Journal of Mathematical Imaging and Vision manuscript No.
(will be inserted by the editor)

Comparative Study and Proof of Single-Pass Connected Components Algorithms

Michael J. Klaiber · Donald G. Bailey · Sven Simon

Received: August 15th 2018 / Accepted: June 1st 2019

Abstract Union-find algorithms form the basis of managing sets of equivalent labels within most connected components labelling algorithms. The new class of single-pass connected components analysis (CCA) algorithms (where a feature vector of each component is extracted during processing) are analysed and compared within this context. Such algorithms have been developed for stream processing, using customised hardware architectures. Many of these use an improved union-find algorithm requiring only a single lookup for its find operation. This paper analyses this optimisation, and formally proves that the resulting single lookup connected components algorithm (*SLCCA*) associates each pixel with its correct component when extracting the components' feature vectors. Analysis of the algorithm led to a new double lookup algorithm that reduces the total number of memory accesses, and is a step towards unifying pixel based methods and run based methods. State-of-the-art CCA algorithms are compared in terms of the number of memory accesses, which is a limiting factor for hardware based acceleration, with key implementation trade-offs identified between hardware resources and worst case processing speed.

This work was funded by the German Academic Exchange Service (DAAD) under scholarship id 91510664 and by the German Research Foundation (DFG) under grant SPP 1423. This work is part of the OpenCCA project.

Michael J. Klaiber
Robert Bosch GmbH, Corporate Research Campus, 71272 Renningen, Germany E-mail: michael.klaiber@de.bosch.com

Donald G. Bailey
Department of Mechanical and Electrical Engineering, School of Food and Advanced Technology, Massey University, Palmerston North, New Zealand E-mail: D.G.Bailey@massey.ac.nz

Sven Simon
Institute of Parallel and Distributed Systems, University of Stuttgart, Stuttgart, Germany E-mail: simon@ipvs.uni-stuttgart.de

Keywords Connected Component Analysis · Connected Component Labeling · Feature Extraction · Union-Find · Stream Processing · FPGA · Hardware Architecture

1 Introduction

Connected components analysis (CCA) is a common step in many image processing applications, extracting features such as area or size of arbitrary shaped objects in a binary image. It is based on *connected components labelling* (CCL), which creates a labelled image of the same dimensions as the original image where all pixels of each connected component are assigned a unique label. Most recent CCL algorithms carry out three phases: scan, analyse and relabelling [34, 8, 35, 19, 13]. In the scan phase, a provisional label is assigned to each object pixel. If more than one label is assigned to a single connected component, this relationship is detected and memorised. In the analysis phase one label is chosen to represent each connected component in the labelled image. Most state-of-the-art connected components labelling algorithms perform this analysis using some form of union-find data structure and algorithm [35, 12, 9], although they may not always explicitly mention it by this name [7]. The relabelling phase requires a second pass through the image, and replaces each provisional label by its representative label. As a result, all pixels of a connected component are assigned the same label.

In connected components analysis, a feature vector is derived from each connected component. Since the set of feature vectors is of primary interest, a labelled image is essentially only an auxiliary data structure. If features are extracted during the scan phase then relabelling is redundant and the three processing phases can, therefore, be reduced to only two: scan and analyse. Analysing the image while it is scanned resolves data associations on-the-fly [2] and this is

the principle behind the recently developed class of single-pass CCA algorithms ([1,32]). Such CCA algorithms allow stream processing of the input image and reduce memory requirements [18] since only the labels of the current and the previous row are required for further processing. Previous single-pass CCL algorithms are based on contour tracing [5].

Some CCA and CCL algorithms are adapted and optimised to the instruction sets or memory architectures of the hardware device they are used on [3,11,19]. Many single pass algorithms are motivated by the idea of creating a CCA algorithm from which an efficient customised high-performance architecture can be derived by basic processing and storage elements [18]. This is realised by:

– *Single-pass processing*

For CCA, a labelled image does not need to be stored, therefore, there is no need to maintain, optimise or accelerate the labelled image data structure or its memory accesses when it is processed in only a single pass.

– *Linear processing time*

A necessary condition for real-time processing is that the algorithm complexity is linear in the number of pixels in the image because the binary input image is either read from a memory or received as a pixel stream.

– *One lookup per pixel to determine the representative label*

In many CCL and CCA algorithms, the union-find data structures which represent equivalence relations are mapped to arrays [8,35,13,12,28]. Their union-find algorithms require several lookups per pixel to identify which connected component a pixel is associated with. Most single-pass CCA algorithms reduce this to one lookup per pixel implicitly using a novel, context-based, optimisation of the classical union-find algorithm. The single lookup property is especially important for a dedicated hardware architecture because it enables the system to process the pixel stream at the pixel clock rate.

The contributions of this paper are:

– State-of-the-art CCL and CCA algorithms are analysed in terms of the union-find algorithm (Section 2). In particular, single-pass algorithms are placed within this context, and the corresponding optimised union-find algorithm is identified and analysed.
– Section 3 presents a full algorithmic description of the state-of-the-art *Single Lookup CCA SLCCA* hardware architecture from [18].
– A proof of the correctness of the *SLCCA* algorithm is provided (Section 4). This proves that the single lookup of the optimised union-find algorithm is sufficient for CCA. This is the first formal proof of single-pass CCA algorithms; prior outlines of proof [1] are both informal and incomplete.
– From this, a novel optimised *Double Lookup CCA* algorithm (*DLCCA*) is derived in Section 5, with fewer total lookups required.
– Pixel-based and run-based algorithms are unified by proving that it is only necessary to find the equivalent label of the first pixel in a run when propagating labels from one row to the next, enabling run length encoding to be used for storing the label image.
– The trade-offs between different CCA algorithms are analysed in terms of memory operations and the required resources in Section 6.

2 Union-find in CCL and CCA algorithms

First, what is meant by a connected component is formally defined. The binary input image I identifies object and background pixels on a discrete grid in Cartesian space of width W and height H. Let *imagePos* be the set of all positions in I,

$$imagePos = \{(i,j) : 0 \leq i < W, 0 \leq j < H,\ i,j \in \mathbb{N}\}. \quad (1)$$

Pixels outside the image are assumed to be background.

$$I[p] = \begin{cases} 0, & \forall p \notin imagePos, \\ 1, & \text{if } p=(i,j) \text{ is an object pixel,} \\ 0, & \text{if } p=(i,j) \text{ is not an object pixel.} \end{cases} \quad (2)$$

Two pixels p_1 and p_2, are *adjacent* if

$$\|p_1 - p_2\| = 1. \quad (3)$$

Adjacent object pixels are connected. Here, 8-connectivity is assumed (i.e. using $\| \ \|_{L\infty}$), although the same techniques can be applied for 4-connectivity (using $\| \ \|_{L1}$).

Definition 1 *Connectedness*: Two object pixels in I, p_1 and p_2, belong to the same connected component if there is a path of connected object pixels in I between p_1 and p_2.

This is denoted as $p_1 \longleftrightarrow p_2$, which can be defined recursively as:

$$\begin{cases} I[p_1]{=}I[p_2]{=}1 \wedge \|p_1 - p_2\|{=}1 & \text{or} \\ \exists p_i : I[p_i]{=}1 \wedge p_1 \longleftrightarrow p_i \wedge p_i \longleftrightarrow p_2. \end{cases} \quad (4)$$

The base case holds true if p_1 and p_2, are adjacent object pixels of I. The recursive case holds true if there is an object pixel p_i with a connected path to both p_1 and p_2.

Definition 2 *Connected component:* A maximal set of mutually connected object pixels in I is called a connected component. Each connected component represents a separate image object in I.

2.1 Union-find

Problems which require the manipulation of disjoint sets by carrying out intermixed find and union operations are called *union-find problems* [31]. Within the context of CCL, union-find is used for managing the set of labels associated with a single connected component, and for selecting the representative label for a component.

2.1.1 Graph notation

The most common union-find data structure to represent disjoint sets (distinct components) is a directed forest. Each provisional label assigned to a connected component is represented by a vertex. A directed forest is an acyclic graph where directed edges, referred to as arcs, link pairs of vertices, indicating the relationship between the associated labels. The following graph notation represents the directed forest structure F as a set of vertices $V(F)$ and edges $E(F)$:

$$\begin{aligned} F &= (V,E) \\ V(F) &= \{v_0,\ldots,v_{n-1}\} \\ E(F) &= \{(v_{i_0} \rightarrow v_{j_0}),\ldots,(v_{i_{m-1}} \rightarrow v_{j_{m-1}})\}. \end{aligned} \tag{5}$$

For each edge, $v_i \rightarrow v_j$, v_i is the child vertex, and v_j is the parent. Each vertex has exactly one parent (except for a root vertex which has no parent), with the edge represented by a pointer to its parent. A vertex may have many children; vertices with no children are leaf vertices. A path from vertex v_1 to vertex v_2 is denoted $v_1 \mapsto v_2$, which consists of a sequence of vertices $v_1 \rightarrow v_i \rightarrow \ldots \rightarrow v_2$, where each pair of two consecutive vertices is an arc in $E(F)$.

Definition 3 *Rooted Tree*: A tree (or more formally, a directed rooted tree) T is a sub-graph of F comprising a root vertex v_r and all of its children.

Each vertex belongs to exactly one tree, and there is a path following the edges of T from every vertex in the tree to v_r [24]. Therefore:

$$V(T_{v_r}) = \{v_i : v_i \in V(F) \wedge v_i \mapsto v_r\}. \tag{6}$$

A tree is associated with one connected component in the image. The root vertex of each tree, serves as the representative element for the set. Each tree is referred to by its root v_r (and its associated representative label for the connected component).

Definition 4 *Level of a vertex in a tree:* The level of a vertex v, $level(v)$, is the number of arcs between v and the root, v_r.

The level of the root vertex is therefore 0, and for all other vertices the level is one higher than the level of its parent:

$$level(v) = \begin{cases} 0, & v = v_r \text{ (it is root)}, \\ level(parent(v)) + 1, & \text{otherwise.} \end{cases} \tag{7}$$

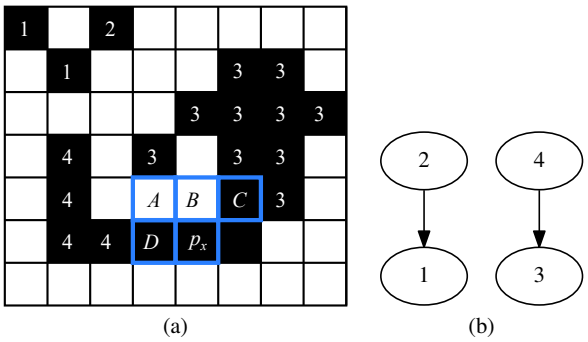


Fig. 1 (a) A label is assigned to each pixel in raster scan order. Also shown is the neighbourhood of a pixel at position $p_x = (x,y)$ and (b) union-find data structure F of the image from (a).

Definition 5 *Height of a tree:* The height of a tree T, $height(T)$, is the maximum level of a vertex in $V(T)$.

$$height(T) = \max\{level(v_i) : v_i \in V(T)\}. \tag{8}$$

Connected components labelling sequentially assigns a label $L[p]$ to each pixel p, with the goal of eventually assigning the same label to all pixels belonging to a single connected component. Since there is a one-to-one relationship between labels and vertices of the forest F, in the discussion here the term *label* is synonymous to a vertex of $V(F)$. Assigning $L[p] := L_{v_p}$ is therefore equivalent to associating p with vertex L_{v_p}. An example image is shown in Figure 1(a) and the corresponding forest derived from this image is shown in Figure 1(b).

2.1.2 Union-find algorithms

Union-find algorithms have three key operations. MAKESET(e) creates a set S_e consisting of a single element e; a UNION(e,f) replaces the sets S_e and S_f by $S_e \cup S_f$ [15]; and a FIND(e) returns the representative element of the set containing e [15].

With a forest structure in the context of CCL, the operations have the following meanings: MAKESET creates a new tree within F and corresponds to assigning a new label to a new connected component; UNION joins two trees into a single tree, corresponding to merging two previously disjoint connected components; and FIND returns the root vertex of the tree which contains a specified vertex, corresponding to finding the representative label of a connected component.

Algorithms 1 to 3 present common variations of union-find which are discussed in the following. These algorithms operate on a directed forest data structure, F, which contains n_F vertices. Adding a vertex to F, changing the parent of a vertex or looking up the parent of a vertex in F are each referred to in the following as one *uf*-instruction (union-find instruction).

