A State-of-the-Art Review With Code About Connected Components Labeling on GPUs

Federico Bolelli, *Member, IEEE*, Stefano Allegretti, Luca Lumetti, and Costantino Grana, *Member, IEEE*

***Abstract*—This article is about Connected Components Labeling (CCL) algorithms developed for GPU accelerators. The task itself is employed in many modern image-processing pipelines and represents a fundamental step in different scenarios, whenever object recognition is required. For this reason, a strong effort in the development of many different proposals devoted to improving algorithm performance using different kinds of hardware accelerators has been made. This article focuses on GPU-based algorithmic solutions published in the last two decades, highlighting their distinctive traits and the improvements they leverage. The state-of-the-art review proposed is equipped with the source code, which allows to straightforwardly reproduce all the algorithms in different experimental settings. A comprehensive evaluation on multiple environments is also provided, including different operating systems, compilers, and GPUs. Our assessments are performed by means of several tests, including real-case images and synthetically generated ones, highlighting the strengths and weaknesses of each proposal. Overall, the experimental results revealed that block-based oriented algorithms outperform all the other algorithmic solutions on both 2D images and 3D volumes, regardless of the selected environment.**

***Index Terms*—Parallel image processing, connected components labeling, state-of-the-art review, GPU, CUDA.**

I. Introduction

CONNECTED Components Labeling (CCL) is an image processing algorithm that plays a central role in machine vision, whenever object recognition and measurement are required. The task itself can be easily defined as the procedure of assigning to each pixel of a connected component (object) a unique identifier, typically an integer number. Starting from a binary input, a CCL algorithm generates the output symbolic image where pixels belonging to an object are given the same label. Once computed, the labeled image can be used to extract object(s) and further calculate its features such as area, perimeter, circularity, centroids, bounding boxes, etc. The process of extracting features is usually referred to as Connected Components Analysis, or CCA in short, and has been addressed by different authors in the past, both for CPU and GPU architectures [1], [2].

Received 24 October 2023; revised 19 March 2024; accepted 18 July 2024. Date of publication 29 July 2024; date of current version 23 February 2026. This work was supported in part by the Department of Engineering "Enzo Ferrari" of the University of Modena and Reggio Emilia under Grant FARD-2023. Recommended for acceptance by Y. Zhang. *(Corresponding author: Federico Bolelli.)*

The authors are with Dipartimento di Ingegneria "Enzo Ferrari", Università degli Studi di Modena e Reggio Emilia, 41121 Modena, Italy (e-mail: federico.bolelli@unimore.it; stefano.allegretti@unimore.it; luca.lumetti@unimore.it; costantino.grana@unimore.it).

Digital Object Identifier 10.1109/TPDS.2024.3434357

Multiple modern computer vision pipelines employ CCL as a pre- or post-processing algorithm when tracking [3], [4], segmenting [5], [6], [7], localizing [8], generating [9], or counting [10] objects inside images and videos, in different medical imaging applications ranging from skin lesion segmentation and classification [11], [12], [13] to Whole-Slide Image (WSI) analysis [14], [15], for document restoration [16], [17], and graph analysis [18], [19]. All the aforementioned applications benefit from efficient CCL implementations and this is why the algorithmic proposals of the last decades have focused on performance optimization [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30].

It is worth mentioning that given the input and selected pixel connectivity, the final result of CCL and CCA is uniquely defined and the main differences among existing algorithmic solutions are the execution time and memory requirements. Moreover, given that the object geometry and shape may be extremely complex, the task is known to be more time-consuming than any other binary image operator, e.g., filtering, thresholding, interpolation, edge detection [31]. Most importantly, the labeling procedure cannot be completed by mere parallel local operations, requiring intrinsically sequential procedures [32].

However, the fast advance of Graphics Processing Units (GPUs) in the last years encouraged the development of algorithms specifically designed to work in data parallel environments. Indeed, along with sequential solutions [31], [33], [34], many algorithms exploiting hardware parallelism have been proposed [1], [25], [26], [35], [36], [37], [38], [39].

