1 and one root CU of *lvlgrp* 0 (Figure 5.9(c)).

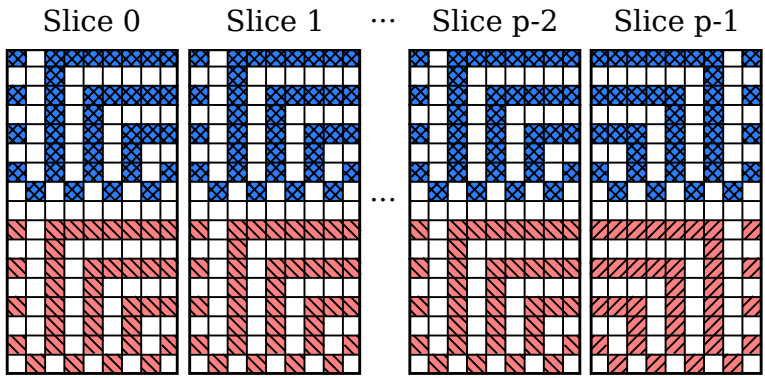



**Figure 5.10:** Worst case image with maximum number of global operations.

Section 5.5 shows the hardware resources required for the implementation on an FPGA device and the maximum possible clock frequency the PCCA architecture can be operated with. To determine the maximum throughput which can be processed in real-time with these hardware architectures, a worst case image pattern is derived in the following issuing the maximum number of global operations to the coalescing unit.

### 5.4.1 Determination of Maximum Throughput for Real-Time Processing

The *PCCA* architecture is only able to process the data rate of the pixel stream at its input if both of its major sub-components, the SPUs and the CUs, are able to keep up with the data rates at their respective inputs, independent of the values of the pixel stream. An SPU receives a stream of binary pixels from the IDU and issues global operations (GOs). These GOs are forwarded to the input of a single CU or multiple CUs. In Section 3.2.3 it was shown that for an SPU there is a direct relation between the input image size and the number of required clock cycles to process the input image. To increase the throughput of the *PCCA* architecture, the number of SPUs is increased, where each SPU processes one of the $p$ image slices. Such an increase in the number of slices also increases the number of slice borders increasing the maximum number of GOs to be processed by a single CU or multiple CUs. The rate an SPU issues GOs varies significantly depending on the number of border edges the pixel graph of the input image contains. For an input image in which the pixel graph $G_P$ does not contain any slice-components, for instance, no GOs are issued. The throughput of a CU is, therefore, determined by an input image $I$ for which the $p$ SPUs issue a maximum number of GOs. Such a *worst case image* resulting in a maximum number of GOs issued by the SPUs is shown in Figure 5.10 and derived in the following.

Construction of the Worst Case Image with a Maximum Number of Global Operations

'To analyse the performance and determine whether real-time requirements can be met, it is crucial to determine the maximum number of global operations which can be induced by patterns of binary pixels in the input image. The CUs carry out global label operations (*GLOs*) and global component operations (*GCOs*). The number of clock cycles the implementation of the PCCA hardware architecture requires to execute the maximum number of GOs depend on the image width $W_{image}$ and the number of image slices $p$ processed in parallel, i.e. the slice width of $W_S = \lceil W_{image}/p \rceil$. For analysis of the maximum number of global label operations, the properties of *GOs* are examined at first: A *GNLO* is the only *GO* which associates a new global label with a connected component, a $GMO_{Slice}$ joins the vertices associated with two global labels and a *GCO* recycles a global label. The pattern inducing $GMO_{Slice}$ operations is called $GMPat_{Slice}$. A maximum of $\lfloor W_S/2 \rfloor$ $GMPat_{Slice}$ patterns are possible in one row of an image slice, since each $GMO_{Slice}$ operation requires two different global labels associated with the pixels of the image row above. A prerequisite for each of these $GMO_{Slice}$ is that two different global labels are generated by *GNLO* operations before, whose vertices are not yet joined with other vertices of global labels. The maximum number of $\lfloor W_S/2 \rfloor$ $GMPat_{Slice}$ patterns which induce one $GMO_{Slice}$ each, therefore, requires at least $\lceil W_S/2 \rceil$ global labels.

The maximum number of $\lfloor W_S/2 \rfloor$ $GMPat_{Slice}$ patterns in a single image row can only occur every $W_S$ image rows, as at least $\lfloor W_S/2 \rfloor$ *GNLO* are necessary to generate the required number of global labels. For this reason, the worst case image with respect to the maximum number of required global label operations has the following properties:

1. $\lfloor \frac{W_S}{2} \rfloor$ global labels are generated by *GNLOs* at each slice border before a merger operation is induced.
2. As soon as $\lceil W_S/2 \rceil$ global labels are generated via *GNLO* operations, their associated vertices are joined using $GMO_{Slice}$ operations in the following row.
3. The image row after the series of $GMPat_{Slice}$ patterns inducing $GMO_{Slice}$ consist of background pixels, i.e. all connected components are completed.
4. The properties given in 1 through 3 repeat periodically every $W_S$ rows of the image. '[68]

An image complying with these properties is shown in Figure 5.10. The cross-hatched pixels show a pattern matching the properties 1 through 3; for the following benchmark it is, therefore, considered the worst case pattern inducing the maximum number of global operations. To fulfil property 4 this worst case pattern is placed every $W_S$ image rows, as shown in Figure 5.10 by the hatched pixels. The maximum number of worst case patterns fit in an image if the slice width, $W_S$, is an integer

multiple of the image height $H$. Then, there are $\lfloor \frac{H}{W_S} \rfloor$ worst case patterns, inducing a total number of $N_{max}$ GOs, where

$$N_{max} = \Big( \underbrace{(p-1) \times \lfloor \frac{W_S}{2} \rfloor}_{\#\text{ GNLOs}} + \underbrace{\lfloor \frac{W}{2} \rfloor}_{\#\text{ GMO}_{Slice}} + \underbrace{(p-2) \times \lfloor \frac{W_S}{2} - 1 \rfloor + p}_{\#\text{ GCOs}} \Big) \times \underbrace{\Big( \lfloor \frac{H}{W_S} \rfloor \Big)}_{\#\text{ Worst case patterns}} . \tag{5.5}$$

Evaluation of the Throughput

A *PCCA* architecture for a specific set of the parameters *image width*, *number of image slices* and *number of CUs* is called an instantiation of PCCA. An instantiation of *PCCA* discussed in Section 5.5 is capable of real-time processing if the following criterion applies: The time $t_{WC,IDU}$ in which one worst case image is received at the input of the *IDU* is less than or equal to the maximum of

- the time to process the received pixel data with the SPUs $t_{WC,SPU}$, and
- the time to process the issued GOs in one or multiple CUs $t_{WC,CU}$.

$$t_{WC,IDU} \geq \max(t_{WC,SPU}, t_{WC,CU}) \tag{5.6}$$

If the criterion from Equation 5.6 is fulfilled, a true upper bound for the processing latency is determined and the architecture is able to keep up with the data rate of the pixel stream at its input, i.e. real-time processing is possible.

For each instantiation of PCCA presented in Section 5.5 the real-time criterion from Equation 5.6 is verified using a behavioural simulation of the *VHDL* description. If the real-time criterion is fulfilled, the throughput $T_{RT}$ of the instantiation is calculated from the product of the PAR (place&route) frequency $f_{PAR}$, the number of SPUs $p$ and the factor 5/6 for stack processing in the SPUs (explained in Section 3.1.2).

$$T_{RT} = f_{PAR} \times p \times \frac{5}{6} \tag{5.7}$$

## 5.5 Experimental Results and Discussion

The diagrams in the following show the number of slice registers, lookup tables and block-RAM (BRAM) bits used, as well as the maximum clock frequency the implementation of the *PCCA* architecture can be operated at. The used device families, *Xilinx Virtex 6* and *Xilinx Virtex 7*, provide BRAMs with a size of 18 KBit and 36 KBit, respectively. The diagrams showing the number of required BRAM bits, therefore, consider the full capacity each used BRAM for the comparison even if not the entire capacity of a BRAM is used. The lookup tables (LUTs)

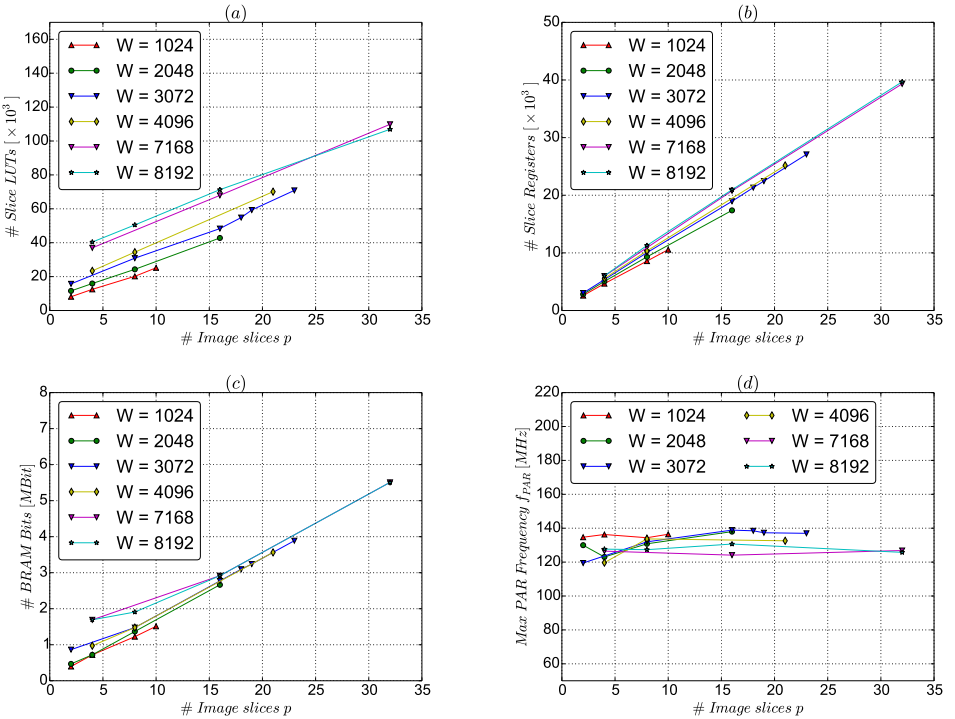



**Figure 5.11:** These diagrams show the number of used lookup tables (LUTs) in (a), slice registers in (b) and BRAM Bits in (c) required to implement the PCCA architecture on a *Xilinx Virtex 6* $VLX240T-2$ FPGA device for different image widths $W$ and different numbers of image slices $p$. In (d) the maximum place&route (PAR) frequency $f_{PAR}$ is shown.

shown in the diagrams are either used to realise logic functions with up to six inputs [132, 134] or to realise small memories as distributed RAMs [130, 133]. The reported maximum frequency, the maximum *place&route (PAR) frequency*, is the maximum frequency reported by Xilinx PlanAhead/Vivado tool-chain after the place and route step, i.e. it considers the wire delay and the logic delay. The *PAR frequency* is the maximum frequency the *PCCA* architecture can be operated at. To acquire comparable mapping and timing results, the *PlanAhead 14 default* implementation strategy was used on Virtex 6 and the *Vivado Implementation Defaults (2015)* implementation strategy was used on Virtex 7 and nothing but the *PCCA* architecture was implemented on the FPGA device.