Algorithm 1 *QuickFind* based union-find.

```
1: procedure MAKESET(vertex e)
2:     parent[e] := ∅
3: end procedure

4: function FIND(vertex e)
5:     if parent[e] = ∅ then
6:         return e
7:     else
8:         return parent[e]                    ▷ Only a single lookup required
9:     end if
10: end function

11: procedure UNION(vertex e, vertex f)
12:     root_e := FIND(e)
13:     root_f := FIND(f)
14:     for v in V(F) do
15:         if parent[v] = root_e then          ▷ Every vertex of tree added
16:             parent[v] := root_f             ▷ is linked directly to the root
17:         end if
18:     end for
19:     parent[e] := root_f
20: end procedure
```

Algorithm 2 *QuickUnion* based union-find.

```
1: procedure MAKESET(vertex e)
2:     parent[e] := ∅
3: end procedure

4: function FIND(vertex e)
5:     if parent[e] = ∅ then
6:         return e
7:     else
8:         return FIND(parent[e])              ▷ Recursively search for root
9:     end if
10: end function

11: procedure UNION(vertex e, vertex f)
12:     root_e := FIND(e)
13:     root_f := FIND(f)
14:     if root_e ≠ root_f then
15:         parent[root_e] := root_f            ▷ Link one tree to the other
16:     end if
17: end procedure
```

QuickFind based union-find (Algorithm 1) [15] maintains F so that every leaf is directly connected to the root. A FIND, therefore, consists of one *uf*-instruction. A UNION checks which vertices of F belong to the changed rooted tree and changes each of their parents to the new root. A UNION can, therefore, require up to $2n_F$ *uf*-instructions in the worst case.

QuickUnion based union-find (Algorithm 2) [15] combines two trees by making the root of one tree the parent of the root of the other tree. This requires two FINDs for one UNION; which requires up to n_F *uf*-instructions in the worst case [27]. Both *QuickFind* and *QuickUnion* have quadratic run time in the worst case [27].

QuickUnion with path compression (Algorithm 3) [15] joins all vertices which are visited during a FIND directly to the root vertex. Whenever these values are accessed again

Algorithm 3 *QuickUnion with path compression*.

```
1: procedure MAKESET(vertex e)
2:     parent[e] := ∅
3: end procedure

4: function FIND(vertex e)
5:     if parent[e] = ∅ then
6:         return e
7:     else
8:         root := FIND(parent[e])             ▷ Recursively search for root
9:         parent[e] := root                   ▷ Compress path
10:        return root
11:    end if
12: end function

13: procedure UNION(vertex e, vertex f)
14:     root_e := FIND(e)
15:     root_f := FIND(f)
16:     if root_e ≠ root_f then
17:         parent[root_e] := root_f            ▷ Link one tree to the other
18:     end if
19: end procedure
```

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 [30] (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 [30], which is not discussed in this paper.

For connected components labelling or analysis, the sequence of UNION and FIND operations depends on the input image. This can be used to derive a more efficient union-find algorithm for the special case of CCA and CCL of two dimensional images.

2.2 Improved union-find

Single pass CCA requires the label for each pixel to be resolved on-the-fly so that the contribution of the pixel to the feature vector can be allocated to the correct component. For streamed images, the order of operations is determined by the order in which the pixels are scanned, along with the local connectivity.

The pixels of the input image I are streamed or scanned row-wise from the top-left position $(0,0)$ to the bottom-right position $(W-1,H-1)$. A position p_1 preceding another position p_2 in the raster scan order is denoted as $p_1 \prec p_2$.

As the data structures are updated dynamically as the pixels are processed, it is necessary to define the structures that represent the state after processing each pixel. Let p_x be the current pixel. The set *visited* contains all pixels which have already been visited after processing the current pixel:

$$visited = \{p_x\} \cup \{p : p \prec p_x\}. \tag{9}$$

Two pixels p_1, p_2 are connected in image I as scanned so far, if they are connected by a path of adjacent object pixels

Algorithm 4 *Context-based* union-find algorithm.

```
1:  procedure MAKESET(vertex e)
2:      parent[e] := e
3:  end procedure

4:  function FIND(vertex e)
5:      return parent[e]
6:  end function

7:  procedure UNION(vertex e, vertex f)
8:      root_e := FIND(e)
9:      root_f := FIND(f)
10:     if root_e ≺ root_f then                    ▷ Age-balancing heuristic
11:         parent[root_f] := root_e
12:     else
13:         Stack.PUSH(root_f,root_e)              ▷ Caching for path compression
14:         parent[root_e] := root_f
15:     end if
16: end procedure

17: procedure FLATTEN( )
18:     while ¬Stack.empty do                      ▷ Path compression
19:         L_min,L_max := Stack.POP( )
20:         parent[L_max] := FIND(L_min)
21:     end while
22: end procedure
```

in $visited$. Equation (4) can be extended to define $p_1 \underset{visited}{\longleftrightarrow} p_2$ as

$$\begin{cases} p_1, p_2 \in visited \wedge I[p_1]{=}I[p_2]{=}1 \wedge \|p_1 - p_2\| {=} 1 \\ \text{or} \qquad \exists p_i : I[p_i]{=}1 \wedge p_1 \underset{visited}{\longleftrightarrow} p_i \wedge p_i \underset{visited}{\longleftrightarrow} p_2. \end{cases} \tag{10}$$

As F is updated as each pixel is processed, let $F^-_{p_x}$ be the state of F before processing pixel p_x, and F_{p_x} be the resultant state after processing.

Definition 6 *Component segment:* All pixels belonging to the same connected component after processing pixel p_x are a component segment.

Component segments therefore correspond to sets of pixels with labels associated with individual trees in F_{p_x}, and are subsets of the final connected components of I.

Algorithm 4, a *context-based union-find* algorithm, exploits the order of UNION and FIND operations combined with an age-balancing heuristic [8] to achieve linear run time and requires fewer *uf*-instructions in the worst case than *QuickFind* (Algorithm 1), *QuickUnion* (Algorithm 2) or *QuickUnion with path compression* (Algorithm 3). Age-balancing ensures that the label assigned to the earliest pixel of a connected component encountered during a scan is always the root vertex.

Context-based union-find combines the best features of *QuickFind* and *QuickUnion*. Like *QuickFind*, the FIND requires only one *uf*-instruction. The UNION of two vertices makes one root vertex the parent of the other, similar to *QuickUnion* with the addition of age-balancing. In addition to MAKESET, UNION and FIND operations, a fourth operation, FLATTEN, is introduced which performs the equivalent of path compression by making the root vertex the parent of all vertices in a tree.

In *QuickUnion with path compression* (Algorithm 3), path compression is performed within the FIND, processing from the leaves towards the root by following the arcs of $E(F)$ as they are searched by FIND [29]. In contrast, FLATTEN starts at the root vertex and processes towards the leaves. To accelerate this, arcs joining vertices with $level(v) > 1$ that will be encountered in subsequent processing are recorded in a stack during the UNION operations.

Normally, FIND is used to determine the root of a label vertex [30]. Since *context-based union-find*, replaces FIND by a single lookup, it therefore returns only the parent of a vertex. For convenience, every root vertex points to itself, i.e. $parent[v_r] = v_r$. A single lookup is equivalent to a FIND for trees of $height(T) \leq 1$; this will only be the root vertex for vertices of level zero or one.

Definition 7 *Stale label:* A label L_s is called a stale label if a single lookup does not yield the root label.

A necessary condition for the FIND not to return a stale label, is that the CCA algorithm using Algorithm 4 must ensure that FLATTEN is always called before a FIND is applied on a vertex with level larger than one. As outlined in [1], and proven in Section 4, this can be achieved by calling FLATTEN after processing each image row. One situation where $height(T) = 2$ during processing is identified. In *SLCCA* this exception is managed by deferring the second lookup, whereas in *DLCCA* the second lookup is performed explicitly.

2.3 State-of-the-Art CCL and CCA algorithms

Since the introduction of the classic connected components labelling algorithm by Rosenfeld *et al.* [26], CCL has been improved in many aspects. A summary of several properties of (mainly) modern CCL and CCA algorithms is given in Table 1 which compares:

- Number of passes
- Scan mode and scan order
- Worst case run time, and how this was evaluated
- Categorisation of set merging algorithm used

Rosenfeld's classical CCL algorithm [26], is a two-pass algorithm where the first pass uses a binary image as an input and creates a provisionally labelled image. If more than one label is assigned to a connected component, these labels are stored in an equivalence table. These equivalence relations detected during the first scan are resolved at the end of

Table 1 A comparison of properties of representative CCL and CCA algorithms. The set merging algorithm according to the definitions from Section 2.1 are identified. Some algorithms use an optimised variant of the algorithm from Section 2.1, some use path compression.

Method		Abbr.	Passes	Form	Scan	Scan order	Connect	Run time complexity	Set merging algorithm
Rosenfeld, 1966	[26]	Classical	2	CCL	pixel	raster scan	8	N/A	Rosenfeld [26]
Dillencourt, 1992	[8]	*GCCL*	2	CCL	pixel	raster scan	4	Linear (formal proof)	QuickUnion + path compression
Di Stefano, 1999	[7]	*SEL*	2	CCL	pixel	raster scan	4	N/A	QuickFind
Suzuki, 2003	[28]	*SCT*	multi	CCL	pixel	raster scan	8	Linear (experimental)	Iterative connection table
Wu, 2009	[35]	*SAUF*	2	CCL	pixel	raster scan	8	Linear (formal proof)	QuickUnion + path compression
He, 2008	[12]	*RTS*	2	CCL	run	raster scan	8	N/A	Optimised QuickFind
He, 2015	[11]	*HCS*	2	CCL	run	raster scan	8	Linear (experimental)	Equivalent label sets
Lacassagne, 2011	[19]	*LSL*	3	CCL	run	raster scan	8	N/A	QuickUnion
Chang, 2004	[5]	*CT*	1.5	CCL	pixel	contour tracing	8	Linear (formal proof)	None
Grana, 2010	[10]	*Block*	2	CCL	block	modified raster	8	Linear (experimental)	QuickUnion + path compression
Bailey, 2007	[1]	*OSP*	1	CCA	pixel	raster scan	8	Linear (informal)	Context-based union-find
Trein, 2007	[32]	*RLSP*	1	CCA	run	raster scan	8	N/A	QuickUnion
Ma, 2008	[22]	*AR*	1	CCA	pixel	raster scan	8	Linear (informal)	Context-based + relabelling
Klaiber, 2016	[18]	*SLCCA*	1	CCA	pixel	raster scan	8	Linear (formal proof †)	Context-based union-find
Jeong, 2016	[16]	*CAM*	1	CCA	pixel	raster scan	8	Linear (informal)	Direct
proposed		*DLCCA*	1	CCA	pixel	raster scan	8	Linear (formal proof †)	Context-based union-find

† Proof is provided in this paper.

the first pass by iteratively sorting and replacing the entries of the equivalence table until the table contains one entry for each connected component. After this process, each entry of the equivalence table contains all provisional 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 provisionally labelled image are replaced by their representative values from the equivalence table. This assigns the same label to each pixel of a connected component.

Dillencourt *et al.* [8] proposed a general two-pass CCL algorithm (*GCCL*) for different image representations such as 2-D arrays and quad-trees. This algorithm uses *QuickUnion* with path compression extended by an age-balancing heuristic embedded into the UNION operation. Using this property it is formally proven that this algorithm scales linearly with the number of pixels in I.

In [7], Di Stefano *et al.* describe a *simple and efficient connected components labelling (SEL)* algorithm. It requires two passes to label all pixels using an equivalence table as the 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, however, still requires a quadratic number of *uf*-instructions.

In [28], Suzuki *et al.* proposed a multi-pass CCL algorithm using a connection table to store the relations between provisional labels. This algorithm is, therefore, referred to as *scan plus connection table* (*SCT*) CCL algorithm. Previous multi-pass algorithms propagated labels by neighbourhood operations. The algorithm in [28] creates a forest structure stored in the connection table 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 the connection table by one. The algorithm merges disjoint sets, however, it cannot be categorised as a union-find algorithm such as those of Section 2.1. The run time is stated to be linear in the number of pixels which is determined by experimental evaluation. It should be noted, however, that it would be difficult to experimentally distinguish between linear processing, and runtimes proportional to the inverse Ackermann function [30] with small images. Most of the images used for evaluation require four or fewer passes for final labelling [28].