Most of the improvements introduced for sequential implementation have been sooner or later ported to GPU algorithms, or inspired for additional optimizations on such architectures. An example is the forest-based implementation of the *union-find* data structure, originally applied to CCL in [40] to solve equivalences between labels, and later employed for implementing GPU algorithms [36], [41]. The *block-based* approach is another noticeable example: introduced in [42] and subsequently employed in different CPU-based proposals [27], [43], [44] has been recently ported to GPU-based algorithms [36], [45], [46].

This article aims to provide a state-of-the-art review of GPU-based algorithms, highlighting innovative and common traits of each proposal, and discussing their strengths and weaknesses. Whenever possible, the union-find technique will be used as the common denominator. All the algorithms are evaluated under different perspectives (e.g., total and step-level execution time using both real and synthetically generated images and memory

usage) using different experimental settings (e.g., operating systems, compilers, and GPUs). Moreover, the implementations used for the evaluation are publicly released in the YACCLAB benchmark [47], allowing other research to reproduce the experiments and verify our claims. All the code is taken from the original publication or implemented carefully following the description provided in the corresponding paper. We strongly believe in reproducible research and our effort is heading in this direction, letting everybody compare with existing solutions in the fairest possible way.

The rest of the article is organized as follows. Section II introduces a formal definition of the task, defining all the concepts that will be later employed in the discussion, including the union-find data structure. All the major GPU-based CCL proposals will be categorized and later described in Section III. To better understand algorithms' details, the pseudo-code is often included. In Section IV, the benchmark used for the evaluation is detailed providing a specific description about the datasets and evaluation criteria employed. Performance analysis and discussion is then provided in Section V. Finally, Section VI draws conclusive remarks and future research directions.

II. Problem Formalization

This Section formalizes all the core elements related to the connected components labeling task, laying out the foundations for the rest of the discussion.

Let us start from the concept of image. A binary image I_d is a function defined over a d-dimensional rectangular lattice $\mathcal{L}_d$, where $I_d(x)$ is the value of pixel (or voxel) $x \in \mathcal{L}_d$, with x identified by its coordinates in the lattice, $(x_1, \ldots x_d)$. We are specifically interested in 2 and 3-dimensional images, i.e., volumes.

The distance between pixels x and y can be defined in two different ways, D_1 and D_∞, also known as Manhattan and chessboard or Chebyshev distance, respectively:

$$D_1(x,y) = \sum_{i=1}^{d} |x_i - y_i| \tag{1}$$

$$D_\infty(x,y) = \max_{i=1..d} |x_i - y_i| \tag{2}$$

Each distance has an associated neighborhood:

$$V_1^i(x) = \{y \mid D_1(x,y) \leq i\} \tag{3}$$

$$V_\infty^i(x) = \{y \mid D_\infty(x,y) \leq i\} \tag{4}$$

We can now use these generic definitions to describe the different connectivities employed for 2D and 3D. In 2D images, there are two possibilities:

- *4-connectivity*. Two pixels of the lattice, p and q, are said to be 4-connected if they share a side. Formally, $\mathcal{N}_4(p) = V_1^1(p)$. We can say that p and q are 4-connected if $q \in \mathcal{N}_4(p)$, which implies $p \in \mathcal{N}_4(q)$. The name of the connectivity is intuitively derived from the number of sides a square has: four (Fig. 2(a)).
- *8-connectivity*. If pixels are considered connected also when they share only a vertex, four more neighbors appear, obtaining the 8-connectivity. In other words, two pixels are said to be 8-connected if they share at least one vertex. We can define $\mathcal{N}_8(p) = V_\infty^1(p)$ and say that p and q are 8-connected if $q \in \mathcal{N}_8(p)$, implying that $p \in \mathcal{N}_8(q)$ (Fig. 2(b)).

Similar definitions can be provided for 3D volumes. In this context, pixels are often called voxels, and can be visualized as cubes; since two of them can share a face (with at most 6 different cubes), an edge (with at most 12 cubes), or a vertex (with at most 8 cubes), three different kinds of connectivity can be defined, i.e., 6-, 18-, and 26-connectivity, whose formal definition is reported below:

$$\mathcal{N}_6(v) = V_1^1(v) \tag{5}$$

$$\mathcal{N}_{18}(v) = V_1^2(v) \cap V_\infty^1(v) \tag{6}$$

$$\mathcal{N}_{26}(v) = V_\infty^1(v) \tag{7}$$

If we want to define a mapping between 2D and 3D worlds, we can say that 4- and 8-connectivity on 2D images correspond respectively to 6- and 26-connectivity on 3D volumes. To simplify the discussion and ease the reading, in the following the word *image* is used to refer to 3D volumes also. The generic symbols $\mathcal{L}$ and I will be used, when suitable, for both dimensionalities.