The diagrams in Figure 5.11 show the results of the implementation of the *PCCA* architecture utilising a single CU (Figure 5.9(a)) on a *Xilinx Virtex 6 VLX240T-2* FPGA device. The image size was varied from 1 Megapixel to 32 Megapixel distributed over 32 image slices. The required resources are dependent on the width

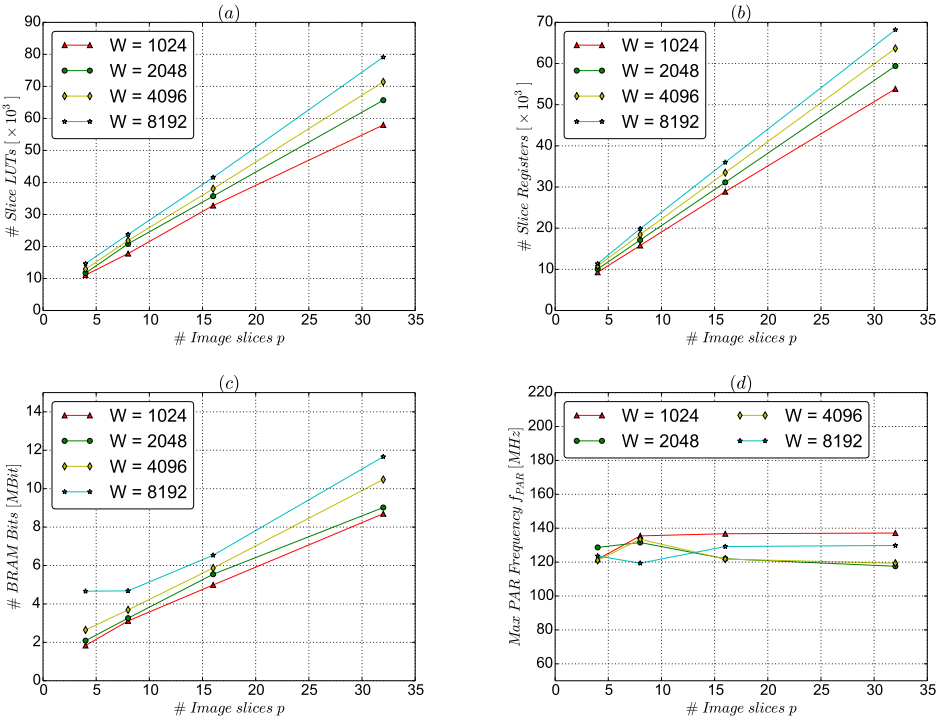



**Figure 5.12:** These diagrams show the number of used LUTs in (a), slice registers in (b) and BRAM Bits in (c) required by the implementation of the PCCA architecture with three CUs on a *Xilinx Virtex 7 XC7V585T−2* FPGA device for different image widths $W$ and different numbers of image slices $p$. In (d) the maximum place&route (PAR) frequency $f_{PAR}$ is shown.

$W$, shown in the legends of these diagrams. Note that for small image width, the degree of parallelism (number of slices) is limited by the processing capacity of a single CU. The number of LUTs, slice registers and the number of used BRAM bits all scale approximately linearly with the number of image slices. Figure 5.11(d) shows that the PAR frequency is between 120 MHz and 140 MHz independent of the image width or the number of image slices when *PCCA* uses a single CU.

The diagrams in Figure 5.12 show the results of the implementation of the *PCCA* architecture utilising a tree of three CUs on a *Xilinx Virtex 7 XC7v585t-2* FPGA device. This tree of CUs is arranged as shown in Figure 5.9(b). There is one root CU of *lvlgrp* 0 and two CUs of *lvlgrp* 1. Results are shown for images from 1 Megapixel to 32 Megapixel which are distributed over up to 32 image slices. The number of required LUTs and slice registers both increase linearly with the number of image slices $p$. The number of BRAM bits depend on the image size and scale linearly

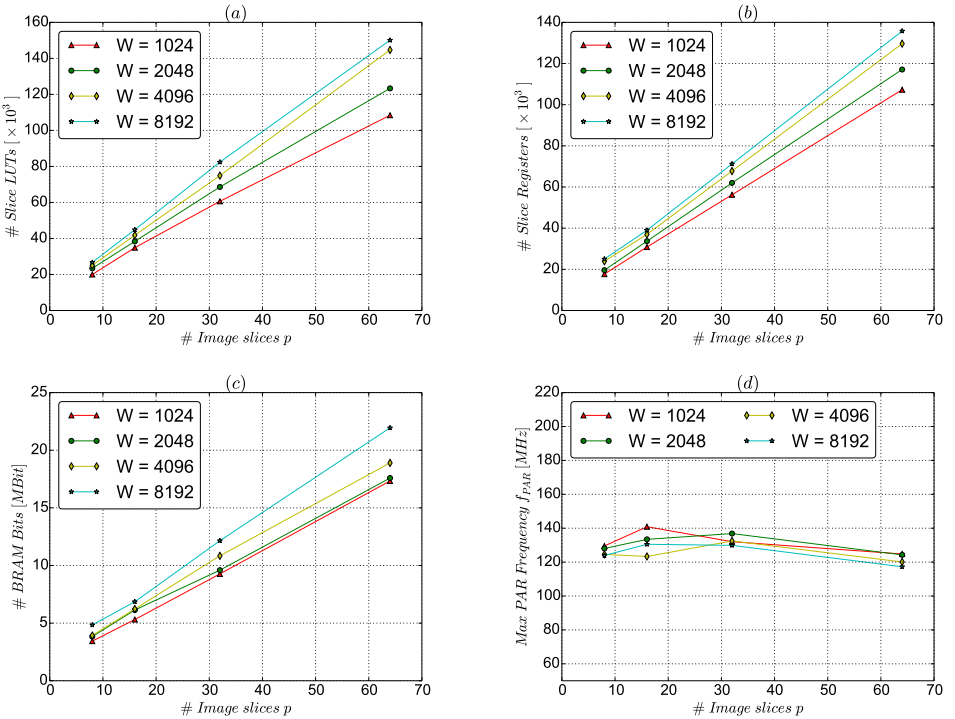



**Figure 5.13:** These diagrams show the number of used LUTs in (a), slice registers in (b) and BRAM Bits in (c) required by the implementation of the PCCA architecture applying five CUs implemented on a *Xilinx Virtex 7 XC7V585T − 2* FPGA device for different image widths $W$ and different numbers of image slices $p$. In (d) the maximum place&route (PAR) frequency $f_{PAR}$ is shown.

with the number of image slices. Figure 5.11(d) shows that the PAR frequency is between 118 MHz and 138 MHz independent of the image width or the number of image slices when *PCCA* uses a tree of three CUs.

The results of the implementation of the *PCCA* architecture utilising a tree of five CUs on a *Xilinx Virtex 7 XC7v585t-2* FPGA device are shown in the diagrams in Figure 5.13 for image sizes from 1 Megapixel to 32 Megapixel distributed over up to 64 image slices. These five CUs are arranged as shown in Figure 5.9(c). There is one root CU of *lvlgrp* 0 and four CUs of *lvlgrp* 1. Both the number of LUTs and slice registers increase linearly with the number of image slices $p$. The number of used BRAM bits depends on the image size and the number of image slices. This shows that the number of required BRAM bits scale linearly with the number of image slices. The PAR frequency $f_{PAR}$ for 8 to 32 images slices is between 120 MHz

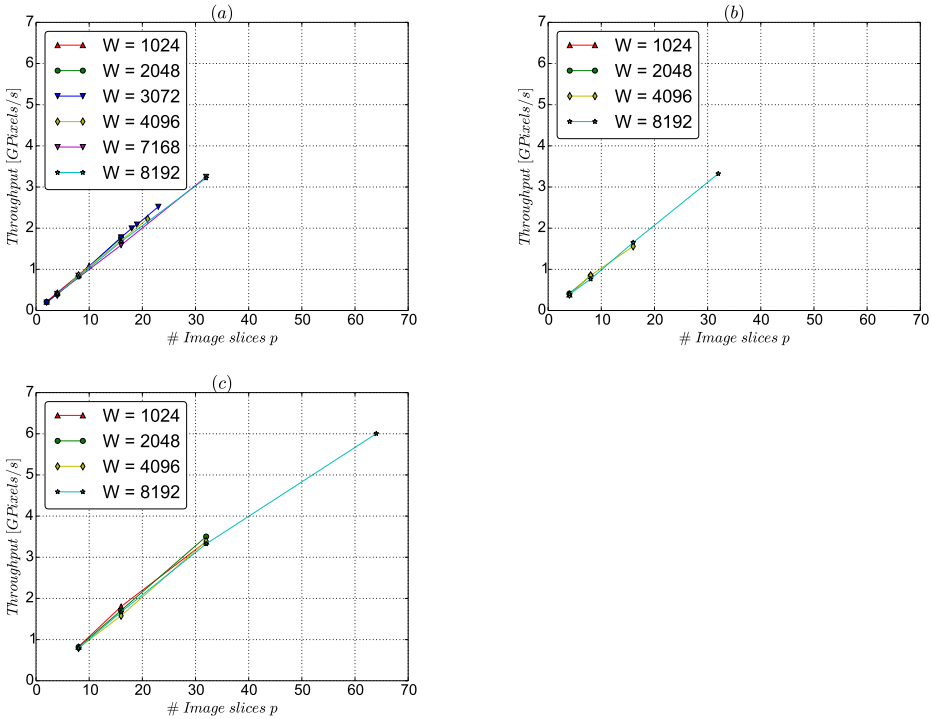



**Figure 5.14:** Throughput for processing the worst case image shown in Figure 5.10 using a single CU (a), three CUs (b) and five CUs (c).

and 140 MHz. For 64 image slices, the PAR frequency $f_{PAR}$ is slightly below 120 MHz.