In the two-pass CCL algorithm presented by Wu *et al.* [35], the union-find data structure is represented by an array, therefore, it is referred to as *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 by He *et al.* [12] is a two-pass algorithm which run-length encodes 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 the array for a UNION operation, an additional linked list is maintained for each tree structure in the array containing all the vertices of the tree structure. A UNION on two vertices links the two lists and updates the equivalence table entries of these vertices to the root vertex. This set merging algorithm is referred to as *Equivalent Label Sets* strategy (ELS) [14, 11]. In [13] they optimise their algorithm to only process runs of object pixels in the second pass and in [11] extend the algorithm to

also compute the Euler number. Since, [11] focuses on feature extraction, only the part involved in CCL is considered, and is referred to as *HCS*.

For *light speed labeling* (*LSL*) [19] Lacassagne *et al.* identified memory accesses and conditional statements to be the key issue slowing down CCL on state-of-the-art processors with a RISC architecture. Their algorithm consequently optimises these by distributing the labelling process to three passes, replacing the conditional operations. For set merging, a variation of *QuickUnion* is applied. In [3] *LSL* was identified to require the fewest processing cycles per pixel when carried out on a general-purpose processor.

Chang *et al.* [5] follow 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. During the raster scan, when an unlabelled component is encountered, the border is traced using contour tracing. During this process, control information is stored in the labelled image so that pixels surrounded by already labelled pixels can be labelled when scanning resumes. *Contour tracing* (*CT*) avoids the need for set merging. The authors claim this to be a single-pass algorithm, however random access to the input image during contour tracing effectively means that more than one pass is required. In Table 1 it is, therefore, denoted as a 1.5 pass algorithm. The required random access makes this algorithm less practical for recent processors and dedicated hardware architectures.

Grana *et al.* [10] made the observation that all of the pixels within a 2×2 block will have the same label. They extended the idea of pixel-based labelling to processing a 2×2 block of pixels at a time. Block-based processing operates in a raster scan of 2×2 blocks, hence it is identified as a modified raster scan in Table 1. Like Wu *et al.* [35], a decision tree approach is used to minimise the number of neighbourhood accesses during label assignment. The decision tree is considerably more complex than that for processing single pixels, so Grana *et al.* developed an algorithm to derive the optimal decision tree. The set merging algorithm is *QuickUnion with path compression*, with the trees updated online (whenever a merger occurs).

All of the two-pass CCL algorithms use a set merge algorithm which requires either a minimum of two instructions for a FIND, or have a UNION operation which scales quadratically with the number of labels.

Single-pass CCA algorithms require that the component feature vector be accumulated while determining the connectivity in the first pass. All of the two-pass CCL algorithms use the second pass for relabelling, so could potentially be converted into single pass CCA algorithms by accumulating the feature data during the first pass. However, single-pass CCA algorithms have generally been designed in terms of hardware architectures, optimised for directly processing a video stream. With stream processing, the processing, including feature vector accumulation, is performed in a pipelined manner in hardware.

The *original single-pass* (*OSP*) CCA algorithm by Bailey and Johnston [1] introduced the principle behind the *context-based union-find algorithm*, although it was not identified in terms of union-find. The union-find graph was represented by storing the links in a merger table. The algorithm was based on the one lookup per pixel paradigm, with the use of a stack to optimise the FLATTEN operation. This built on earlier work [2] which introduced the parallel data table and merging the data on-the-fly as regions merged.

Trein *et al.* [32] accelerated the processing by using run-length encoding. Hence it is labelled *RLSP* for *run-length single-pass* CCA. The run-length encoding takes multiple input pixels in parallel, with the runs subsequently processed as one segment (or one overlap between segments) per clock cycle. To manage mergers, they used a pointer from the old label to the new label so that the data from extending an old label could be assigned to the correct component, and the current label assigned to the run. The simple use of pointers in this way corresponds to a *QuickUnion*, which requires a quadratic number of *uf*-instructions in the worst case. Data accumulated for each component is output as soon as it is detected that a run is not extended, enabling the memory for data accumulation to be reused.

Bailey's *OSP* algorithm was optimised by Ma *et al.* [22] to significantly reduce the size of the data and merger tables through *aggressive relabelling* (*AR*). Each row is relabelled beginning with label 1 on the left, requiring translation of labels from one row to the next. The original *context-based union-find* is used, although a second lookup is required for the translation associated with relabelling. The two lookups are pipelined in the hardware implementation. One interesting feature of relabelling is that many mergers are managed by the translation table rather than the merger table, reducing the time required for the FLATTEN operation at the end of each row.

Klaiber *et al.* [18] took a different approach to reduce the memory requirements while retaining the *single lookup* paradigm (*SLCCA*) through label recycling. Augmented labels are introduced to maintain the age-balancing heuristic to ensure correct operation of the *context-based union-find algorithm*. This algorithm is described more fully in Section 3, and proven to have linear run time in Section 4.6. Insights gained from the proof of correctness have led to the optimised *DLCCA*, presented later in this paper.

Jeong *et al.* [16] removed the need for union-find completely by directly replacing all instances of the old label by the new label whenever a merger occurs. This removes the need for an equivalence table or merger table. However, it requires implementing the label memory (or the buffer caching the temporary labels) as *content addressable mem-*

ory (*CAM*). In hardware, the parallel update of the content addressable memory cannot be implemented using the memory blocks on an FPGA; instead Jeong used a multiplexed shift register. Although this method recycles labels immediately after mergers, it does not detect completed objects until the end of the frame, requiring the size of the data table to be the proportional to the image area in the worst case.

3 Algorithmic Description of *SLCCA*

Of all the single-pass algorithms, the *SLCCA* algorithm [18] was chosen for formal proof because it satisfies all of the requirements outlined in the introduction, and it currently represents the state-of-the-art of single-pass CCA algorithms in terms of efficiency of resources and processing speed. The algorithm underlying the hardware architecture of SLCCA is presented in Algorithm 5. Its constituents are explained in Algorithms 5.1 to 5.5 presented at the points in the paper where the corresponding algorithmic background is explained in detail.

The double **for** loop in lines 1 and 2 of Algorithm 5 performs the raster scan through the image. When processing streamed data, these loops are implicit in the order that pixels arrive. The three operations for each pixel can be implemented in one clock cycle each in hardware, and can be pipelined enabling one pixel to be processed per clock cycle. At the end of each row, a FLATTEN is invoked to ensure that the level of any vertices accessed during the following row have $height(T) \leq 1$. In parallel with the pixel processing, when it is detected that a connected component is complete, its associated feature vector is output.

The union-find label graph, F, is realised as a 1-D array, the *merger table*, MT, indexed by the label, L_v, corresponding to each vertex, v. The arcs, $E(F)$, are represented by storing the label of $parent(v)$ in $MT[L_v]$ (each vertex has only one parent). So that lookup of a root vertex, v_r, returns a valid label, every root vertex points to itself, i.e. $MT[L_{v_r}] = L_{v_r}$.

The provisional label assigned to pixel $p_x = (x,y)$ is saved in a label image, $L[p_x]$. In CCA, the labelled image is not required as output, however one row must be maintained for propagation of labels. L is therefore stored as a 1-D array indexed by column, i.e. $L[x]$.

Algorithm 5 SLCCA algorithm.

Input: Binary image I of width W and height H
Output: A feature vector for each connected component in I

```
1: for y = 0 to H − 1 do
2:   for x = 0 to W − 1 do
3:     UpdateNeighbourhood                ▷ Algorithm 5.1
4:     UpdateDataStructures               ▷ Algorithm 5.2
5:     ResolveStaleLabels                 ▷ Algorithm 5.3
6:   end for
7:   Flatten                              ▷ Algorithm 5.4
8: end for
9: ReadFinishedFeatureVectors             ▷ Algorithm 5.5
```

Table 2 Nomenclature used in the following sections.

Abbreviation	Name / description
DT	Data table for accumulating feature vector
F	Forest structure for L
FS	Flatten stack to accelerate FLATTEN operation
FV	Feature vector
H	Image height
I	Source image
IFV	Initial feature vector
$IsRoot$	Flag indicating that a label is a root
L	Labelled image
$LabelFIFO$	FIFO for recycling labels
$LastLine$	Last line the component was updated
MT	Merger table
p_x	The current pixel during processing
SLS	Stale label stack for managing FVs of stale labels
W	Image width

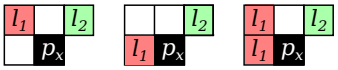


Fig. 2 Merger patterns possible in the labels of neighbourhood L_η.

The abbreviations and names of data structures used in the following are summarised in Table 2.

3.1 Update Neighbourhood

Definition 8 *Neighbourhood:* The neighbourhood η is the set of four positions that have already been processed, adjacent to the current pixel at position p_x (see Figure 1(a)), i.e.

$$\begin{aligned}\eta &= \{(-1,-1),(0,-1),(1,-1),(-1,0)\} + p_x \\ &= \{A,B,C,D\}.\end{aligned} \tag{11}$$

The provisional labels assigned to positions in η are therefore $L[A]$, $L[B]$, $L[C]$ and $L[D]$. The current resolved labels (after a FIND) associated with η are contained in variables L_A, L_B, L_C and L_D. These are realised as registers in the *SLCCA* hardware architecture. The label assigned to the current pixel is denoted L_{p_x}.

Since adjacent object pixels at the positions η will already have the same label as a result of prior processing, a merger of component segments (requiring a UNION of the corresponding trees) can only occur between non-adjacent pixels, i.e. between L_A and L_C, or L_D and L_C [35], as shown in Figure 2. As an optimisation, L_{AorD} is introduced to refer to the label of L_A or L_D, i.e. all mergers consist of the two labels L_{AorD} and L_C.

To move from one window position to the next of the same row, the label values are shifted as given in Algo-

rithm 5.1. The superscript $^-$ denotes the corresponding neighbourhood at the previous position. This shifting requires only the label coming into position C to be looked up with a FIND operation (on line 20).

Algorithm 5.1 UPDATENEIGHBOURHOOD

```
10: if I[A] then                                          ▷ Select L_AorD
11:     L_AorD := L_B^-                                   ▷ Next value of L_A
12: else
13:     L_AorD := L_{p_x}^-                               ▷ Next value of L_D
14: end if
15: if I[B] ∧ I[D] then
16:     L_B := L_{p_x}^-        ▷ Propagate new label into next neighbourhood
17: else
18:     L_B := L_C^-
19: end if
20: L_C := MT[L[C]]   ▷ Single lookup of label on previous row: FIND
```

3.2 Update Data Structures

3.2.1 Label Selection

The set L_η denotes all object pixel labels in the neighbourhood of the current pixel.

$$L_\eta := \{L_{AorD}, L_B, L_C\} \backslash \{0\}. \tag{12}$$

When a pixel is processed, it is assigned a label L_{p_x}. Background pixels are assigned label 0. For object pixels, a label from L_η is propagated to the current pixel where possible.

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 (a MAKESET on $F_{p_x}^-$ creating a new tree). Conceptually, to achieve age-balancing, the new label (called *newLabel* in line 23) is provided by a counter, which is incremented for each new label. The new label operation sets $MT[newLabel] := newLabel$. To more easily detect stale labels, a flag *IsRoot* is associated with each label.

A *label copy* operation propagates the one label in L_η (as determined by the function POSMIN in line 46) to the current position of the labelled image $L[p_x]$.

A merger pattern is detected when L_{AorD} and L_C have different labels and neither is background, i.e. when $(I[A] \vee I[D]) \wedge I[C] \wedge L_{AorD} \neq L_C$. The last term on line 28 is required to manage the case where the label of C is stale (this will be discussed further in section 3.3). 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 merging separate trees in $F_{p_x}^-$. The vertex associated with L_{max} is no longer a root so the flag $IsRoot[L_{max}]$ is cleared.

Definition 9 *Propagating and non-propagating patterns:* A merger pattern is *propagating* if $L_{AorD} \prec L_C$ otherwise it is *non-propagating*.