As mentioned, the CCL task aims at identifying objects within a binary image, so it is mandatory to distinguish meaningful regions from the rest of the image. The formers are usually called *foreground*, $\mathcal{F}$, while the latter is identified as *background*, $\mathcal{B}$. It is a common convention to assign value 1 (or 255) to foreground pixels, and value 0 to background ones:

$$\mathcal{F} = \{p \in \mathcal{L} \,|\, I(p) = 1\} \tag{8}$$

$$\mathcal{B} = \{p \in \mathcal{L} \,|\, I(p) = 0\} \tag{9}$$

Connected components labeling aims at identifying disjoint objects composed of foreground pixels. Given a neighborhood definition $\mathcal{N}$ and two foreground pixels $p, q \in \mathcal{F}$, $\diamond$ is the *connectivity* relation defined as:

$$p \diamond q \Leftrightarrow \exists \{s_i \in \mathcal{F} \,|\, s_1 = p, s_{n+1} = q, s_{i+1} \in \mathcal{N}(s_i), i = 1, \ldots, n\} \tag{10}$$

Two pixels p and q are said to be *connected* when condition $p \diamond q$ is satisfied, meaning that a path of connected pixels from p to q exists or, in other words, you can move from p to q crossing only foreground pixels connected two by two. The proposed formalism does not consider background pixels, which are excluded from the concept of connectivity. $\diamond$ is an equivalence relation since *reflexivity*, *symmetry* and *transitivity* are satisfied by pixel connectivity. For this reason, we can define $[p]$, the equivalence class of a pixel p, as:

$$[p] = \{q \in \mathcal{F} \,|\, p \diamond q\} \tag{11}$$

Equivalence classes based on $\diamond$ relationship are called *Connected Components* (CCs). Every pair of connected components $[p]$ and $[q]$ is either equal or disjoint, meaning that the set of all connected components is a partition of $\mathcal{F}$.

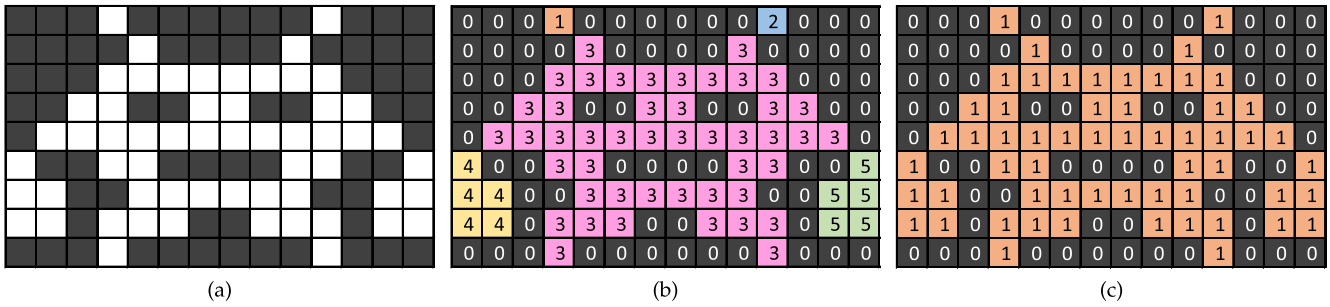


Fig. 1. Example of the labeling procedure applied on the input binary image (a), which depicts a fantasy character inspired by Space Invaders. (b) and (c) are the symbolic images produced by the labeling procedure when using 4- and 8-connectivity respectively.

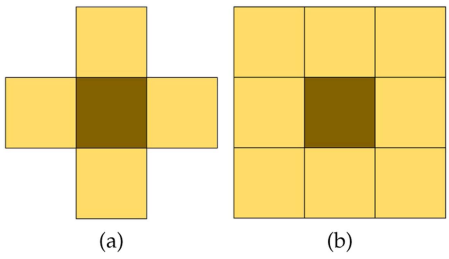


Fig. 2. Graphical representation of (a) 4-connectivity and (b) 8-connectivity. The "current" pixel is depicted in dark yellow and its neighborhood (connected pixels) is represented in yellow.