The diagrams in Figure 5.14 show the maximum throughput for processing a series of worst case images, as presented in Section 5.4: in (a) when using a single CU, in (b) for using three CUs and in (c) for using five CUs. Only the instantiations which fulfil the real-time criterion from Equation 5.6 are shown in the diagrams in Figure 5.14. The processing of global operations for PCCA architectures with multiple CUs requires more instructions (see Pseudocode 19 and Pseudocode 20). Figure 5.14(a) and 5.14(b) show that using a single or three CUs results in almost the same throughput, with a maximum of approximately 3.2 $Gigapixel/s$ when dividing the input image to 32 slices. However, multiple CUs require more complex processing. For these cases, the PCCA instantiation with a single CU is more efficient, as it requires fewer hardware resources. However, a single CU limits further acceleration, as the number of GOs increases with the number of image slices. The diagram in Figure 5.14(c) shows the maximum throughput of the PCCA instantiations with five CUs, arranged as shown in Figure 5.9(c), which fulfil the real-time criterion from

Equation 5.6. A maximum throughput of 6 Gigapixel/s is achieved by the PCCA instantiation applying 64 SPUs and five CUs.

| | Parallelism $[\frac{Pixels}{cycle}]$ | Max. frequency $f_{max}$ [MHz] | Processing device | Throughput $[\frac{GPixels}{s}]$ |
|---|---|---|---|---|
| Kumar et al. [74] | 6 | 100[a] | Xilinx Virtex 5 | 0.27 |
| Li et al. [80] | $4 \times p$ | 100 | ASIC 0.35 $\mu m$ | 0.13[b] |
| PCCA predecessor [71] | 32 | 138.8 | Xilinx Virtex 6 | 3.5 |
| PCCA | 64 | 117.2 | Xilinx Virtex 7 | 6.0 |

**Table 5.1:** Comparison of throughput with other parallel hardware implementations which process multiple pixels per clock cycle.

[a] This is the frequency the architecture is operated at, as described in [74].
[b] This throughput value is extrapolated, as described in Section 5.5.1.

### 5.5.1 Comparison to Other Parallel Hardware Architectures

Table 5.1 compares the implementation of the architecture of *PCCA* with the results published for other CCA or CCL hardware architectures that process multiple pixels per clock cycle. The column *parallelism* in Table 5.1 compares the maximum number of pixels which are processed simultaneously. In the *maximum frequency* column the highest clock frequency $f_{max}$ that the architecture can be operated with is shown. For *PCCA*, $f_{max}$ is equal to $f_{PAR}$ evaluated in Section 5.5. The *throughput* column in Table 5.1 shows the maximum throughput reported for an architecture. As the processing devices used for the implementation of the considered architectures are realised with a different process technology, from a 0.35 $\mu m$ on an *ASIC* [80] to 28$nm$ on an $FPGA$, an isolated comparison based on the throughput is of limited value. In the following, each of the architectures from Table 5.1 is compared at an architectural level and with respect to the maximum throughput with the architecture *PCCA*.

Comparison with Kumar et al. [74]

The CCL architecture in [74] divides the input image into $p$ horizontal image slices processed in parallel. The reported results are achieved when operating the architecture at 100 MHz. A maximum clock frequency for the implementation of the architecture on a *Xilinx Virtex 5 FX130T* FPGA device is not provided, i.e. the reported 100 MHz is used for the comparison in Table 5.1. The maximum throughput is reported for processing a pixel stream with an image size of $800 \times 600$ pixels at a frame-rate of 580 fps resulting in the maximum throughput of 278 Megapixel/s. This maximum throughput is achieved when the input image is divided into 6 slices, i.e. 6 pixels are processed simultaneously.

As the architecture presented in [74] processes horizontal image slices, an image stream providing pixels in forward raster scan order, e.g. as provided by most modern image sensors, must buffer most of the input image before CCL is started. *PCCA*, in contrast, does not require such a buffer of the entire input image. In [74], the first and the last row of each slice of the labelled image have to be buffered until the end of the image to combine feature vectors of slice-components. *PCCA* combines feature vectors of slice-components on the fly without storing several rows of the labelled image. The instantiation of *PCCA* applying 64 image slices and five CUs achieves a throughput which exceeds the throughput reported in [74] by a factor of more than 20.

Comparison with Lin et al. [80]

The CCL architecture presented in [80] uses a two-pass algorithm which divides the image into $p$ horizontal image slices processed in parallel. In each of these slices, four binary pixels are processed simultaneously. The maximum number of slices that can be processed in parallel is not specified in [80]. As four pixels are processed simultaneously in every one of the $p$ image slices, the maximum number of pixels processed simultaneously is given as $4 \times p$ in Table 5.1. To reduce the required on-chip memory for the parallel CCL architecture, a maximum of 4096 labels are used by the architecture in [80]. However, for two-pass algorithms, worst case images (with the maximum number of labels) requiring $\lceil \frac{W \times H}{4} \rceil$ labels are usually considered, as every second pixel in every second image row can be a distinct connected component. It is not reported whether such worst case images can be processed.

In [80], it is reported that when operated at a frequency of 21 MHz, the presented architecture is able to process a pixel stream with an image size of $1280 \times 720$ at a frame-rate of 30 fps (≈28 Megapixel/s) by separating the input image to $p = 4$ slices. As a maximum frequency, $f_{max} = 100\ MHz$ is reported. The maximum throughput of this architecture, shown in Table 5.1 is, therefore, a result of an extrapolation of these values: 131.65 Megapixel/s. The instantiation of *PCCA* applying 64 image slices and five CUs achieves a throughput which exceeds the throughput reported in [80] by a factor of more than 40.

Comparison with PCCA predecessor [71]

The CCA architecture in [71] is an early version of *PCCA*. In a similar manner to *PCCA*, it divides the image into $p$ vertical slices which are processed simultaneously. The feature vectors associated with slice-components are combined by a coalescing unit (CU). A major difference of *PCCA* compared to [71] is that the latter requires all labels associated with border pixels to be stored for association of slice-components with their connected component at the end of the frame. This leads to a significantly higher amount of on-chip memory required [64]. The CU of the architecture of [71]

starts combining feature vectors associated with slice-components at the end of the image. Therefore, also the latency to extract feature vectors of connected components spanning several image slices is significantly longer than for *PCCA*. The throughput of *PCCA* is almost doubled compared to [71] to a maximum of 6 Gigapixel/s. There are two major reasons for this: The reduction of the on-chip memory resources and the arranging of several CUs into a CU tree.

## 5.6 Summary and Contributions of the PCCA Hardware Architecture to the State of the Art

The *PCCA* hardware architecture proposed in this chapter is an improvement on the state of the art on several levels, as pointed out in the following.

- **Increased throughput to process high-speed pixel streams:** The *parallel SLCCA*, *PCCA*, hardware architecture achieves a high throughput of up to 6 Gigapixel/s. This allows a stream of ultra-high-definition *UHD8k* images to be processed with a frame-rate of up to 169 fps, which is higher than the maximum specified frame-rate for *UHD8k* of 120 fps.
- **Scalability of hardware resources with throughput:** The throughput is increased by using multiple CUs processing up to 64 pixels per clock cycle.
- **Hierarchical tree structure of coalescing units:** The aforementioned high throughput is possible by distributing workload to multiple coalescing units. These coalescing units are connected in a tree structure. Previous parallel CCA architectures were limited by a single coalescing unit becoming the bottleneck for a further increase of the throughput.

# 6 Demonstration of the PCCA Architecture

The *PCCA* architecture was demonstrated in mechanical process engineering to measure the droplet size [70], the droplet size distribution [72], the pulsation frequency [70], the droplet circularity [69] and to characterise non-spherical objects [69]. In this chapter, the *PCCA* hardware architecture (which contains the *SLCCA* hardware architecture) is demonstrated in case study to detect droplet collisions in real-time in an atomisation process, as described in the following.

Atomisation is a sub-field of mechanical process engineering concerned with producing droplets by separating liquids with nozzles. In foundational research in this field, high-speed camera systems are used to explore the impact of different process parameters to atomisation processes [122]. The image data collected in such experiments range from several MByte to several TByte for each set of atomisation parameters. State-of-the-art high-speed camera systems store the captured image frames to an internal memory which is read out and transferred to a workstation PC for further evaluation where information of the observed droplets and particles such as object size, object position, size distribution or shape is extracted. If the event to examine occurs seldom, a repetitive cycle of image capturing and offline analysis is required, because the recording time of the high-speed camera system is limited by the internal memory. To accelerate this process a high-speed *Real-time Process Analysis System* is presented in Section 6.1 using the *PCCA* architecture from Chapter 5 to analyse all captured image frames in real-time. The extracted results are evaluated as explained in Section 6.2 and used to trigger the read-out process of the *Real-time Process Analysis System* allowing to only send images to the workstation which match predefined criteria. The case study in Section 6.3 shows the application of this system for monitoring and detecting droplet collision.