Algorithm 5.2 UPDATEDATASTRUCTURES

```
21: if I[p] then
22:     if ¬I[A] ∧ ¬I[B] ∧ ¬I[C] ∧ ¬I[D] then      ▷ New label operation
23:         L_{p_x} := newLabel                     ▷ (L_{p_x} ← LabelFIFO)
24:         MT[L_{p_x}] := L_{p_x}                              ▷ MAKESET
25:         IsRoot[L_{p_x}] := true
26:         DT[L_{p_x}] := IFV(p_x)
27:     else                                        ▷ Merger operation: UNION
28:         if (I[A] ∨ I[D]) ∧ I[C] ∧ L_AorD ≠ L_C ∧ L_AorD ≠ L[C] then
29:             if L_AorD ≺ L_C then                          ▷ Propagating merger
30:                 L_min := L_AorD
31:                 L_max := L_C
32:             else                                      ▷ Non-propagating merger
33:                 L_min := L_C
34:                 L_max := L_AorD
35:                 FS.PUSH(L_min, L_max)          ▷ Stack labels for FLATTEN
36:             end if
37:             if IsRoot[L_max] then                  ▷ Prevent multiple recycling
38:                 L_max → LabelFIFO     ▷ Recycle old label after merging
39:             end if
40:             L_{p_x} := L_min
41:             MT[L_max] := L_min
42:             IsRoot[L_max] := false
43:             DT[L_min] := DT[L_min] ∘ DT[L_max] ∘ IFV(p_x)
44:             DT[L_max] := ∅
45:         else                                          ▷ Label copy operation
46:             L_{p_x} := POSMIN(L_AorD, L_B, L_C)
47:             DT[L_{p_x}] := DT[L_{p_x}] ∘ IFV(p_x)
48:         end if
49:         if ¬IsRoot[L_{p_x}] ∧ (L_{p_x} ≠ SLS.head) then
50:             SLS.PUSH(L_{p_x})      ▷ Manage stale labels in building FV
51:         end if
52:     end if
53:     LastLine[L_{p_x}] := y                  ▷ For detecting completed FVs
54: else
55:     L_{p_x} := 0                                      ▷ Background pixel
56: end if
57: L[p_x] := L_{p_x}                          ▷ Save label for processing next row
```

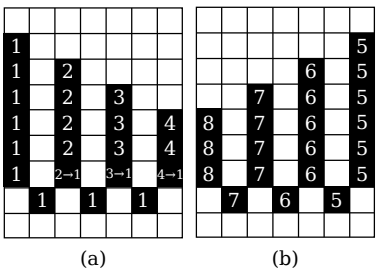


Fig. 3 (a) Propagating, and (b) non-propagating, merger patterns.

Figure 3(a) shows an example of a propagating merger pattern, where the label L_{AorD} is propagated through several mergers. Figure 3(b) is an example of a sequence of non-propagating merger patterns.

A sequence of non-propagating mergers can result in labels having $level(v) \geq 1$ on the next row. These are resolved by flattening the trees at the end of each row.

Table 3 Data structure and combining operator for the feature vectors *area*, *bounding box* and *first order moment*.

Feature	Feature vector	*IFV* $p_x = (x,y)$	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	$\begin{pmatrix} M_{10} \\ M_{01} \end{pmatrix}$	$\begin{pmatrix} x \\ y \end{pmatrix}$	$\begin{pmatrix} M_{10a} + M_{10b} \\ M_{01a} + M_{01b} \end{pmatrix}$

3.2.2 Feature Vector Collection

Definition 10 *Feature vector:* The *feature vector* of an image component is an n-tuple composed of functions of the component's pixel pattern.

Connected components analysis is concerned with deriving the feature vector for each connected component. To accumulate the feature vectors of component segments, a data table, DT, maintains one feature vector for each label. 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 3 presents the data structures, the initial feature vectors, and the combining operation for extracting *area*, *bounding box* and *first-order moment* of connected components.

For a background pixel, nothing needs to be saved in DT. A *new label* operation writes the IFV of the current pixel to $DT[L_{p_x}]$ (line 26). A *label copy* operation combines the current pixel's IFV with the feature vector stored in DT (line 47). A *merger* operation combines the feature vectors of the object labels in L_η with the IFV and stores the result in $DT[L_{min}]$ (line 43); the data table entry at index L_{max} is also invalidated.

3.2.3 Label Reuse

The memory requirements of MT and DT are proportional to the number of labels used, which in the worst case is proportional to the image area [1]. However, at any time, the number of feature vectors updated in one image row is only proportional to the image width [21,17]. Memory requirements can be significantly reduced by recycling labels no longer in use, enabling entries of MT and DT to be reused after a connected component is completed. Rather than use a counter, *newLabel* is obtained from a FIFO, $LabelFIFO$, initialised with the set $LabelFIFO_{init}$, which contains all possible labels [18]:

$$LabelFIFO_{init} = \left\{1, \ldots, \lceil \tfrac{W+5}{2} \rceil \right\}. \tag{13}$$

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

To detect when a connected component has been completed, a tag, *LastLine*, is associated with each label. During the raster scan, whenever a label, L_{p_x}, is updated, its *LastLine* tag is updated with the current image row

$$LastLine[L_{p_x}] := y \quad \text{when } L_{p_x} \neq 0 \tag{14}$$

to reflect that the component is not completed. Labels for which *LastLine* is not updated from one line to the next are detected as completed (as described in section 3.5), enabling the labels of completed components to be recycled and reused.

After every merger operation, label 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 $LabelFIFO$ ensures that it is not assigned to a new connected component within the following image row.

The reuse of labels in this way requires modifying the method used to determine L_{min} and L_{max}. New labels produced by a counter strictly increase in scan order. Therefore, realising the $\prec$-operator as a comparison is sufficient. When reusing labels, the numeric labels are not necessarily assigned to component segments in increasing order. Therefore augmented labels are introduced to realise the functionality of the $\prec$-operator with label reuse.

An augmented label is a two-tuple consisting of the row number $L.rw$ in which the label is first assigned and $L.index$ which is used as an address to access array data structures. For example, $DT[L_{p_x}]$ translates to $DT[L_{p_x}.index]$. The row number rw is used for decisions in merger operations. The evaluation of $L_{AorD} \prec L_C$ (line 29) is thus realised as

$$L_{AorD}.rw \leq L_C.rw. \tag{15}$$

This ensures that L_{min} is always the label created earlier during processing, leading to correct age-balancing behaviour when a merger pattern is detected [1].

When a new label is assigned to a component segment, its index is pulled from the head of $LabelFIFO$, i.e. *newLabel* in Algorithm 5.2 line 23 is realised as

$$\begin{aligned} newLabel.rw &:= y, \\ newLabel.index &\leftarrow LabelFIFO. \end{aligned} \tag{16}$$

3.3 Resolve Stale Labels

A stale label within L_η requires an additional lookup to determine the root vertex. Rather than performing this lookup immediately, *SLCCA* defers this until the root label appears in L_η. If a non-root label is assigned to L_{p_x}, as determined

from the *IsRoot* flag, the feature vectors of the object labels in L_η are combined and stored to data table entry $DT[L_{p_x}]$ for later combination with the feature vector of the root of L_{p_x}. The non-root label is pushed onto the *stale label stack* (*SLS*) (Algorithm 5.2 line 50) until its root appears in L_η. To avoid duplicate entries which lead to increased memory requirements and processing times, a label is only added to *SLS* if it differs from the top entry, *SLS.head*.

When *SLS.head* is equal to $L[C]$, then the lookup to determine L_C will return the label associated with the root vertex. Algorithm 5.3 then combines the feature vector of the stale label with the feature vector of the current component segment, and stores the result in $DT[L_C]$. The data table entry associated with the label popped from the stack is then invalidated.

This enables an on-the-fly processing of feature vectors of reachable stale labels.

Algorithm 5.3 RESOLVESTALELABELS

```
58: if SLS.head = L[C] then
59:     L_stale := SLS.POP()
60:     DT[L_C] := DT[L_C] ∘ DT[L_stale]
61:     DT[L_stale] := ∅
62:     FS.PUSH(L_C, L_stale)                    ▷ Stack for FLATTEN
63: end if
```

3.4 Flattening Trees in F

A prerequisite for Algorithm 4 to produce correct results is that all trees of the forest structure in M are reduced to $height(T) \leq 1$. This can be achieved by using path compression, which is embodied in the FLATTEN operation.

Since minimum labels propagate to the right due to the raster scan by assigning L_{min} to L_{p_x}, the height of a tree in F_{p_x} 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 FS to accelerate flattening (Algorithm 5.2 line 35).

Algorithm 5.4 FLATTEN

```
64: while ¬FS.empty do
65:     L_min, L_max := FS.POP()
66:     MT[L_max] := MT[L_min]                   ▷ FIND on RHS
67: end while
```

At the end of each image row, FLATTEN is invoked as listed in Algorithm 5.4. This pops the arcs off the flatten stack FS, visiting them in reverse order, effectively performing a scan from the root to the leaves in the reverse order that the tree was constructed. The vertex associated with label L_{max} in FS is made the child of the minimum label L_{min} which successively connects each label to the root, flattening the forest structure in M to a height of one.

3.5 Detecting Completed Connected Components

Label reuse requires the data of completed components to be removed from the data table DT so that the label to be recycled. A connected component is *completed* when no further pixels are added to the component in the current row. This cannot be checked until the end of the current row is reached, so in practise, it is checked while the next row is processed. That is a connected component with label l can be detected as completed if $LastLine[l]$ was last updated on row $y-2$ (it was not extended onto the previous row as indicated in Figure 4), i.e.

$$LastLine[l] = y - 2. \tag{17}$$

The data table, DT, is searched for feature vectors of completed 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 recycled for subsequent components by returning the label to the end of the $LabelFIFO$. This process is represented in Algorithm 5.5.

Of course, all remaining objects are completed after processing the last row of the image.

Note that in a hardware implementation, it is unnecessary to store all the bits of y in *LastLine*. Two bits are sufficient to satisfy (17) unambiguously.

Algorithm 5.5 READFINISHEDFEATUREVECTORS

```
68: while ¬ end_of_image do
69:     for l := 1 to ⌈(W+5)/2⌉ do
70:         if DT[l] ≠ ∅ ∧ (LastLine[l] = y − 2) then
71:             Output: DT[l]
72:             DT[l] := ∅
73:             l → LabelFIFO                    ▷ Recycle the label
74:         end if
75:     end for
76: end while
77: for l := 1 to ⌈(W+5)/2⌉ do                   ▷ End of image
78:     if DT[l] ≠ ∅ then
79:         Output: DT[l]
80:         DT[l] := ∅
81:         l → LabelFIFO                        ▷ Recycle the label
82:     end if
83: end for
```

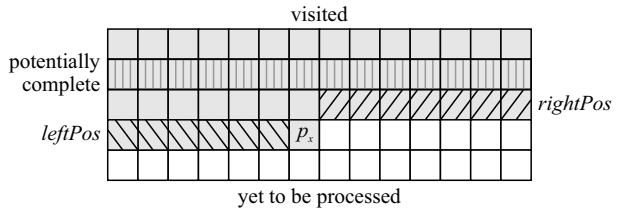


Fig. 4 Visualisation of the positions in the sets *visited*, *rightPos*, *leftPos* in the image.

4 Proof of Correctness of the *SLCCA* Algorithm

In this section, it is shown that the correct feature vector is extracted for each connected component in a binary input image, I, using the algorithm presented in Section 3. In particular, it is shown that replacing FIND of the classical union-find algorithm by a single lookup as in (Algorithm 5.1), the FLATTEN operation of Algorithm 5.4, and the deferred lookup of stale labels in Algorithm 5.3 all result in the extraction of the correct feature vectors for the connected components in I. To do this, a top-down hierarchical proof will be used.

I is only processed once in raster scan order. The current pixel $p_x = (x,y)$ is assigned a label based on the labels in its neighbourhood η. Therefore, only the labels in L of the previous line are used to determine the subsequent labels in the scan process. It is convenient to divide the corresponding positions into two sets relative to p_x, as depicted in Figure 4. The set *leftPos* contains the pixel positions of the current row to the left of p_x:

$$leftPos = \{(i,y) : 0 \leq i < x,\ i \in \mathbb{N}\}, \tag{18}$$

and *rightPos* contains the pixel positions of the previous row to the right of p_x:

$$rightPos = \{(i,y-1) : x < i < W,\ i \in \mathbb{N}\}. \tag{19}$$

Replacing FIND by a single lookup to determine the connected component's root label works correctly for labels associated with vertices of $level(l) \leq 1$. The feature vectors of these labels can be easily accumulated and associated with their connected components.

However, as a result of several mergers, a label can become stale ($level(l) > 1$). To associate such labels correctly with their connected components, additional steps are required. For this, it is convenient to identify the set of vertices (labels) that may be encountered when processing the rest of the current row (before the next call of FLATTEN).