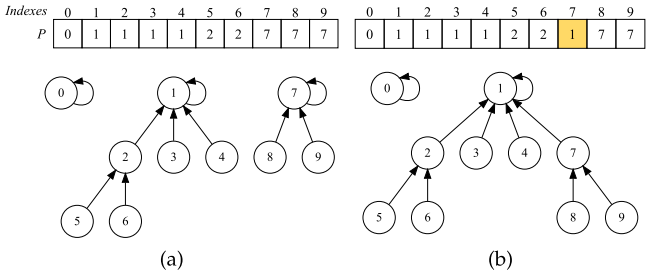


Fig. 3. Visualization of P represented as a forest of trees and stored in memory as an array. A *union(9,4)* performed on (a) will produce (b). Indeed, the root nodes of 9 and 4 are found through the *find* that respectively return 7 and 1. The tree with higher root value is then connected to the root of the other tree by updating $P[7]$ with 1.

Connected components labeling algorithms aim at assigning a different label to every connected component. When applied to I, the output of such an algorithm is a symbolic image L where, for every $p \in \mathcal{F}$, $L(p)$ is the label of the connected component that p belongs to ($[p]$), and for every $q \in \mathcal{B}$, $L(q) = 0$. Fig. 1 depicts an example of the labeling procedure applied to a fantasy character inspired by Space Invaders (Fig. 1(a)). When using 4-connectivity, the output will be that of Fig. 1(b), while using 8-connectivity will produce the symbolic image reported in Fig. 1(c).

A. The Union-Find Data Structure

During the labeling procedure, pixels belonging to the same connected component may be assigned different provisional labels; this requires a mechanism to keep track of possible equivalences and eventually solve them. One of the most effective strategies to achieve this goal is the forest-based implementation of the union-find data structure, which provides a quasi-linear solution to the disjoint-set union problem. This technique has been applied to sequential CCL for the first time by Dillencourt et al. [48], and it represents the basis of most modern approaches for label resolution, including those involving GPUs. As the name suggests, it consists of two basic operations —*find* and *union*— that are performed on a forest of *anti-arborescences*, i.e., directed rooted trees oriented towards the root. This forest, $\mathcal{P}$, represents a partition of the set $\mathcal{S}$ that corresponds to the lattice $\mathcal{L}$ in our scenario. The two operations can be defined as:

- *find* takes a node of a tree ($a \in \mathcal{P}$) as input and returns the corresponding root;
- *union* joins together the trees two given nodes ($a, b \in \mathcal{P}$) belong to, by setting the root of one tree (usually the one with the smallest value) as the father of the other one.

An implementation intended for GPU architectures is shown in Algorithm 1. To avoid concurrency issues, the union makes use of atomic operations.

When referring to CCL, all the pixels of the image are nodes in the data structure. Initially, each foreground pixel is a standalone tree; then, by means of union operations, trees of connected pixels are merged together, until a single tree for each connected component is obtained.

From an implementation point of view, the forest can be stored in an array P, with one index per pixel: at $P[i]$ we have the parent of node i. When $P[i] = i$, the node i is the root of the tree. In Fig. 3, an example of union and find operations executed on a P array is reported.

In sequential CCL algorithms, the union-find array is usually a stand-alone data structure, while most of the GPU implementations available in the literature store it directly in the output image, avoiding expensive system calls for GPU memory allocation and improving performance. Once a single tree per connected component is obtained, the *flattening* operation can be performed to complete the CCL task. The flattening procedure links each node of the array directly to the root of its tree, meaning that all the pixels of each connected component are associated with the same identifier. A possible GPU implementation of the flattening operation is represented by the `Compress` kernel (Algorithm 1). As first proposed in [49], the `Compress`