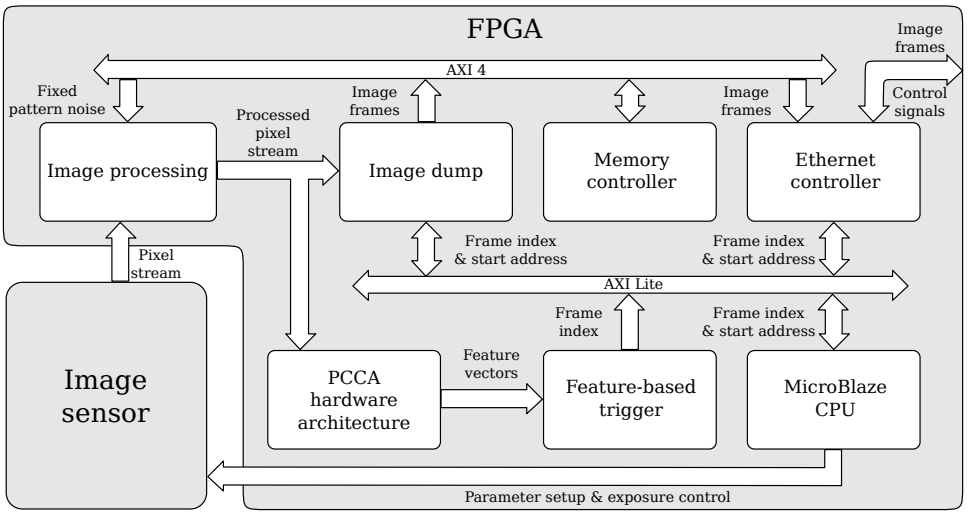
## 6.1 A Real-time Process Analysis System based on FPGA Hardware Acceleration

'The *Real-time Process Analysis System* consists of two main components: a high-speed image sensor to capture image series at high frame rate and a field-programmable gate array (FPGA) to process the massive amount of image data created by the high-speed image sensor. To display and store images and to control the presented imaging system, it is connected to a workstation PC via a Gigabit Ethernet connection. The high-speed image sensor used, captures image frames with a resolution of up to 3 Megapixel at a frame rate between 485 fps and 10,000 fps [96]. All image frames received by the image sensor are written to a memory inside the *Real-time Process Analysis System* used as a *ring buffer*. Additionally, the feature vectors of all connected components in an image frame are extracted using the *PCCA* architecture. The extracted feature vectors describe certain properties of each object in a captured image, such as size or area. In this chapter, a feature vector consists of the *height* and the *width* of an image object extracted from a captured image frame. All feature vectors extracted from one image frame usually require only a few KByte depending on the complexity of the image content. The pixel data add up to several MByte per image, depending on the image resolution. The Ethernet interface to the workstation PC, however, is limited to transfer up to 125 MByte/s. Therefore, it is possible to transfer all the extracted feature vectors of each image frame to the workstation, but not the pixel data of each captured image frame. To decide which image frames to send, three transfer modes are available:

- Consecutive mode
- Skip mode
- Trigger mode

In *consecutive mode* writing to the *ring buffer* is stopped and the image frames stored in this memory are transferred to the workstation. After these frames are transferred successfully, image data received from the high-speed image sensor is stored in the *ring buffer*, again. In *skip mode* continuously the latest image frame, which is captured, is sent to the workstation. All the image frames captured by the image sensor while transferring this image to the workstation are discarded. In *trigger mode* a novel feature-based trigger mechanism identifies image frames which objects match a predefined pattern. Only these image frames are transferred to the workstation.'[69]

'The block diagram in Figure 6.1 depicts the communication links on the FPGA and to the image sensor, together with the processing units on the FPGA. The direct chip-to-chip connection of the high-speed image sensor and the FPGA which consists of one high bandwidth link to transfer the image data as pixel stream from the image sensor to the FPGA and a control channel to set parameters such




[69]

**Figure 6.1:** Block diagram of the architecture of the *Real-time Process Analysis System* .

frame rate, image resolution or exposure time. The high bandwidth link consists of 16 serial source synchronous LVDS (Low Voltage Differential Signalling) channels providing sufficient bandwidth for the output data of the high-speed image sensor of more than 10 Gigapixel/s. The individual image processing units on the FPGA communicate via an *AXI bus* [131]. This on-chip communication is illustrated in the sequence diagram in Figure 6.2. Each step in Figure 6.2 marked by a circled number is described in the following. The first processing step on the FPGA is carried out in the *image processing unit* which receives the high-speed pixel stream from the image sensor (step ①) and enhances the image stream by removing fixed pattern noise, which is a constant deviation of the intensity for each pixel individual to each image sensor introduced by its manufacturing process. After this step the enhanced frame and its frame index are forwarded to the *image dump unit* (step②) and the *PCCA* unit (step ③). The *image dump unit* stores the pixel data of each frame to the *ring buffer* by using the *memory controller* (step ④) and keeps a record of the starting address in the memory for each frame which is forwarded to the *MicroBlaze CPU* (step ⑤). In skip mode and consecutive mode the *MicroBlaze CPU* provides either the frame index and start address of the latest captured image to *Ethernet controller* or the frame indices and start addresses of all images stored in the *ring buffer* (step ⑧). The *Ethernet controller* reads from the *ring buffer* via the *memory controller* and transfers images to the workstation (steps ⑨-⑪). The *PCCA* unit extracts the feature vectors of all image objects from each image frame, as described in Chapters 4 and 5. Here, the height and the width of each image object is extracted. The extracted feature vectors are forwarded to the *Ethernet*

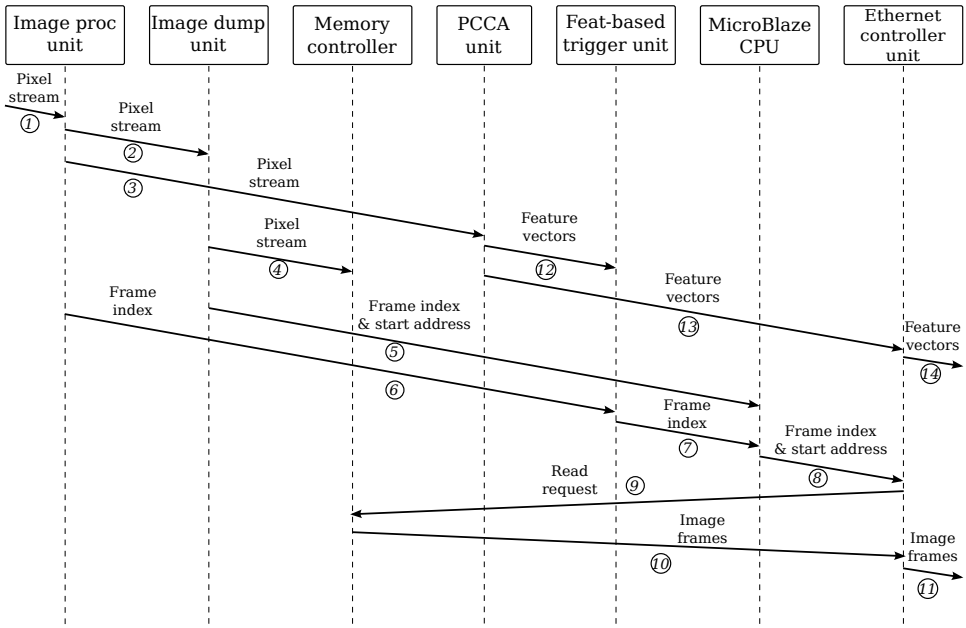



**Figure 6.2:** Sequence diagram for communication between image processing components of Real-time Process Analysis System . *[69]*

*Controller* (⑬) which transfers all feature vectors to the workstation PC (⑭) and to the *feature-based trigger* unit (⑫).

In trigger mode only the frame indices of image frames which are identified to contain feature vectors to match predefined criteria are forwarded to the *MicroBlaze CPU*, as shown in Figure 6.2 ⑦. For the consecutive and skip mode all frame indices received by the *feature-based trigger* unit (⑥) are passed to the *MicroBlaze CPU* (⑦). The *feature-based trigger* unit compares every feature vector received by the *PCCA* unit with a set of previously defined criteria. In Figure 6.3 the architecture of the *feature-based trigger* unit is shown on register-transfer level. Every feature vector containing the width and the height of one image component compares these values with a minimal width $W_{min}$, a minimal height $H_{min}$ and a minimal difference $D_{min}$ which is the absolute value of the difference of $W_{min}$ and $H_{min}$. The values of $W_{min}$ and $H_{min}$ are used to filter out image objects below a minimum size. The value of $D_{min}$ is an efficient way to implement and approximation of the circularity $C$ (defined in Section 6.2) without using divisions. It is especially useful to identify objects which height and width differ from each other. If the feature vector of at least one image component fulfils all of these three comparisons, a trigger match is detected. Then the image index of the current frame and the indices of $F_{con}/2$

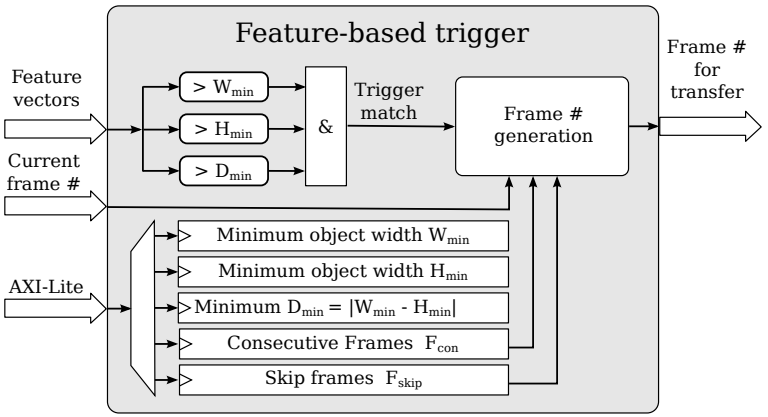



**Figure 6.3:** Architecture of *feature vector trigger* unit on register-transfer level. *[69]*

image frames before and after the current frame are passed on to the *MicroBlaze CPU* to transfer the corresponding image frames to the workstation. However, the bandwidth of the image data transfers resulting from frequent trigger matches still can exceed the bandwidth of the Ethernet connection. To prevent this, no image transfer is initiate for the duration of capturing $F_{skip}$ images. This limits the maximum output bandwidth by the ratio of $F_{con}/F_{skip}$.'[69]