Definition 11 *Reachable vertices:* These are the labels of L in *rightPos* and their parents:

$$V_{reachable} = \{L[p_r] \cup parent(L[p_r]) : p_r \in rightPos\}. \tag{20}$$

4.1 Outline of Correctness Proof

Labels in *leftPos* of $level(l) > 1$ (created by a sequence of non-propagating mergers) are not reachable in the current row, which will be shown in Lemma 12. For these labels, calling FLATTEN at the end of the image row is sufficient as shown in Corollary 13. Therefore, the feature vectors of the associated patterns are correctly determined. The correctness of FLATTEN for compressing the forest structure, F, represented within the merger table, MT, is shown in Theorem 14.

Labels in *rightPos* of $level(l) > 1$ can only be created by a combination of two merger patterns, one in *leftPos* and one in *rightPos* as shown in Lemma 16. In this case, Lemma 18 shows that the root will always be encountered before the end of the image row. Therefore, by storing the stale label on the stale label stack, *SLS*, enables the additional lookup to be deferred, while still associating the accumulated data with the correct connected component (Theorem 20). Finally, it is shown in Theorem 21 that any resulting labels of $level(l) > 1$ are also reduced to level 1 by FLATTEN.

These show that the results of *SLCCA* are correct.

4.2 Non-propagating Mergers

Since each connected component is represented by a tree in F, the arguments given in the following sub-sections of this proof refer to a single connected component.

A non-propagating merger has $L_C \prec L_{AorD}$, so L_{AorD} is made a child of L_C, increasing $level(L_{AorD})$ by 1.

Lemma 12 *After a non-propagating merger, only the root label, L_C, is reachable.*

Proof For L_{AorD} to be reachable, it must be connected to a position in *rightPos* through the pixels that have already been processed. This requires $L_{AorD} \prec L_C$ which contradicts the requirements of a non-propagating merger. Therefore L_{AorD} is not reachable [1]. □

A sequence of two or more non-propagating mergers will result in stale labels in *leftPos* (see Figure 3(b)). However, none of these will appear in the neighbourhood before the end of the image row.

Corollary 13 *Delaying the* FLATTEN *operation until the end of the row will not affect the assigning of correct labels.*

When moving to the start of the next row, *leftPos* becomes *rightPos* so all of the labels in the current row become reachable again. Therefore FLATTEN must reduce the maximum level of a label to 1.

Theorem 14 *The* FLATTEN *operation as described in Algorithm 5.4 results in a forest structure F_p where each rooted tree is of height ≤ 1.*

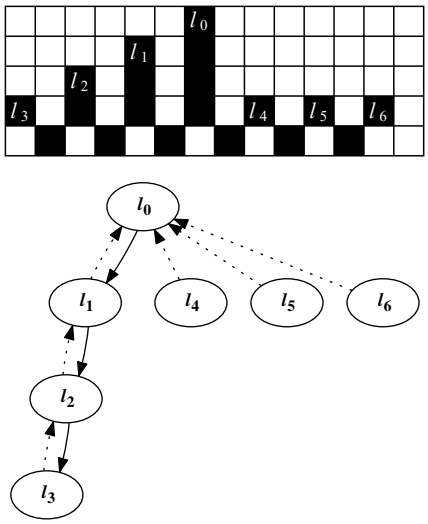


Fig. 5 A sequence of 3 labels followed by 3 propagating merger patterns. Arcs recorded after the last pixel of the current row is processed. The solid arrows represent the arcs pushed onto stack FS which are used for FLATTEN. The dotted arrows represent the arcs stored in merger table M.

Proof Each non-propagating merger pattern increases the level of the vertex associated with label L_{AorD} by one. Since the labels of successive non-propagating mergers are strictly decreasing ($L_C \prec L_{AorD}$), each successive merger grows the tree adding a new root vertex. Therefore revisiting the mergers in reverse order will follow the vertices of a sequence of non-propagating mergers in order from the root to leaf, as illustrated in Figure 5. This is facilitated by pushing the non-propagating mergers onto a stack, FS, as they occur, saving L_{AorD} as L_{max} and L_C as L_{min} respectively. Popping the pair of labels off the stack performs the reverse scan from root back to the leaves. If $level(L_{min}) \leq 1$ then assigning $MT[L_{max}] \coloneqq \text{FIND}(L_{min})$ will make $level(L_{max}) = 1$ for each iteration within Algorithm 5.4.

As a result of the reverse scan, $level(L_{min}) \leq 1$ for non-propagating mergers. It will be shown in Theorem 21 that this is also true for stale labels following a propagating merger (referred to as reachable stale labels).

Consequently, $level(v) \leq 1 \; \forall \; v \in V(F_p)$ before processing the next line. □

Since non-propagating mergers can result in trees requiring the FLATTEN operation, an obvious question is "why not make all mergers propagating mergers?", i.e. to always select L_{AorD} as the root of a merger. As demonstrated in [1], this does not prevent the building of trees of height greater than 1, and since it is not known in advance which vertices will have their level increased, such a scheme would require all mergers to be stacked for checking, not just non-propagating mergers.

4.3 Propagating Mergers

After a propagating merger ($L_{AorD} \prec L_C$), the label $L[C]$ is still reachable, and its level will be increased by 1. If $level(L[C]) > 1$ then $L[C]$ becomes a reachable stale label. In contrast to stale labels resulting from non-propagating mergers, reachable stale labels can appear in the neighbourhood η of p_x before the end of the current image row. The following will investigate how a reachable stale label can be created.

Theorem 15 *The labels resulting from a merger from more than one row previously will be reduced to the root label before the current row.*

Proof FLATTEN will reduce the maximum level of a label to level 1 at the end of a row (theorem 14). In the absence of additional mergers, when scanning the following row $\text{FIND}(L[C])$ will perform the lookup, returning the root label. □

Lemma 16 *A reachable stale label can only be created by a non-propagating merger in rightPos followed by a propagating merger in leftPos.*

Proof The level of a label can only increase as a result of a merger. Therefore at least two mergers are required to make a reachable label stale. From Theorem 15, these mergers must have occurred in the previous W scanned pixels, where W is the image width, i.e. in *leftPos* or *rightPos*. From Lemma 12, the label increased by a non-propagating merger is not reachable, therefore to create a reachable stale label, any mergers in *leftPos* must be propagating mergers. In a sequence of such mergers, each merger links L_C to the root label so successive propagating mergers do not increase the height of the tree (see Figure 3(a)). Similarly, from a propagating merger in *rightPos*, only the root label is reachable. A sequence of one or more non-propagating mergers in *rightPos* will only provide a reachable label of $level(v) \leq 1$ (Theorem 14). Therefore, the only way to get a reachable label with $level > 1$ is through a non-propagating merger in *rightPos* followed by a propagating merger in *leftPos*. □

Two examples of such mergers are shown in Figure 6. On the previous row, labels l_1 and l_2 merge, which makes $level(l_2) = 1$. Then, the propagating merger between l_0 and l_1 in *leftPos* results in $level(l_2) = 2$. Note that this also requires $l_0 \prec l_1$ so that the level of l_2 is increased. The single lookup of l_2 at position p_x results in assigning $L_{p_x} \coloneqq \text{FIND}(l_2) = l_1$, which is a non-root label, rather than the root l_0.

Definition 17 *Bridge patterns:* A bridge pattern is a component segment in which an object label appears more than once in the same image row separated by background pixels.

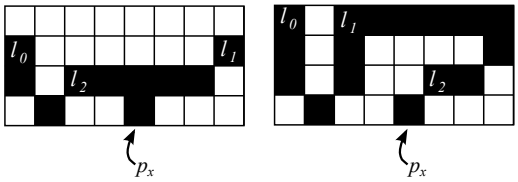


Fig. 6 Two examples of images containing stale label l_2. A non-root label is assigned to L_{p_x}, because a stale label is in the neighbourhood.

A reachable stale label requires a bridge pattern between the merger in *leftPos* and the merger in *rightPos* as is seen in Fig. 6.

4.4 Feature Vector Accumulation of Reachable Stale Labels

To determine the root vertex of reachable stale labels, a maximum of two lookups are necessary, which are distributed to two different positions in the image (in Algorithms 5.1 line 20 and 5.3). It is, therefore, necessary to show that for every possible image pattern which contains a reachable stale label, these two lookups are performed.

Lemma 18 *The appearance of a reachable stale label l_2 in the neighbourhood L_η of the current pixel is always followed by the appearance of $l_1 = parent(l_2)$ in rightPos before the end of the current row.*

Proof From Lemma 16, a reachable stale label in L_η implies a non-propagating merger pattern in *rightPos*, between l_2 and l_1. Since $l_1 \prec l_2$, $l_1 = parent(l_2)$, and l_1 will have been written to the labelled image, L. During the ongoing scan, label l_1 will therefore appear in $L[C]$. □

The stale label stack, *SLS*, is used for caching the stale label while waiting for its parent to appear in the neighbourhood. The stack is necessary, because the stale label may not necessarily be in the neighbourhood when its parent appears.

Lemma 19 *A stack is sufficient for searching for the parent of a stale label.*

Proof Consider the case where a different stale label l_2 is encountered before the parent of the current stale label l_1 is found i.e. $parent(l_1)$ has not yet been encountered. Therefore, there is a path in *visited* between l_1 on the left of l_2 and $parent(l_1)$ on the right of l_2. To become a stale label, l_2 requires an earlier label on each side (Lemma 16): l_{left} and l_{right}, such that $l_{left} \prec l_{right} = parent(l_2) \prec l_2$. Since l_1 appears on both sides of this group, this implies either $l_1 \prec l_{left}$ or $l_1 = l_{left}$. This requires $parent(l_1) \prec parent(l_2)$, therefore $parent(l_1)$ cannot be in between l_2 and $parent(l_2)$. So the stale label l_2 must be resolved before l_1, making a stack appropriate. □

Theorem 20 *The feature vectors of the pixels of a reachable stale label pattern are always associated with their connected component.*

Proof The use of the flag *IsRoot* enables every non-root label assigned to L_{p_x} to be detected when the feature vector is updated in the data table DT. Since *IsRoot* indicates that the label is stale, temporarily buffering the feature vector and recording the label in *SLS* (Algorithm 5.2 line 50), enables the feature vector to later be combined with that of the root label. When the parent of the stale label in encountered (Lemma 18), the data is combined with that of the correct root label. Since the merger in *rightPos* is non-propagating (Lemma 16) the stale label will not appear again beyond the merger point. □

4.5 Flattening Reachable Stale Labels

When reaching the end of a row, there will be no instances of the previously stale label in *leftPos*, because the stale label will have been looked up returning its parent. The parent of the stale label may have been propagated into the label image, L. Since $level(parent(l_{stale})) = 1$ any subsequent non-propagating mergers involving that component will increase the level to 2 or more. Therefore, to ensure that the maximum height after calling FLATTEN is 1, it is also necessary to include in the FLATTEN operation any reachable stale label assigned to p_x (pushed onto *SLS* and resolved before the end of a row).

Theorem 21 *Pushing the reachable stale label onto the flatten stack, FS, when the reachable stale label is resolved is sufficient to correctly flatten reachable stale labels.*

Proof Non-propagating mergers following the event of resolving a reachable stale label are pushed onto the flatten stack after the reachable stale label. Therefore, these non-propagating mergers will be flattened first, ensuring that FLATTEN on the reachable stale label will yield the root label. Any non-nested sequence of non-propagating mergers will similarly be correctly flattened in the reverse order.

Next consider a nested sequence of reachable stale labels, where an inner reachable stale label l_{inner} is created after an earlier reachable stale label l_{outer} is created, but before it is resolved. Since $l_{outer} \prec l_{inner}$ (see the proof of Lemma 19) then if they are part of the same tree, l_{outer} will be closer to the root than l_{inner}. Therefore, l_{outer} must be flattened before l_{inner} requiring it to be pushed onto the flatten stack later. During the processing, l_{outer} is encountered before l_{inner} and will be pushed onto the stale label stack earlier than l_{inner}. When the labels are resolved (Lemma 19), l_{inner} will be resolved first, and consequently be pushed onto the flatten stack earlier than l_{outer} as required. □

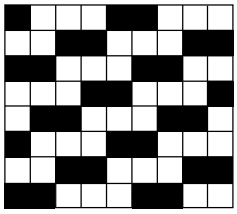


Fig. 7 Stair pattern inducing the maximum number of non-propagating merger operations [1].

4.6 Processing Complexity of SLCCA

New label, label copy and merger operations require constant processing time per pixel. The processing time for these is clearly linear in the number of pixels. Processing stack FS at the end of each image row by FLATTEN is data dependent, but is bounded by the number of non-propagating merger operations. There can be a maximum of $\lfloor \frac{W-1}{2} \rfloor$ non-propagating merger patterns per line of W pixels, so the processing time for this is also linear in the number of pixels.

The worst case pattern with regards to the total number of *uf*-instructions [1] has an average of $\lfloor \frac{W}{5} \rfloor$ merger patterns per row and is shown in Figure 7. This clearly shows that the algorithm from Section 3 scales linearly with the image size.

4.7 Insights Gained

The proof of correctness of *SLCCA* demonstrates that performing a FLATTEN at the end of each row is not only a necessary condition of the improved *context-based union-find*, but it is also a sufficient condition. In particular, this allows the processing of sequences of non-propagating mergers to be deferred until the end of each image row.

It has also identified a limitation of the *OSP* algorithm of Bailey and Johnston [1]. There, reachable stale labels were not considered, and as a result, were not included within the FLATTEN operation at the end of each row, potentially leading to erroneous results in some circumstances. An example of this is using CCA for blob counting, by incrementing the count for each new label operation, and decrementing the count for each merger operation. A reachable stale label can result in an additional merger between already merged components, giving an incorrect count. In *SLCCA*, the stale label stack ensures the data from pixels with stale labels are assigned to the correct feature vector.

5 Optimised *DLCCA* Algorithm

The problem associated with reachable stale labels may be overcome if a second lookup can be performed. However, to determine whether or not a label is a root, it is necessary to either look up the *IsRoot* flag (from the data table DT) or perform a second lookup within the merger table MT.

In this section, it is shown that if two lookups are made within the merger table then it is only necessary to look up the first pixel in a run of pixels. Consequently, the total number of lookups is less than the number of pixels in the image.

5.1 Properties of a Double Lookup

Lemma 22 *A double lookup will always yield the root label.*

Proof A stale label has $level(l) > 1$. Lemma 16 describes the only way of achieving reachable stale labels, which is through a specific combination of two mergers. The maximum height of a reachable tree is 2 levels. Therefore two lookups are always sufficient to reach the root label. □

Theorem 23 *Within a run of consecutive object pixels within rightPos, the root label of all pixels in the run is the same as the root label of the first pixel in the run.*

Proof The labels within a run in *rightPos* can only be different if there has been a merger (if there is no merger, the label simply propagates). After a merger, any adjacent labels have the same root. □

These imply that it is only necessary to look up the first pixel in a run to find the root, and a double lookup is sufficient. The first object pixel in a run will either be followed by another object pixel or background pixel. It is not necessary to look up background pixels, therefore the total number of accesses to the merger table, MT, is less than the total number of pixels in the image, satisfying the single lookup per pixel requirement (on average).

5.2 DLCCA Algorithm

The *DLCCA* algorithm (Algorithm 6) is similar to that for *SLCCA*, with minor changes to UPDATENEIGHBOURHOOD, and UPDATEDATASTRUCTURES. The double lookup means that RESOLVESTALELABELS is no longer required, however FATTEN and READFINISHEDFEATUREVECTORS remain unchanged.

UPDATENEIGHBOURHOOD (Algorithm 6.1) has to be modified to perform the double lookup at the start of a run. During a run (on line 21), the label assigned to L_B on line 15 or 17 is repeated.

UPDATEDATASTRUCTURES (Algorithm 6.2) is simpler than for *SLCCA*. It is no longer necessary to record the *IsRoot* since the double lookup will always return the root. Similarly, the stale label stack, *SLS*, is no longer required. Label assignment, tree flattening, data table update, and completed object detection remain the same.

Algorithm 6 DLCCA algorithm.

Input: Binary image I of width W and height H
Output: A feature vector for each connected component in I

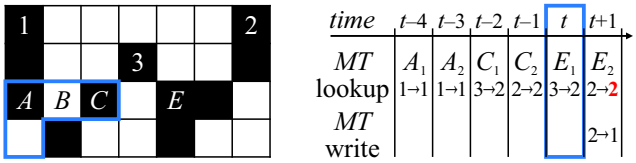
```
1:  for y = 0 to H − 1 do
2:    for x = 0 to W − 1 do
3:      UpdateNeighbourhood                          ▷ Algorithm 6.1
4:      UpdateDataStructures                         ▷ Algorithm 6.2
5:    end for
6:    Flatten                                        ▷ Same as Algorithm 5.4
7:  end for
8:  ReadFinishedFeatureVectors                       ▷ Same as Algorithm 5.5
```

Algorithm 6.1 UpdateNeighbourhood

```
9:  if I[A] then                                     ▷ Select L_AorD
10:   L_AorD := L_B^-                                ▷ Next value of L_A
11: else
12:   L_AorD := L_px^-                               ▷ Next value of L_D
13: end if
14: if I[B] ∧ I[D] then
15:   L_B := L_px^-          ▷ Propagate new label into next neighbourhood
16: else
17:   L_B := L_C^-
18: end if
19: if I[C] then
20:   if I[B] then                                   ▷ Part of a run of consecutive pixels
21:     L_C := L_B                                   ▷ Repeat latest label
22:   else
23:     L_C := MT[MT[L[C]]]                          ▷ Double lookup of start of run
24:   end if
25: else
26:   L_C := 0                                       ▷ Lookup of background is unnecessary
27: end if
```

Algorithm 6.2 UpdateDataStructures

```
28: if I[p] then
29:   if ¬I[A] ∧ ¬I[B] ∧ ¬I[C] ∧ ¬I[D] then          ▷ New label operation
30:     L_px := newLabel                             ▷ (L_px ← LabelFIFO)
31:     MT[L_px] := L_px                             ▷ MakeSet
32:     DT[L_px] := IFV(p_x)
33:   else                                           ▷ Merger operation: Union
34:     if (I[A] ∨ I[D]) ∧ I[C] ∧ L_AorD ≠ L_C ∧ L_AorD ≠ L[C] then
35:       if L_AorD ≺ L_C then                       ▷ Propagating merger
36:         L_min := L_AorD
37:         L_max := L_C
38:       else                                       ▷ Non-propagating merger
39:         L_min := L_C
40:         L_max := L_AorD
41:         FS.push(L_min, L_max)                    ▷ Stack labels for Flatten
42:       end if
43:       L_px := L_min
44:       MT[L_max] := L_min
45:       DT[L_min] := DT[L_min] ∘ DT[L_max] ∘ IFV(p_x)
46:       DT[L_max] := ∅
47:     else                                         ▷ Label copy operation
48:       L_px := posMin(L_AorD, L_B, L_C)
49:       DT[L_px] := DT[L_px] ∘ IFV(p_x)
50:     end if
51:   end if
52:   LastLine[L_px] := y                            ▷ For detecting completed FVs
53: else
54:   L_px := 0                                      ▷ Background pixel
55: end if
56: L[p_x] := L_px                                   ▷ Save label for processing next row
```



Neighbourhood at time t (resulting in a merger)

Fig. 8 DLCCA race condition. The merger in cycle t is written to MT in cycle t+1. The second lookup of E looks up the same label in cycle t+1.

The fact that only the first object pixel in a run needs to be looked up implies that the label image L may be compressed using run length encoding. *DLCCA* therefore unifies pixel-based processing with run-based processing methods, since any subsequent processing can be done on runs. For example, it is similar to He *et al.*'s CCL algorithms [12, 11] which use run length encoding to optimise the second (relabelling) pass, and in [11] for feature extraction (Euler number). However, *DLCCA* differs from Trein's single pass run length algorithm (*RLSP*) [32] in that it still processes the neighbourhood one pixel at a time, whereas *RLSP* processes one overlap between runs in each clock cycle.

In the worst case (with alternating object and background pixels), there is no speed advantage of run-based processing, although it can reduce the processing for more typical images at the expense of more complex processing logic. However, unless the image is streamed in at more than one pixel per clock cycle, then there is limited real advantage.

5.3 Implementation Issues of *DLCCA*

In hardware, each lookup requires a clock cycle. The two lookups can be pipelined, to ensure that the root label is loaded into neighbourhood window. Pipelining the second lookup is possible because the following pixel does not need to be looked up (it is either a background pixel, or part of a run).

However, pipelining can create a race condition where the label being looked up is updated in MT in the same clock cycle as the second lookup, as illustrated in Fig. 8. At time t, the merger between A and C links label 2 to label 1. With pipelining, this is written to the merger table in clock cycle t+1. In parallel with this, pixel E is looked up in the merger table in cycles t and t+1. The first lookup (at t) looks up label 3 and returns 2. The second lookup (at t+1) is of label 2, which would return the old root, label 2, because the merger table has not been updated from merger until the end of t+1. For correct operation, this requires the hardware for MT to perform a write-before-read, or have bypass logic constructed to read the new value being written.

6 Comparison and Discussion

6.1 Evaluation Method

Modern CCA and CCL algorithms are often tailored to the cache hierarchy of general-purpose processors (GPP) [3, 11, 19] 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 algorithms [4]. However, the suitability of a CCA or CCL algorithm to a hardware architecture depends on the interaction of the algorithm with the basic building elements of the technology used and the arrangement of these elements. The freedom to arrange the basic building elements of the hardware device facilitates the use of parallelism and helps to reduce processing or I/O bottlenecks.

The number and speed of lookup operations are crucial for carrying out CCA and CCL, as discussed in the introduction. Unlike in a GPP, hardware architectures realised on an ASIC or FPGA do not have a fixed memory model; the three available memory types *on-chip registers*, *on-chip memory* and *off-chip memory* can be arranged and connected to make the best use of 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. Therefore, *SLCCA* and *DLCCA* are designed to fit completely in on-chip registers and memories, which are limited for current FPGA devices.

Unlike in a cache hierarchy, where the cost of a read or write operation depends on the hierarchy-level, the FPGA on-chip memory model provides random read and write operations at constant cost. Therefore, the total number of memory operations required to process an image provides a good estimate on how suitable a CCA or CCL algorithm is for a hardware architecture.

To compare 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* is considered.

Definition 24 *Memory access instruction:* A memory access instruction (MAI) is a single read access from or a single write access to an indexed data structure.

In particular, the state-of-the-art CCA and CCL algorithms are examined with regards to

- total number of memory accesses,
- degree of parallelism and
- required memory resources

to process a stream of binary pixels.

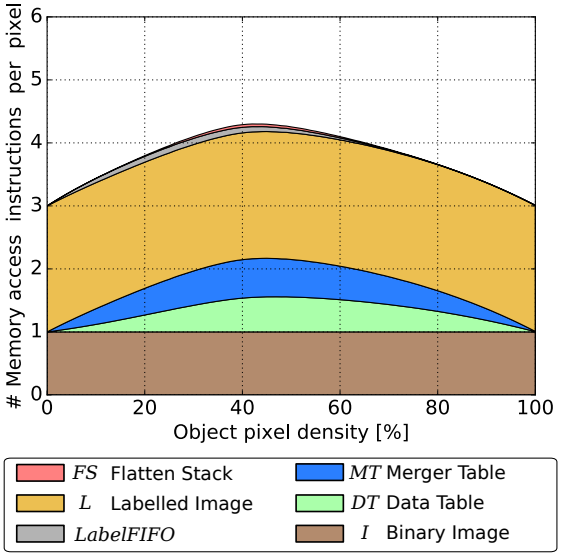


Fig. 9 Memory access instructions (MAIs) on each data structure of *DLCCA* for processing 512×512 images with different object pixel densities.

6.2 MAIs for *DLCCA*

To evaluate and compare the state-of-the-art CCA or CCL algorithms, each algorithm was implemented in C++ or Java. The code was instrumented to count the number of MAIs. Since CCL algorithms are only concerned with outputting a labelled image, feature vector collection was also added to their implementations.

Figure 9 shows the number of MAIs *DLCCA* requires to extract the feature vectors of the components in a random 512×512 pixel image as a function of object pixel density. Each colour in Figure 9 depicts the number of MAIs on one of the data structures; the upper bound shows the total number of MAIs. Read and write accesses to the labelled image, L, are also in parallel using dual-port memory. Although the number of MAIs required for L could be reduced by run-length encoding, these accesses are in parallel to the other data structures, so in practise little would be gained. *DLCCA* is designed to access all data structures in parallel (except for the flatten stack, FS, during FLATTEN at the end of each row). Therefore, the maximum number of MAIs carried out in parallel depend on the maximum number of MAIs on a single data structure plus the MAIs required on FS.

The label and the feature vector associated with the current pixel, L_{p_x}, can be stored in registers, with the other feature vectors stored in on-chip memory. As the label $L[C]$ can be from a different connected component than L_{AorD}, for every pixel, the parent label of $L[C]$ must be looked up. *DLCCA* performs a double lookup (i.e. $M[M[L[C]]]$, in successive clock cycles) to find the root label. Since the root labels of consecutive object pixels are all the same, two look-

ups are always sufficient to determine the label to assign to all pixels of a run. Therefore at most two MAIs on the merger table MT are necessary for a run of consecutive object pixels.