## 6.2 Feature Vector Evaluation and Interpretation

'The *PCCA* architecture provides the feature vectors of connected components extracted from a binary image. However, taking images from spray and atomisation processes requires processing on object level rather than on a connected component level, because one physical object might be detected as several connected components in the image due to optical and mechanical properties of the measurement system. These effects are depicted in Figure 6.4 and require further processing which is explained in the following. The image of an out-of-focus object gets blurred depending on its distance to the focal plane. After transforming a grayscale image to a binary black and white image using a global threshold level [39], some of the pixels of the blurred image are detected as background and some as object pixels. Figure 6.4(a) shows such a grayscale image and Figure 6.4(b) the corresponding binary image. This results in detecting several connected components from one physical object. In Figure 6.4(c) an example is shown where several small connected components are detected due to the gradient in the grayscale image. Capturing image series of objects with high velocity requires a low exposure rate in the range of micro-seconds or below, which makes illumination challenging and hard to establish homogeneous

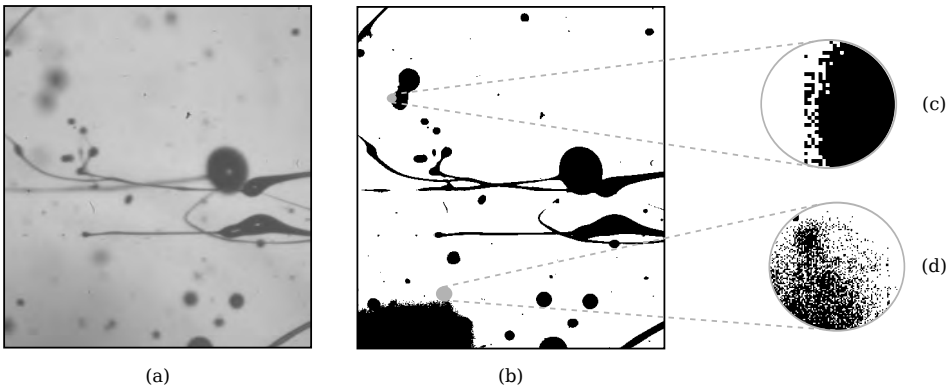
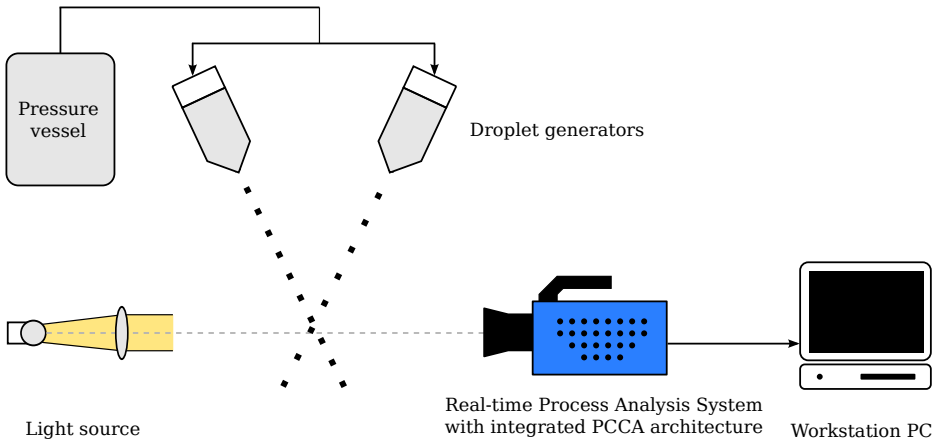



**Figure 6.4:** (a)Grayscale image, (b)Binarised image after global thresholding, (c)Out-of focus object, (d)Noise due to insufficient illumination. *[69]*

background illumination. This affects the difference of the intensity level between object and background pixels which lead to several detected connected components from one physical object, as well. Figure 6.4(d) shows an example of an image where insufficient illumination leads to the detection of a large connected component and noise pixels, both of which would be detected as droplets or particles. The relative error due to quantization is the higher, the closer the image of an object is to the pixel size of the image sensor. For this reason only feature vectors of connected components above a minimum height $H_{min}$ and above a minimum width $W_{min}$ create result accurate enough to be considered for further evaluation.

The feature vectors extracted by the *PCCA* unit consist of the height $h$ and width $w$ for each connected component. For real-time evaluation each feature vector is considered and analysed. '[69]




[69]

**Figure 6.5:** Setup of droplet generators and imaging system to induce droplet collisions as proposed in [75]. *[69]*

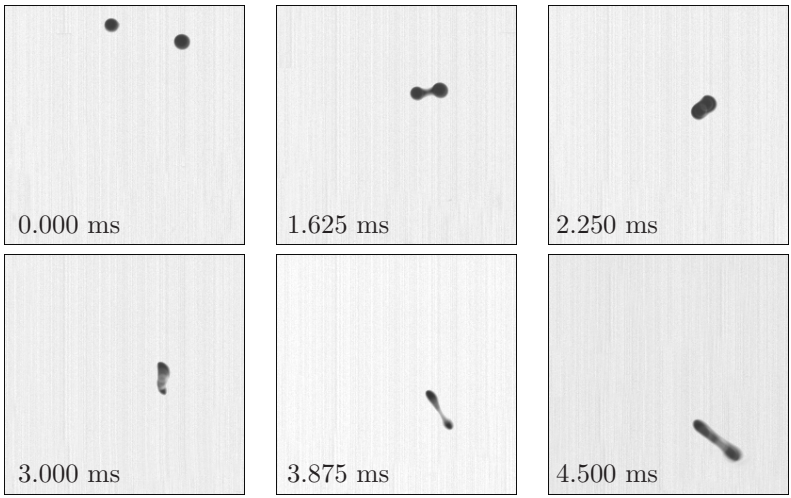
## 6.3 Case Study: Detection of Collisions in Atomisation Processes

' The collision behaviour of droplets and the collision outcome for high viscous polymer solutions are a recent research topic in mechanical process engineering requiring high-speed imaging [75]. To generate collisions, two droplet generators are directed to each other. Capturing a collision requires many cycles of image capturing and offline processing. Using the Real-time Process Analysis System , as presented in Section 6.1, all captured images can be analysed. In order to evaluate the detection of droplet collisions using this system, a setup similar to [75] is used which consists of two droplet generators connected to a liquid supply in a pressure vessel, a high-power LED light source and the Real-time Process Analysis System capturing images and transferring them together with the associated measurement data to a workstation PC. This setup is depicted in Figure 6.5. There droplet collisions are distinguished in the taken images from the other droplets by their size and their shape. The droplets resulting from a collision are assumed at least twice as large in area or width. Based on this assumption the *feature-based trigger* unit, proposed in Section 6.1, is used to detect image frames in which a droplet collision occurs. For this to works the minimum width $W_{min}$ and height $H_{min}$ is increased above the size of a spherical droplet. Figure 6.6(a) six image frames of a droplet collision captured at a rate of 8,000 fps. The second image in Figure 6.6(a) with the time stamp *1.625 ms* complies with the trigger criteria. Therefore, the image frames before and after this frame are transferred to the to the workstation. '[69]

'Depending on the angle of incidence and the velocity of two colliding droplets, two cases can be distinguished, *separation* and *agglomeration*. In the case of *separation*, the two colliding droplets separate to two or more droplets after the collision, for *agglomeration* the droplets merge to one larger droplet. The collision of two droplets captured for the case of separation is shown in Figure 6.6. The parameters of the *feature-based trigger* unit is set to a minimum height of $H_{min} = 30$ pixels, a minimum width $W_{min} = 60$ pixels and $D_{min} = 20$ pixels. Therefore, image transfer is only triggered for images containing objects where the width $H$ is at least double the height $H$, which was observed in previous experiments comply with the feature vector of an object in the separation case. As a result three droplet collisions per minute were detected in average. Agglomeration is shown in the image series in Figure 6.6(b). To detect this the *feature-based trigger* unit is set to a minimum height of $H_{min} = 45$ pixels, a minimum width $W_{min} = 45$ pixels and $D_{min} = 0$ pixels, which filters only for image objects where the width and height exceed the size of not colliding droplets. In this case an average of three droplet collisions per minute can be observed, as well. With an off-the-shelf high-speed camera more than twenty minutes was required to capture a droplet collision, in average.'[69]

Therefore, it could be demonstrated that the application of the *PCCA* hardware architecture (PCCA unit) reduces the time required to capture a series of images containing seldom events, such as droplet collision. This is achieved by realising a feature-based triggering mechanism to evaluate the feature vectors which are extracted by the *PCCA* unit. For the foundational research in atomisation the *Real-time Process Analysis System* has the potential to reduce the time required for carrying out a single experiment significantly - from tens of minutes to several seconds.

(a) Separation




(b) Agglomeration

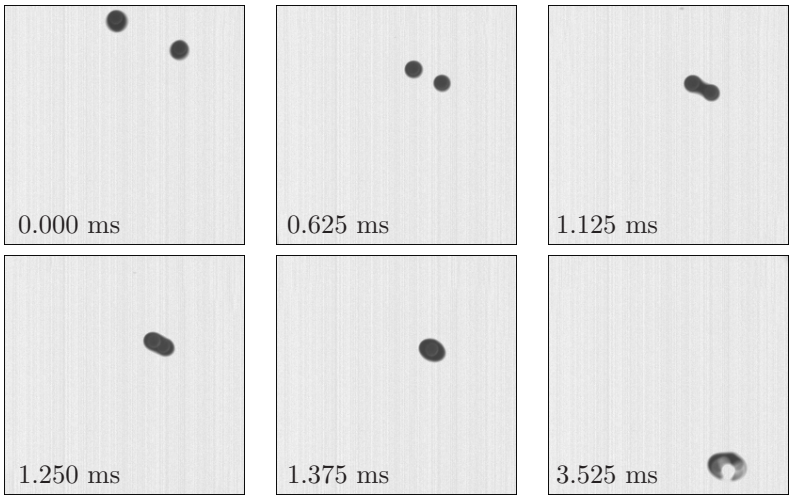



**Figure 6.6:** Series of high-speed images taken at a frame rate of 8,000 fps showing agglomrtation and separation after a collision of two droplets of Polyvinylpyrrolidon (PVP) K17-50%. *[69]*

# 7 Conclusion

The contributions of this dissertation to the state of the art are two connected components analysis hardware architectures capable of processing high-speed image streams in real time with high frame rates, e.g. 100 to 10,000 frames per second and beyond. This throughput makes it also possible to process pixel streams of ultra-high definition images, which have comparable data rates even for frame rates below 100 frames per second.