The labels of L_{AorD} and L_B are derived from the labels of $L[C]$ and $L[p_x]$ of the previous position. It is, therefore, sufficient to store them in registers. As the $LabelFIFO$ is only accessed when new label patterns or completed connected components are detected, the number of MAIs on the $LabelFIFO$ is highest around an object pixel density of 40%. *DLCCA* does not store a fully labelled image, therefore L is only accessed for labels assigned to the previously processed image row. For each pixel in the input image I there is one read access on L to retrieve the pixel coming into the local neighbourhood, and one write access to store the provisional label assigned to a pixel. For consecutive object pixels the feature vector associated with label L_{p_x} is cached in registers [18] which optimises the number of MAIs on the data table (DT) having the highest number of MAIs at an object pixel density of around 50%. The number of MAIs on the flatten stack (FS) is highest for images with a stair pattern which have an object pixel density of 40%.

6.3 Evaluation of MAIs

To compare the number of memory access instructions of CCA and CCL algorithms, the following cost metric is applied:

- Successive reads from the same position of a data structure can be buffered in a register and are, therefore, counted as one MAI.
- Successive writes to the same position of a data structure can be cached in a register and are, therefore, counted as one MAI.
- Receiving the input image I as a stream (as in a hardware implementation) is not a memory access instruction per se, i.e. requires zero MAIs. However, 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 a general purpose processor, but indicates the potential speed of a CCA or CCL algorithm when realised as a hardware architecture. In fact, the results of [3] show that *LSL* requires the smallest number of processing cycles per pixel on Intel and ARM processors.

Figure 10 represents the total number of MAIs for extracting the feature vectors of the connected components in a random 512×512 pixel image of by *DLCCA*, *SLCCA* [18], *OSP* [1], *AR* [22], *RLSP* [32], *CAM* [16], *LSL* [19], *HCS* [11], *CT* [5] and Rosenfeld's classical algorithm [26] applying *QuickUnion (RQU)* (see Table 1 for algorithm abbreviations).

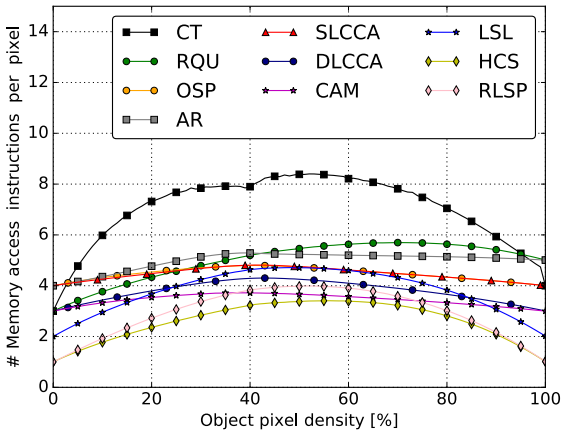


Fig. 10 Comparison of the number of memory access instructions required by CCL and CCA algorithms for processing 512×512 images with different object pixel densities.

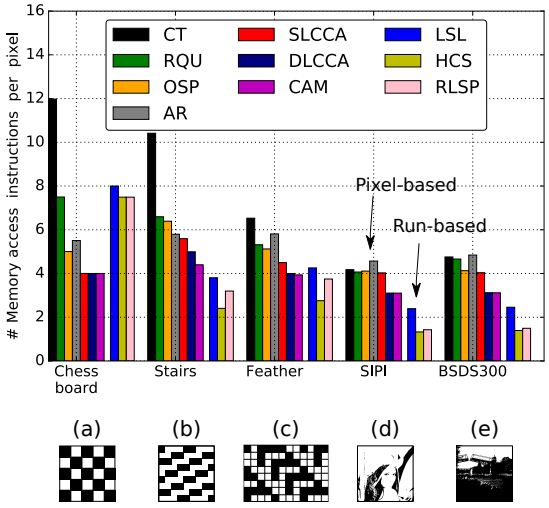


Fig. 11 The number of memory access instructions required for processing worst case images (chess board, stairs, feather pattern) and natural images from the SIPI database [33] and from the Berkley BSDS300 dataset [23]. All images are 512×512 pixels.

RLSP, *HCS* and *LSL* encode and process runs of pixels from the input image, which explains the large difference of MAIs between an image with object pixel density around 50% and an empty or filled image. For the algorithms *HCS*, *CAM*, *RLSP* and *LSL* most MAIs are required when processing random images between 48% and 55% object pixel density.

The number of MAIs for *SLCCA* and *OSP* is almost equal since the basic processing principle is very similar (although *OSP* does not use relabelling). *AR* is also similar to *SLCCA* and *DLCCA*, but requires one additional lookup per pixel for the translation table associated with aggressive relabelling. *DLCCA* is an advancement of *SLCCA* and requires up to 25% fewer MAIs due to caching lookups. The number

of MAIs of *DLCCA* increases with the object pixel density until 43% object pixel density. Above 43% the number of MAIs decrease again.

The bar diagrams in Figure 11(a) through (c) show the number of MAIs required for processing worst case images with chess board pattern, stair pattern and feather pattern [1]. In the scope of the explored algorithms, the chess board pattern with a granularity of one pixel has been shown to require the maximum number of MAIs for *LSL*, *HCS*, *CT*, *RLSP* and *RQU*. Although DeBock and Philips [6] identified a tree pattern as the worst case pattern for *HCS* with respect to the run time, our analysis shows that the chess board pattern requires more MAIs. The stair pattern from Figure 7 requires the maximum number of MAIs for *OSP*, *CAM*, *SLCCA* and *DLCCA*. For *AR*, all of the mergers associated with the stair pattern are managed by relabelling of objects from one row to the next. The merger table (and flatten stack) is only required when both component segments already have a label on the current row, which can only occur with a bridge pattern. This requires an image such as the feather pattern to induce the maximum number of MAIs for *AR* [22].

Figure 11(d) and (e) show the average number of MAIs required for processing the more than 300 natural reference images from the *USC-SIPI* database [33] and the *Berkley BSDS300* dataset [23]. For the comparison, these images are scaled to a size of 512×512 pixels and binarised with a global threshold value determined by Otsu's algorithm [25]. In general, the methods which make use of run-length encoding are able to benefit from such images through their ability to access complete runs of pixels with a single MAI.

To compare the minimal guaranteed processing time, the worst case pattern of each algorithm is used for a comparison. Table 4 lists the total number of MAIs per pixel (the sum of MAIs on all memory structures) for processing the worst case patterns.

CAM requires the fewest number of MAIs due to its content-addressable memory. Every update of the content-addressable memory is counted as a single MAI, even if multiple locations in the memory with the same label are updated. *OSP*, *AR*, *SLCCA* and *DLCCA* have a similar range for the sum of MAIs, as these methods are all based on *OSP*. *AR* requires more MAIs than *OSP* due to the additional translation table. *SLCCA* requires more MAIs than *OSP* because the data table is continuously searched for finished feature vectors. *DLCCA* requires fewer MAIs than *OSP* as it caches labels of continuous runs, whereas *OSP* requires one lookup for each pixel to determine a label's parent.

HCS, *RLSP* and *LSL* perform run-length encoding of the input images before processing the images. The maximum number of MAIs for those algorithms is required when processing the chess board pattern which essentially is a series of runs with a length of a single pixel. Therefore, run-length encoding does not gain an advantage for the worst case.

RQU assigns up to $\lceil W \times H/4 \rceil$ provisional labels when passing the input image the first time. Merging these provisional labels at the end of the image and assigning the final labels to L in a second pass constitute the difference in comparison to *CAM*.

CT traces the contour of each connected component in the input image and, therefore, requires multiple read and write operations which are strictly sequential for each input pixel.

6.4 Evaluation of Parallelism

This sub-section evaluates the parallelism of the examined algorithms. Some algorithms (especially software algorithms) accomplish parallel processing by means of static or dynamic scheduling on a superscalar general-purpose processor. This scheduling cannot be identified from the algorithm's description alone. A good measure for software algorithms is, therefore, the cycles-per-pixel (cpp) measure established by Cabaret and Lacassagne [3], as it relates to an algorithm executed on a specific processor. The parallelism of those algorithms is, therefore, implicit and dependent on the hardware the algorithm is run on.

To compare parallelism for a hardware architecture, such as an FPGA implementation, an explicit description of the hardware architecture and a mapping of the algorithm to it is necessary. This is the case for the following algorithms: *OSP*, *RLSP*, *AR*, *SLCCA*, *DLCCA* and *CAM*. Therefore, only these are discussed with regards to parallel MAIs in Table 4. Other algorithms may well contain pipelined or parallel MAIs, however these are dependent on the particular processor used making their analysis beyond the scope of this evaluation.

AR's additional lookup in the translation table operates in parallel to the other memory structures and, therefore, does not diminish the performance. Similarly, scanning the data table by *SLCCA* and *DLCCA* to detect completed components makes use of a second memory port enabling it to operate in parallel.

CAM requires one parallel MAI per pixel due to the use of a content-addressable memory. Whenever two component segments merge, the provisional label is immediately replaced by the representative label, so no further processing is required.

In contrast, *OSP*, *AR*, *SLCCA* and *DLCCA* require additional MAIs at the end of each row for flattening stale labels from non-propagating mergers. This overhead is proportional to the number of mergers cached on the flatten stack (*FS*). As indicated earlier, *AR* caches fewer mergers as a result of the relabelling process, with a worst case over-

Table 4 Comparison of MAIs per pixel for the worst case patterns.

Algorithm	Worst case pattern		# MAIs [normalised]	
	sum	parallel	sum	parallel
CAM	stairs	all3	4.39	1.00
DLCCA	stairs	stairs	4.99	1.19
OSP	stairs	stairs	5.39	1.19
SLCCA	stairs	stairs	5.59	1.19
AR	feather	feather	5.81	1.06
RLSP	chess	chess	7.49	1.99
HCS	chess	N/A	7.49	N/A
RQU	chess	N/A	7.49	N/A
LSL	chess	N/A	8.00	N/A
CT	chess	N/A	11.97	N/A

head of 1 in 16 (approximately 6.3%). *OSP*, *SLCCA* and *DLCCA* have a worst case overhead of 1 in 5 (20%). However, *OSP* also has an additional sequential overhead of processing completed components at the end of the frame. With *AR*, *SLCCA* and *DLCCA*, label reuse requires identifying completed components on the fly.

This suggests that by making the worst case image more complex, the number of MAIs required for FLATTEN could be reduced. In examining the stair pattern, a new label is assigned to a component segment, only for it to subsequently be merged with an existing segment. If the allocation of a new label could be deferred, then the merger would be unnecessary. Such a change would also require modifying the row buffer, to make it run-length encoded, rather than storing every pixel. This would reduce the worst case overhead from 1 in 5 to 1 in 8 (or 12.5%). For typical images, however, the number of non-propagating mergers is relatively low and the overhead of the FLATTEN operation is negligible (see *FS* in Figure 9).

While *RLSP* can gain on images with large blobs (with long runs), in the worst case, with alternating sequences of individual pixels from the chess board pattern, run-length coding does not help. *RLSP* has serial dependencies within the matching process that cannot easily be pipelined, resulting in an increased parallel MAI score.

6.5 Evaluation of Resources

While *CAM* gives the best performance in terms of MAIs, this comes at a heavy cost in terms of resources. Rather than implementing the provisional label cache, *L*, as a simple memory, the need to replace every instance of an old label during a merger requires the buffer to be implemented using registers. On an FPGA, this is implemented as a shift register with a multiplexer between each stage. While this situation may be improved by implementing the content addressable memory in VLSI, it would still require significantly more logic than a simple memory-based row buffer.

The main limitation of *OSP* is that it must maintain data structures (*DT* and *MT*) that are proportional to the image area [22]. *AR* improved this by relabelling each row from *L* as it is processed, reducing the size of the data structures to the width of the image. *SLCCA* reduced the total on-chip memory required with improved memory management through label recycling (with augmented labels) [18]. This avoids the need for the additional translation table required by *AR*, making it better suited for hardware implementation. *DLCCA* improves the number of MAIs for finding the representative (root) labels (access to *MT*).

The major advantage of *RLSP* over the other run-based algorithms (such as *HCS* or *LSL*) is that it is a true single-pass algorithm. This allows the memory used by finished connected components to be recycled for subsequent ones. The memory requirements are, therefore, proportional to the image width.

7 Conclusions

Single-pass CCA algorithms are a relatively new class of algorithms designed and optimised for processing streamed image data using an embedded or hardware architecture, by extracting component feature vectors directly from the pixel stream. Real-time operation necessitates processing streamed pixel data at one pixel per clock cycle.

This paper provides the first detailed algorithmic perspective of single-pass CCA algorithms identifying and discussing the implicitly used set merging algorithms. These CCA algorithms have been examined and compared with CCL in terms of the union-find algorithm used for managing object mergers. Through this analysis, single-pass CCA algorithms have been unified with more conventional CCL algorithms on an analytical basis.

It has been shown that many single-pass algorithms use a single lookup variant of union-find. This variant is directly based on the order in which UNION and FIND operations are encountered in the context of processing a two-dimensional image as a pixel stream. The FIND is replaced by a single lookup, which is only valid for trees of height less than or equal to one level. This requires an additional FLATTEN operation to reduce the height of labels to at most level 1 (to avoid stale labels) before any FIND (or UNION) is performed on those labels. It is shown that a sufficient condition for this is performing a FLATTEN at the end of each image row.

One of the key paradigms of single-pass algorithms is the online resolution of mergers, enabling the feature vector extracted from each component to be extracted and merged on the fly. The ability to defer the FLATTEN operation to the end of each row significantly relaxes the sequential data dependencies, enabling pipelined stream processing on an FPGA at 1 pixel per clock cycle.

The proof of correctness, and associated analysis, has shown the circumstances that lead to stale labels, where additional processing is required to ensure that data from each pixel is correctly associated its corresponding connected component. Although early work on single-pass connected components analysis [1,22] had identified sequences of non-propagating mergers as one instance of stale label creation, more complex cases involving propagating mergers had not previously been identified. From this insight, it may readily be seen that some algorithms from the literature are either incorrect in their operation (as described) or incomplete in their description, for example [20].

Algorithm analysis has also shown that the issues associated with stale labels can be avoided by using a second lookup. This led to the *DLCCA* algorithm, which performs the two lookups in successive clock cycles at the start of each run of pixels. Pipelining the lookups in this way, and only looking up the first pixel in a run, is shown to reduce the overall number of memory accesses. It also provides a unification between pixel based and run-length based algorithms.

In analysing the operation of single-pass algorithms, there is an obvious trade-off between processing speed and resources. Jeong et al.'s algorithm [16] avoids the overhead of flattening the trees at the end of each row, by immediately removing all references to the old label. This makes it potentially the fastest single pixel per clock cycle algorithm when implemented in hardware. However, the cost of this is replacing the RAM based row buffer with a significantly more resource intensive multiplexed shift-register. Ma et al.'s aggressive relabelling [22] incurs a small overhead at the end of each row for the FLATTEN, at the cost of additional resources for the translation table (and an additional lookup, although this can be pipelined). Klaiber et al. [18] reduce this (and the associated memory required) at the cost of a higher FLATTEN overhead.

It is hoped that by outlining the necessary and sufficient conditions for correct operation, as well as the comparison of the strength and weaknesses of existing CCA and CCL algorithms this analysis would inspire further attempts at optimising the class of single-pass CCA algorithms.

Acknowledgements The authors would like to thank Prof. Stephen Marsland of Victoria University, Wellington for his helpful comments and suggestions on an early draft of this paper. We would also like to thank the reviewers for their valuable comments and suggestions which have substantially improved the content of the paper.

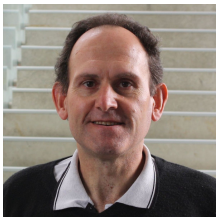
Code A functional Java implementation of the algorithm is published under the following URL:
`http://crisp.massey.ac.nz/code/DLCCA.zip`

References

1. Bailey, D., Johnston, C.: Single pass connected components analysis. In: Image and Vision Computing New Zealand, pp. 282–287 (2007)
2. Bailey, D.G.: Raster based region growing. In: 6th New Zealand Image Processing Workshop, pp. 21–26 (1991)
3. Cabaret, L., Lacassagne, L.: What is the worlds fastest connected component labeling algorithm? In: International Workshop on Signal Processing Systems, pp. 1–6 (2014). DOI 10.1109/SiPS.2014.6986069
4. Cabaret, L., Lacassagne, L., Oudni, L.: A review of worlds fastest connected component labeling algorithms: Speed and energy estimation. In: Conference on Design & Architectures for Signal & Image Processing (DASIP), pp. 1–6 (2014). DOI 10.1109/DASIP.2014.7115641
5. Chang, F., Chen, C.J., Lu, C.J.: A linear-time component-labeling algorithm using contour tracing technique. Computer Vision and Image Understanding **93**(2), 206–220 (2004). DOI 10.1016/j.cviu.2003.09.002
6. De Bock, J., Philips, W.: Fast and memory efficient 2-D connected components using linked lists of line segments. IEEE Trans. on Image Processing **19**(12), 3222–3231 (2010). DOI 10.1109/TIP.2010.2052826
7. Di Stefano, L., Bulgarelli, A.: A simple and efficient connected components labeling algorithm. In: International Conference on Image Analysis and Processing, pp. 322–327 (1999). DOI 10.1109/ICIAP.1999.797615
8. Dillencourt, M.B., Samet, H., Tamminen, M.: A general approach to connected-component labeling for arbitrary image representations. J. ACM **39**(2), 253–280 (1992). DOI 10.1145/128749.128750
9. Fiorio, C., Gustedt, J.: Two linear time union-find strategies for image processing. Theoretical Computer Science **154**(2), 165–181 (1996). DOI 10.1016/0304-3975(94)00262-2
10. Grana, C., Borghesani, D., Cucchiara, R.: Optimized block-based connected components labeling with decision trees. IEEE Trans. on Image Processing **19**(6), 1596–1609 (2010). DOI 10.1109/TIP.2010.2044963
11. He, L., Chao, Y.: A very fast algorithm for simultaneously performing connected-component labeling and Euler number computing. IEEE Trans. on Image Processing **24**(9), 2725–2735 (2015). DOI 10.1109/TIP.2015.2425540
12. He, L., Chao, Y., Suzuki, K.: A run-based two-scan labeling algorithm. IEEE Trans. on Image Processing **17**(5), 749–756 (2008). DOI 10.1109/TIP.2008.919369
13. He, L., Chao, Y., Suzuki, K.: A run-based one-and-a-half-scan connected-component labeling algorithm. International Journal of Pattern Recognition and Artificial intelligence **24**(4), 557–579 (2010). DOI 10.1142/S0218001410008032
14. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-component labeling. Pattern Recognition **42**(9), 1977–1987 (2009). DOI 10.1016/j.patcog.2008.10.013
15. Hopcroft, J., Ullman, J.: Set merging algorithms. SIAM Journal on Computing **2**(4), 294–303 (1973). DOI 10.1137/0202024
16. Jeong, J.w., Lee, G.b., Lee, M.j., Kim, J.G.: A single-pass connected component labeler without label merging period. Journal of Signal Processing Systems **84**(2), 211–223 (2016). DOI 10.1007/s11265-015-1048-7
17. Khanna, V., Gupta, P., Hwang, C.: Finding connected components in digital images by aggressive reuse of labels. Image and Vision Computing **20**(8), 557–568 (2002). DOI 10.1016/S0262-8856(02)00044-6
18. Klaiber, M.J., Bailey, D.G., Baroud, Y.O., Simon, S.: A resource-efficient hardware architecture for connected components analysis. IEEE Trans. on Circuits and Systems for Video Technology **26**(7), 1334–1349 (2016). DOI 10.1109/TCSVT.2015.2450371

19. Lacassagne, L., Zavidovique, B.: Light speed labeling: efficient connected component labeling on RISC architectures. Journal on Real-Time Image Processing **6**(2), 117–135 (2011). DOI 10.1007/s11554-009-0134-0
20. Ling, L., Chen, Z., Li, S., Zhang, X.: FPGA-based connected components analysis algorithm without equivalence-tables. In: Y. Huang, H. Wu, H. Liu, Z. Yin (eds.) 10th International Conference on Intelligent Robotics and Applications (ICIRA 2017), vol. LNAI 10463, pp. 543–553. Springer International Publishing (2017). DOI 10.1007/978-3-319-65292-4_47
21. Lumia, R., Shapiro, L., Zuniga, O.: A new connected components algorithm for virtual memory computers. Computer Vision, Graphics, and Image Processing **22**(2), 287–300 (1983). DOI 10.1016/0734-189X(83)90071-3
22. Ma, N., Bailey, D., Johnston, C.: Optimised single pass connected components analysis. In: International Conference on Field Programmable Technology, pp. 185–192 (2008). DOI 10.1109/FPT.2008.4762382
23. Martin, D., Fowlkes, C., Tal, D., Malik, J.: A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In: 8th International Conference on Computer Vision, vol. 2, pp. 416–423 (2001). DOI 10.1109/ICCV.2001.937655
24. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox, chap. 2, p. 52. Springer (2008)
25. Otsu, N.: A threshold selection method from gray-level histograms. IEEE Trans. on Systems, Man and Cybernetics **9**(1), 62–66 (1979). DOI 10.1109/TSMC.1979.4310076
26. Rosenfeld, A., Pfaltz, J.L.: Sequential operations in digital picture processing. J. ACM **13**(4), 471–494 (1966). DOI 10.1145/321356.321357
27. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley Professional; 4th edition (2011)
28. Suzuki, K., Horiba, I., Sugie, N.: Linear-time connected-component labeling based on sequential local operations. Computer Vision and Image Understanding **89**(1), 1–23 (2003). DOI 10.1016/S1077-3142(02)00030-9
29. Tarjan, R.: Data Structures and Network Algorithms. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (1983)
30. Tarjan, R., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984). DOI 10.1145/62.2160
31. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975). DOI 10.1145/321879.321884
32. Trein, J., Schwarzbacher, A.T., Hoppe, B., Noffz, K.H., Trenschel, T.: Development of a FPGA based real-time blob analysis circuit. In: Irish Signals and Systems Conference, pp. 121–126 (2007)
33. USC-SIPI: USC-SIPI image database. http://sipi.usc.edu/database/
34. Wu, K., Otoo, E., Shoshani, A.: Optimizing connected component labeling algorithms. In: Medical Imaging, vol. SPIE 5747, pp. 1965–1976. International Society for Optics and Photonics (2005). DOI 10.1117/12.596105
35. Wu, K., Otoo, E., Suzuki, K.: Optimizing two-pass connected-component labeling algorithms. Pattern Analysis and Applications **12**(2), 117–135 (2009). DOI 10.1007/s10044-008-0109-y

Michael J. Klaiber received the diploma degree (Dipl.-Ing.) in Electrical Engineering and Information Technology in 2011, and the PhD degree in Computer Science in 2016 from the University of Stuttgart, Germany. From 2016 to 2018, he worked as a Logic Design Engineer for IBM contributing to Mainframe and Power processors. Since October 2018 he has been working for Robert Bosch Corporate Research in Renningen, Germany. His research interests include image processing, computer vision, deep learning, machine learning, computer engineering and hardware architectures.



Donald G. Bailey has BE(Hons) (1982) and PhD (1985) degrees in Electrical and Electronic Engineering from University of Canterbury, New Zealand. He is a Senior Member of IEEE. He is currently Professor of Imaging Systems at Massey University, and is leader of the Centre for Research in Image and Signal Processing. Donald has spent 30 years applying image processing technology to a range of industrial, machine vision and robot vision applications. For the last 17 years one are of particular research focus has been exploring aspects using FPGAs for implementing and accelerating image processing algorithms. He is the author of many publications in this field, including the book Design for Embedded Image Processing on FPGAs, published by Wiley / IEEE Press.

Sven Simon received the diploma from RWTH Aachen (1992) and the Ph.D. from Technische Universitt Mnchen (1996) both in Electrical Engineering. In 1996, he joined Siemens AG and Infineon Technologies AG in 1998 focusing on hardware architectures and digital signal processing. In 1998 he became project manager and was nominated for the Infineon Inventors Award in 2000. In 2001, he became professor at Hochschule Bremen, Germany, heading a research group for hardware architectures and sensor systems. In 2007, he became full professor and head of the Parallel Systems Department at the Institute of Parallel and Distributed Systems of the University of Stuttgart. His research interests include parallel algorithms, hardware architectures and sensor systems. He has numerous publications as well as a number of patents.