The improvements of these architectures dedicated for reconfigurable hardware are: high throughput by parallel processing, low latency by single lookup processing in a single pass, and linear scalability of hardware resources with the throughput. The linear scalability leads to a resource-efficient architecture.

The proposed *single lookup connected components analysis*, *SLCCA*, hardware architecture is more resource-efficient compared to other state-of-the-art architectures focusing on the goal of processing a single pixel per clock cycle. Whereas other architectures have to store the data of an entire image, *SLCCA* only requires that the data of one image row is stored due to its single-pass approach. The advanced union-find algorithm embodied in *SLCCA* only needs a single lookup per pixel, whereas other pairs of algorithm and architectures require multiple lookups. The previously most memory-efficient architecture depended on two data tables to extract feature vectors. As the design of the *SLCCA* architecture only requires a single data table, the amount of on-chip memory for storing feature vectors is halved. The total amount of on-chip memory of the *SLCCA* architecture depends on the image size and the number of features extracted from an image. The memory requirements for extracting features such as the *bounding box* grow logarithmically with the image width. To extract the three features *bounding box*, *area* and *first order image moment* from images with a resolution of 32 Megapixels (which corresponds to the size of ultra-high-definition *UHD8k* images), the amount of on-chip memory is reduced by 42% compared to the state of the art. Implemented on reconfigurable hardware, the *SLCCA* architecture achieves a throughput of up to 124 Megapixel/s. This throughput allows processing a 2 Megapixel image stream with up to 60 fps, which corresponds to processing a high-definition *HD1080p60* image stream. The processing latency of the *SLCCA* architecture is as low as the time needed to process two image rows, therefore it increases linearly with the image width. As a pixel is processed in a single clock cycle, a processing latency below 160$\mu s$ is achieved even for image streams of 32 Megapixel images. The number of memory access instructions *(MAIs)* required to process worst case images was compared between

state-of-the-art algorithms and *SLCCA*. According to these comparisons, *SLCCA* requires the least number of *MAIs* (comparing parallel *MAIs*) among all evaluated algorithms. Compared to the algorithm which is second in this comparison with regards to the number of required *MAIs*, *SLCCA* reduces the number of *MAIs* by a factor of 7.

The parallel *SLCCA*, *PCCA*, hardware architecture achieves a high throughput of up to 6 Gigapixel/s by dividing the input image into multiple slices, each of which is processed by an extended *SLCCA* architecture. This parallelisation allows a stream of 32 Megapixel images to be processed with a frame-rate of up to 169 fps, which is higher than the maximum specified frame-rate for ultra-high-definition *UHD8k* of 120 fps. The parallel algorithm used for the *PCCA* architecture is optimised for reconfigurable hardware as it uses spatial parallelism as well as temporal parallelism to accelerate processing. The hierarchical union-find algorithm embodied in *PCCA* allows simultaneous processing on tree structures of single connected components which makes parallel coalescing of feature vectors possible. The resources and on-chip memory bits required by the *PCCA* architecture scale linearly in the number of image slices due to the memory-efficient design. The input image stream is directly processed in a single pass and does not require a frame buffer to store the entire image, which allows the *PCCA* architecture to process the received image stream on the fly. Due to the scalable design of the *PCCA* architecture consisting of multiple *SLCCA* architectures and light-weight application-specific processors, it can be used for real-time high-speed image processing applications. The evaluated implementations of the *PCCA* hardware architecture processing up to 64 pixels per clock cycle achieve a throughput which is almost double that of the fastest parallel CCA hardware architecture proposed before (which is a predecessor of the *PCCA* architecture).

In a case study, the *PCCA* architecture, which includes the *SLCCA* architecture, is successfully demonstrated to process high-speed image streams captured in an experimental setup in real time. In this case study, the *PCCA* hardware architecture is used to detect droplet collisions in an experimental setup in mechanical process engineering. Therefore, the *PCCA* hardware architecture is integrated in a *Real-Time Process Analysis System* to extract feature vectors of the droplets in all images captured by a high-speed image sensor. The high throughput and low latency of the *PCCA* architecture is used to realise a feature-based triggering mechanism facilitating the *Real-Time Process Analysis System* to only output relevant images: those of droplet collisions. This feature-based triggering mechanism embedded into the *Real-Time Process Analysis System* reduces the time needed for carrying out a single experiment from tens of minutes to several seconds.

# List of Figures

# List of Tables

# Bibliography


[1] B. R. Acharya and P. K. Gantayat, "Recognition of human unusual activity in surveillance videos," *International Journal of Research and Scientific Innovation(IJRSI)*, vol. 2, no. 5, pp. 18 – 23, Jul 2015.

[2] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen*, vol. 99, no. 1, pp. 118–133, 1928.

[3] *Stratix V Device Handbook - Volume 1: Device Interfaces and Integration*, Altera Corporation, Jul 2014.

[4] M. Anandhalli and V. P. Baligar, "Improvised approach using background subtraction for vehicle detection," in *2015 IEEE International Advance Computing Conference (IACC)*, Jun 2015, pp. 303–308.

[5] K. Appiah, A. Hunter, P. Dickinson, and J. Owens, "A run-length based connected component algorithm for FPGA implementation," in *International Conference on Field Programmable Technology. FPT 2008*, Dec 2008, pp. 177 –184.

[6] K. Arulmozhi, S. Perumal, P. Sanooj, and K. Nallaperumal, "Application of top hat transform technique on indian license plate image localization," in *2012 IEEE International Conference on Computational Intelligence Computing Research (ICCIC)*, Dec 2012, pp. 1–4.

[7] D. Bailey and C. Johnston, "Single pass connected components analysis," in *Proceedings of Image and Vision Computing New Zealand 2007*, Dec 2007, pp. 282–287.

[8] D. Bailey, C. Johnston, and N. Ma, "Connected components analysis of streamed images," in *International Conference on Field Programmable Logic and Applications (FPL 2008)*, Sep 2008, pp. 679–682.

[9] B. Bässler, "Implementation and hardware accelerated verification of a connected component architecture," Master's thesis, University of Stuttgart, 2014.

[10] D. Burger, J. R. Goodman, and A. Kägi, "Memory bandwidth limitations of future microprocessors," *SIGARCH Comput. Archit. News*, vol. 24, no. 2, pp. 78–89, May 1996.

[11] L. Cabaret and L. Lacassagne, "What is the world's fastest connected component labeling algorithm?" in *International Workshop on Signal Processing Systems (SIPS)*. IEEE, 2014, pp. 1–6.

[12] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors," in *International Conference on Image Processing (ICIP)*, 2015.

[13] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel light speed labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors," *Journal of Real-Time Image Processing*, pp. 1–24, 2016.

[14] L. Cabaret, L. Lacassagne, and L. Oudni, "A review of world's fastest connected component labeling algorithms: Speed and energy estimation," in *Conference on Design & Architectures for Signal & Image Processing (DASIP)*, 2014, pp. 1–6.

[15] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. Boston, MA: Springer US, 2009, ch. Mapping and Execution Optimizations, pp. 109–154.

[16] S.-C. Chan, S. Zhang, J.-F. Wu, H.-J. Tan, J. Ni, and Y. Hung, "On the hardware/software design and implementation of a high definition multiview video surveillance system," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 2, pp. 248–262, 2013.

[17] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Computer Vision and Image Understanding*, vol. 93, no. 2, pp. 206 – 220, 2004.

[18] G. Chartrand, *Introductory Graph Theory*, ser. Dover Books on Mathematics Series. Dover, 1977.

[19] G. Chartrand, L. Lesniak, and P. Zhang, *Graphs & Digraphs, Fifth Edition*, 5th ed. Chapman & Hall/CRC, 2010.

[20] C.-W. Chen, Y.-T. Wu, S.-Y. Tseng, and W.-S. Wang, "Parallelization of connected-component labeling on tile64 many-core platform," *Journal of Signal Processing Systems*, vol. 75, no. 2, pp. 169–183, 2014.

[21] D. Chen, J.-M. Odobez, and H. Bourlard, "Text detection and recognition in images and video frames," *Pattern Recognition*, vol. 37, no. 3, pp. 595 – 608, 2004.

[22] Y.-L. Chen, "Nighttime vehicle light detection on a moving vehicle using image segmentation and analysis techniques," *WSEAS Transactions Computers*, vol. 8, no. 3, pp. 506–515, 2009.

[23] Y.-L. Chen, C.-T. Lin, C.-J. Fan, C.-M. Hsieh, and B.-F. Wu, "Vision-based nighttime vehicle detection and range estimation for driver assistance," in *IEEE International Conference on Systems, Man and Cybernetics. SMC 2008.* IEEE, 2008, pp. 2988–2993.

[24] Y. Chen, A. Abushakra, and J. Lee, "Vision-based horizon detection and target tracking for UAVs," in *Advances in Visual Computing*. Springer, 2011, pp. 310–319.

[25] N. Chiba and X. Liu, "Character extraction by integrating color into edge-based methods," in *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, May 2015, pp. 73–76.

[26] J. De Bock and W. Philips, "Fast and memory efficient 2-D connected components using linked lists of line segments," *IEEE Transactions on Image Processing*, vol. 19, no. 12, pp. 3222–3231, Dec 2010.

[27] L. Di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Proceedings International Conference on Image Analysis and Processing*, 1999, pp. 322–327.

[28] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *Journal of the ACM*, vol. 39, no. 2, pp. 253–280, Apr 1992.

[29] S. Du, M. Ibrahim, M. Shehata, and W. Badawy, "Automatic license plate recognition (alpr): A state-of-the-art review," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 2, pp. 311–325, Feb 2013.

[30] S. Ebrahimi and V. Y. Mariano, "Image quality improvement in kidney stone detection on computed tomography images," *Journal of Image and Graphics*, vol. 3, no. 1, 2015.

[31] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in DRAM for cache applications," *IEEE Micro*, vol. 28, no. 6, pp. 47–56, 2008.

[32] S. Estable, J. Schick, F. Stein, R. Janssen, R. Ott, W. Ritter, and Y.-J. Zheng, "A real-time traffic sign recognition system," in *Proceedings of the Intelligent Vehicles '94 Symposium*, Oct 1994, pp. 213–218.

[33] J. F. Eusse, R. Leupers, G. Ascheid, P. Sudowe, B. Leibe, and T. Sadasue, "A flexible asip architecture for connected components labeling in embedded vision applications," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, Mar 2014, pp. 1–6.

[34] A. Fossati, P. Schönmann, and P. Fua, "Real-time vehicle tracking for driving assistance," *Machine Vision and Applications*, vol. 22, no. 2, pp. 439–448, 2011.

[35] U. Franke, D. Gavrila, S. Görzig, F. Lindner, F. Paetzold, and C. Wöhler, "Autonomous driving goes downtown," *IEEE Intelligent systems*, no. 6, pp. 40–48, 1998.

[36] E. Frew, T. McGee, Z. Kim, X. Xiao, S. Jackson, M. Morimoto, S. Rathinam, J. Padial, and R. Sengupta, "Vision-based road-following using a small autonomous aircraft," in *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, vol. 5. IEEE, 2004, pp. 3006–3015.

[37] Y. Fu, X. Chen, and H. Gao, "A new connected component analysis algorithm based on max-tree," in *2009 Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2009, pp. 843–844.

[38] D. Geronimo, A. M. Lopez, A. D. Sappa, and T. Graf, "Survey of pedestrian detection for advanced driver assistance systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 7, pp. 1239–1258, 2010.

[39] R. C. Gonzalez and R. E. Woods, "Thresholding," *Digital Image Processing (3rd Edition)*, pp. 738–744, 2006.

[40] C. Grana, D. Borghesani, P. Santinelli, and R. Cucchiara, "High performance connected components labeling on FPGA," in *Workshop on Database and Expert Systems Applications (DEXA)*, Sep 2010, pp. 221 –225.

[41] D.-Y. Gu, C.-F. Zhu, J. Guo, S.-X. Li, and H.-X. Chang, "Vision-aided UAV navigation using GIS data," in *IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, 2010, pp. 78–82.

[42] S. Gupta, D. Palsetia, M. Ali Patwary, A. Agrawal, and A. Choudhary, "A new parallel algorithm for two-pass connected component labeling," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 1355–1362.

[43] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 413–422.

[44] R. Haralick, "Some neighborhood operators," in *Real-Time Parallel Computing*. Springer, 1981, pp. 11–35.

[45] R. M. Haralick and L. G. Shapiro, "Glossary of computer vision terms," *Pattern Recognition*, vol. 24, no. 1, pp. 69 – 93, 1991.

[46] E. S. I. Harba, "Computer-aided diagnosis for lung diseases based on artificial intelligence: A review to comparison of two-ways: Bp training and pso optimization," *International Journal of Computer Science and Mobile Computing*, 2015.

[47] F. Havet, *Lectures notes on "Combinatorial Optimization - Algorithms for telecommunications "*, Apr 2016, ch. 3. Complexity of algorithms, pp. 29–44.

[48] L. He and Y. Chao, "A very fast algorithm for simultaneously performing connected-component labeling and Euler number computing," *IEEE Transactions on Image Processing*, vol. 24, no. 9, pp. 2725–2735, Sep 2015.

[49] L. He, Y. Chao, and K. Suzuki, "A run-based two-scan labeling algorithm," *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 749–756, May 2008.

[50] L. He, Y. Chao, and K. Suzuki, "A run-based one-and-a-half-scan connected-component labeling algorithm," *International Journal of Pattern Recognition and Artificial intelligence*, vol. 24, no. 04, pp. 557–579, 2010.

[51] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, Sep 2009.

[52] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ch. 3, p. 179.

[53] J. Hopcroft and J. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, 1973.

[54] Q. Hu, G. Qian, and W. L. Nowinski, "Fast connected-component labelling in three-dimensional binary images based on iterative recursion," *Computer Vision and Image Understanding*, vol. 99, no. 3, pp. 414 – 434, 2005.

[55] *Recommendation ITU-R BT.2020-1 Parameter values for ultra-high definition television systems for production and international programme exchange*, International Telecommunication Union, Jun 2014.

[56] M. Isenburg and J. Shewchuk, "Streaming connected component computation for trillion voxel images," in *Workshop on Massive Data Algorithmics*, 2009.

[57] Y. Ito and K. Nakano, "Low-latency connected component labeling using an FPGA," *International Journal of Foundations of Computer Science*, vol. 21, no. 03, pp. 405–425, 2010.

[58] C. Johnston and D. Bailey, "FPGA implementation of a single pass connected components algorithm," in *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on*, Jan 2008, pp. 228 –231.

[59] V. G. Kanas, E. I. Zacharaki, C. Davatzikos, K. N. Sgarbas, and V. Megalooikonomou, "A low cost approach for brain tumor segmentation based on intensity modeling and 3d random walker," *Biomedical Signal Processing and Control*, vol. 22, pp. 19 – 30, 2015.

[60] S. Kaur and S. Kaur, "An efficient approach for number plate extraction from vehicles image under image processing," *International Journal of Computer Science and Information Technologies*, vol. 5, pp. 2954–2959, 2014.

[61] C. G. Keller, T. Dang, H. Fritz, A. Joos, C. Rabe, and D. M. Gavrila, "Active pedestrian safety by automatic braking and evasive steering," *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 4, pp. 1292–1304, 2011.

[62] V. Khanna, P. Gupta, and C. Hwang, "Finding connected components in digital images by aggressive reuse of labels," *Image and Vision Computing*, vol. 20, no. 8, pp. 557 – 568, 2002.

[63] J. Kim and T. Chen, "A VLSI architecture for video-object segmentation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 1, pp. 83–96, 2003.

[64] M. J. Klaiber, D. G. Bailey, S. Ahmed, Y. Baroud, and S. Simon, "©2013. Reprinted, with permission from IEEE. A high-throughput FPGA architecture for parallel connected components analysis based on label reuse," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 302–305.

[65] M. J. Klaiber, D. G. Bailey, Y. O. Baroud, and S. Simon, "©2015 Reprinted, with permission from IEEE. A resource-efficient hardware architecture for connected components analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1–16, 2015.

[66] M. J. Klaiber, D. G. Bailey, Y. O. Baroud, and S. Simon, "A resource-efficient hardware architecture for connected components analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1334–1349, 2015.

[67] M. J. Klaiber, D. G. Bailey, and S. Simon, "A linear single-pass single look-up connected components analysis algorithm," *Submitted to IEEE Transactions on Image Processing*, 2016.

[68] M. J. Klaiber, D. G. Bailey, and S. Simon, "©2016 Springer. A single cycle parallel multi-slice connected components analysis hardware architecture," *Journal of Real-Time Image Processing*, pp. 1–11, 2016.

[69] M. J. Klaiber, J. Guhathakurta, W. Li, Z. Wang, A. Lampa, H. Li, S. Simon, M. Sommerfeld, and U. Fritsching, "A high-speed process monitoring system to detect and analyze filaments and droplet collisions in spray processes in real-time," in *ICLASS 2015, 13th Triennial International Conference on Liquid Atomization and Spray Systems*, Aug 2015, pp. 1–10.

[70] M. J. Klaiber, A. Kleinhans, P. Stähle, V. Gaukel, and S. Simon, "A real-time process analysis system for the pulsation detection and measurement in spray processes," in *ILASS – Europe 2014, 26th Annual Conference on Liquid Atomization and Spray Systems*, Sep 2014, pp. 1–4.

[71] M. J. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, and S. Simon, "A memory-efficient parallel single pass architecture for connected component labeling of streamed images," in *International Conference on Field-Programmable Technology (FPT)*, Dec 2012, pp. 159–165.

[72] M. J. Klaiber, Z. Wang, and S. Simon, *Process-Spray: Functional Particles produced in Spray Processes.* Springer International Publishing AG, Cham, 2016,

ch. A Real-Time Process Analysis System for the Simultaneous Acquisition of Spray Characteristics, pp. 265–305.

[73] D. Knuth, *The Art of Computer Programming: Fundamental algorithms*, ser. The Art of Computer Programming. Addison-Wesley, 2008.

[74] V. S. Kumar, K. Irick, A. A. Maashri, and V. Narayanan, "A scalable bandwidth-aware architecture for connected component labeling," in *VLSI 2010 Annual Symposium*, ser. Lecture Notes in Electrical Engineering, N. Voros, A. Mukherjee, N. Sklavos, K. Masselos, and M. Huebner, Eds. Springer Netherlands, 2011, vol. 105, pp. 133–149.

[75] M. Kuschel and M. Sommerfeld, "Investigation of droplet collisions for solutions with different solids content," *Experiments in Fluids*, vol. 54, no. 2, 2013.

[76] L. Lacassagne and B. Zavidovique, "Light speed labeling: efficient connected component labeling on RISC architectures," *Journal of Real-Time Image Processing*, vol. 6, no. 2, pp. 117–135, Jun 2011.

[77] J. Lasheras, E. Villermaux, and E. Hopfinger, "Break-up and atomization of a round water jet by a high-speed annular air jet," *Journal of Fluid Mechanics*, vol. 357, pp. 351–379, 1998.

[78] M. C. Le, "Lane detection and classification for assistive navigation of the visually impaired," Ph.D. dissertation, School of Civil, Mining and Environmental Engineering - Faculty of Engineering and Information Sciences, University of Wollongong, 2015.

[79] C.-F. Lin, C.-S. Chen, W.-J. Hwang, C.-Y. Chen, C.-H. Hwang, and C.-L. Chang, "Novel outline features for pedestrian detection system with thermal images," *Pattern Recognition*, 2015.

[80] C.-Y. Lin, S.-Y. Li, and T.-H. Tsai, "A scalable parallel hardware architecture for connected component labeling," in *17th IEEE International Conference on Image Processing (ICIP)*, Sep 2010, pp. 3753 –3756.

[81] D. Llorens, A. Marzal, V. Palazon, and J. Vilar, "Car license plates extraction and recognition based on connected components analysis and hmm decoding," in *Pattern Recognition and Image Analysis*, ser. Lecture Notes in Computer Science, J. Marques, N. Perez de la Blanca, and P. Pina, Eds. Springer Berlin Heidelberg, 2005, vol. 3522, pp. 571–578.

[82] R. Lumia, L. Shapiro, and O. Zuniga, "A new connected components algorithm for virtual memory computers," *Computer Vision, Graphics, and Image Processing*, vol. 22, no. 2, pp. 287 – 300, 1983.

[83] N. Ma, D. Bailey, and C. Johnston, "Optimised single pass connected components analysis," in *International Conference on Field Programmable Technology. FPT 2008*, Dec 2008, pp. 185 –192.

[84] R. Madhavan and T. Hong, "Robust detection and recognition of buildings in urban environments from ladar data," in *Information Theory, 2004. ISIT 2004. Proceedings. International Symposium on.* IEEE, 2004, pp. 39–44.

[85] D. Makinson, *Sets, logic and maths for computing.* Springer, 2008.

[86] M. Manohar and H. Ramapriyan, "Connected component labeling of binary images on a mesh connected massively parallel processor," *Computer Vision, Graphics, and Image Processing*, vol. 45, no. 2, pp. 133 – 149, 1989.

[87] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proceedings of the 8th International Conference on Computer Vision*, vol. 2, Jul 2001, pp. 416–423.

[88] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008, ch. 2, p. 52.

[89] U. Meis, M. Oberlander, and W. Ritter, "Reinforcing the reliability of pedestrian detection in far-infrared sensing," in *IEEE Intelligent Vehicles Symposium.* IEEE, 2004, pp. 779–783.

[90] M. Mesbahi and M. Egerstedt, *Graph theoretic methods in multiagent networks*, ser. Princeton series in applied mathematics. Princeton (N.J.): Princeton University Press, 2010.

[91] D. Metcalf, R. Kikinis, C. Guttmann, L. Vaina, and F. Jolesz, "4d connected component labelling applied to quantitative analysis of ms lesion temporal development," in *14th Annual International Conference of the IEEEEngineering in Medicine and Biology Society*, vol. 3, Oct 1992, pp. 945–946.

[92] H. M. Moftah, A. E. Hassanien, and M. Shoman, "3d brain tumor segmentation scheme using k-mean clustering and connected component labeling algorithms," in *2010 10th International Conference on Intelligent Systems Design and Applications (ISDA).* IEEE, 2010, pp. 320–324.

[93] D. Müller, "Fast resource sharing in vlsi routing," Ph.D. dissertation, PhD thesis, University of Bonn, 2009.

[94] F. Nabi, H. Yousefi, and H. Soltanian-Zadeh, "Major temporal arcade separation in angiography images of retina using the hough transform and connected components," in *2015 23rd Iranian Conference on Electrical Engineering (ICEE)*, May 2015, pp. 145–150.

[95] J. Neumann, *The Origins of Digital Computers: Selected Papers.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, ch. First Draft of a Report on the EDVAC, pp. 383–392.

[96] ON Semiconductor, *LUPA3000: 3 MegaPixel High Speed CMOS Sensor*, Jun 2012.

[97] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[98] S. Palaniappan and S. Natarajan, "Parallel realization of single pass connected component analysis on a multi-core architecture," in *2014 International Conference on Communications and Signal Processing (ICCSP)*, Apr 2014, pp. 582–586.

[99] F. N. Paravecino and D. Kaeli, "Accelerated connected component labeling using cuda framework," in *Computer Vision and Graphics*, ser. Lecture Notes in Computer Science, L. Chmielewski, R. Kozera, B.-S. Shin, and K. Wojciechowski, Eds. Springer International Publishing, 2014, vol. 8671, pp. 502–509.

[100] M. Patwary, P. Refsnes, and F. Manne, "Multi-core spanning forest algorithms using the disjoint-set data structure," in *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, May 2012, pp. 827–835.

[101] S. Rathinam, P. Almeida, Z. Kim, S. Jackson, A. Tinka, W. Grossman, and R. Sengupta, "Autonomous searching and tracking of a river using an UAV," in *American Control Conference, 2007. ACC'07.* IEEE, 2007, pp. 359–364.

[102] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM*, vol. 13, pp. 471–494, Oct 1966.

[103] A. Saha, D. D. Roy, T. Alam, and K. Deb, "Automated road lane detection for intelligent vehicles," *Global Journal of Computer Science and Technology*, vol. 12, no. 6, 2012.

[104] A. S. Saif, A. S. Prabuwono, and Z. R. Mahayuddin, "Real time vision based object detection from UAV aerial images: a conceptual framework," in *Intelligent Robotics Systems: Inspiring the NEXT.* Springer, 2013, pp. 265–274.

[105] H. Samet and M. Tamminen, "Efficient component labeling of images of arbitrary dimension represented by linear bintrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 4, pp. 579–586, Jul 1988.

[106] H. Samet and M. Tamminen, "An improved approach to connected component labeling of images," in *International Conference on Computer Vision And Pattern Recognition*, 1986, pp. 312–318.

[107] B. C. Schafer, "Enabling high-level synthesis resource sharing design space exploration in fpgas through automatic internal bitwidth adjustments," 2015.

[108] G. Schewior, H. Flatt, C. Dolar, C. Banz, and H. Blume, "A hardware accelerated configurable asip architecture for embedded real-time video-based driver assistance applications," in *2011 International Conference on Embedded Computer Systems (SAMOS).* IEEE, 2011, pp. 209–216.

[109] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley Professional, 2011.

[110] R. Seidel and M. Sharir, "Top-down analysis of path compression," *SIAM Journal on Computing*, vol. 34, no. 3, pp. 515–525, Mar 2005.

[111] S. M. Selkow, "One-pass complexity of digital picture properties," *Journal of the ACM*, vol. 19, no. 2, pp. 283–295, Apr 1972.

[112] L. G. Shapiro, "Connected component labeling and adjacency graph construction," *Machine Intelligence and Pattern Recognition*, vol. 19, pp. 1–30, 1996.

[113] O. Stava and B. Bedrich, "Connected component labeling in CUDA," *GPU Computing Gems - Emerald Edition*, pp. 569 – 581, 2011.

[114] K. Suzuki, "Computerized detection of lesions in diagnostic images," in *Machine Learning in Radiation Oncology*, I. El Naqa, R. Li, and M. J. Murphy, Eds. Springer International Publishing, 2015, pp. 101–131.

[115] K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," *Computer Vision and Image Understanding*, vol. 89, no. 1, pp. 1–23, Jan 2003.

[116] K. Takahashi and N. Sawada, "Apparatus and method for labeling connected component in a three-dimensional image," Feb 1991, US Patent 4,991,224.

[117] R. Tarjan, *Data Structures and Network Algorithms*, ser. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983.

[118] R. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM*, vol. Volume 31 Issue 2, pp. 245 – 281, 1984.

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

[120] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *Design Test of Computers, IEEE*, vol. 18, no. 4, pp. 36–45, Jul 2001.

[121] G. Teodoro, T. Kurc, J. Kong, L. Cooper, and J. Saltz, "Comparative performance analysis of Intel (r) xeon phi (tm), GPU, and CPU: A case study from microscopy image analysis," in *IEEE 28th International Symposium on Parallel and Distributed Processing*, May 2014, pp. 1063–1072.

[122] S. Thoroddsen, T. Etoh, and K. Takehara, "High-speed imaging of drops and bubbles," *Annu. Rev. Fluid Mech.*, vol. 40, pp. 257–285, 2008.

[123] T.-H. Tsai and C.-H. Chang, "Design for an intelligent surveillance system based on system-on-a-programmable-chip platform," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2015, pp. 2049–2052.

[124] USC-SIPI, "USC-SIPI image database," http://sipi.usc.edu/database/.

[125] K. Wu and E. Otoo, "A simpler proof of the average case complexity of union-find with path compression," *Technical Report LBNL-57527, Lawrence Berkeley National Laboratory*, 2005.

[126] K. Wu, E. Otoo, and K. Suzuki, "Two strategies to speed up connected component labeling algorithms," *Lawrence Berkeley National Laboratory*, 2008.

[127] K. Wu, E. Otoo, and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis and Applications*, vol. 12, no. 2, pp. 117–135, 2009.

[128] Xilinx, Inc., San Jose, CA, USA, *Spartan-6 FPGA Configurable Logic Block User Guide UG384 (v1.1)*, Feb 2010.

[129] Xilinx, Inc., San Jose, CA, USA, "7 series FPGAs memory interface solutions - user guide UG586," Mar 2011.

[130] Xilinx, Inc., San Jose, CA, USA, *Xilinx User Guide - Virtex-6 FPGA Memory Resources UG363 (v1.6)*, Apr 2011.

[131] Xilinx, Inc., San Jose, CA, USA, *AXI Reference Guide, Xilinx UG761 (v14.3)*, Nov 2012.

[132] Xilinx, Inc., San Jose, CA, USA, "Virtex-6 family overview DS150," Jan 2012.

[133] Xilinx, Inc., San Jose, CA, USA, "7 series FPGAs memory resources user guide - UG473 (v1.11)," Nov 2014.

[134] Xilinx, Inc., San Jose, CA, USA, *7 Series FPGAs Overview DS180 (v1.16)*, Oct 2014.

[135] Z. Xu and C. Bao, "Detection for deformed and sheltered circular traffic signs," in *MATEC Web of Conferences*, vol. 22. EDP Sciences, 2015, p. 03021.

[136] Z. Yu, L. Claesen, Y. Pan, A. Motten, Y. Wang, and X. Yan, "Soc processor for real-time object labeling in life camera streams with low line level latency," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, Jun 2014, pp. 345–348.

[137] C. Yuan, Z. Liu, and Y. Zhang, "UAV-based forest fire detection and tracking using image processing techniques," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, Jun 2015, pp. 639–643.

[138] F. Zhao, H. Z. Lu, and Z. yong Zhang, "Real-time single-pass connected components analysis algorithm," *EURASIP J. Image and Video Processing*, vol. 2013, p. 21, 2013.