Algorithm 1: Pseudo-Code for `Union` Procedure and `Find` Function. L is Both the Union-Find Array and the Output Labeled Image, a and b are Both Array Indexes and Pixel Identifiers. `Compress` and `InlineCompress` are Two Variations of the Procedure Implementing the *Flattening* Operation.

```
1: function Find(L, a)
2:   while L[a] ≠ a do
3:     a ← L[a]
4:   return a
5: procedure Compress(L, a)
6:   L[a] ← Find(L, a)
7: function InlineCompress(L, a)
8:   id ← a
9:   while L[a] ≠ a do
10:    a ← L[a]
11:    L[id] ← a
12:  return a
13: procedure Union(L, a, b)
14:  done ← false
15:  while done = false do
16:    a ← Find(L, a)
17:    b ← Find(L, b)
18:    if a < b then
19:      old ← atomicMin(&amp;amp;amp; L[b], a)
20:      done ← (old = b)
21:      b ← old
22:    else if b < a then
23:      old ← atomicMin(&amp;amp;amp; L[a], b)
24:      done ← (old = a)
25:      a ← old
26:    else
27:      done ← true
```

implementation can be optimized for data-parallel environments and converted into `InlineCompression` (IC). In this case, the father of a label a is updated at every step of the tree traversal, thus making the update available to any possible concurrent thread that would save memory accesses.

To distinguish between background and foreground pixels a common approach is to shift labels, and start counting from 1 instead of 0. Such an approach ensures that every foreground pixel has a positive label, while 0 is reserved for the background.

In the following, we will simply use "union-find" when referring to its forest-based implementation.

III. Algorithms - Classification and Description

Now that a common terminology has been defined, we can dig into different proposals available in the literature. There are many different ways to categorize GPU CCL algorithms. Broadly speaking, one of the most common categorization distinguishes between *iterative* and *direct* solutions. The former repeats one or more kernels a data-dependent amount of times. Usually, they iterate until no more changes in data are detected. These algorithms tend to tolerate race conditions, demanding the correction of wrong results to the next iteration. Direct algorithms, on the other hand, perform a fixed number of kernel executions. Differently from iterative algorithms, each kernel must produce an exact result: therefore, most direct algorithms employ hard concurrency control to avoid race conditions, usually in the form of atomic operations.

Another possible classification may be based on the minimal set of pixels (or voxels) that are processed together. In this sense, pixel-based, run-based, and block-based algorithms can be identified:

- *Pixel-based* algorithms are the simplest: each foreground pixel has its own label;
- *Run-based* algorithms work on *runs*, i.e., chunks of consecutive foreground pixels in the same row, and assign labels to these runs instead of single pixels. This approach is beneficial in some scenarios, but performance significantly drops when foreground objects within images have elongated shapes on the y-direction;
- *Block-based* algorithms, originally proposed in [42], label 2×2 blocks of pixels at once. This is possible because, when considering 8-connectivity, any two foreground pixels in a 2×2 block are connected to each other. When moving to 3D the concept of blocks can remain at slice level or can be extended to the third dimension, considering $2 \times 2 \times 2$ blocks.

In the following of this Section, we will describe state-of-the-art solutions using the first classification. The timeline reported in Fig. 4 depicts the temporal release of each algorithm, providing the reference context. Later in this Section, a table summarizing all the characteristics of the algorithms (i.e., publication year, authors, the minimal set of pixels analyzed, scanning procedure, connectivity, input supported, data structure(s) required, and space complexity) is also provided (Table I).

A. Iterative Algorithms

1) Neighbor Propagation (NP): The first work that addresses the GPU CCL problem is due to Hawick et al. and is dated back to 2010 [50]. The authors deal with the general problem of identifying CCs in graphs and then propose four specific algorithms for d-dimensional hypercubic meshes, which include binary images. The first, and simplest, of these proposals is Neighbor Propagation. All foreground pixels are initialized with sequential values, each equal to the pixel raster index. Then, the provisional label of each pixel is updated to the minimum of its neighbor labels. This operation must be repeated until convergence, i.e., until there are no more changes in the output image. For this reason, the algorithm is inserted in the iterative group. The pseudocode for this algorithm is provided in Algorithm 2.

2) Local Neighbor Propagation (LNP): Local Neighbor Propagation is an optimization of NP that initializes the output image, splits it into blocks (16×16 in the original paper), and performs the update operations described above on each block separately. The advantage over the previous solution is that, in this case, the shared memory inside the block is exploited. As NP, each thread updates a different pixel. The update kernel is

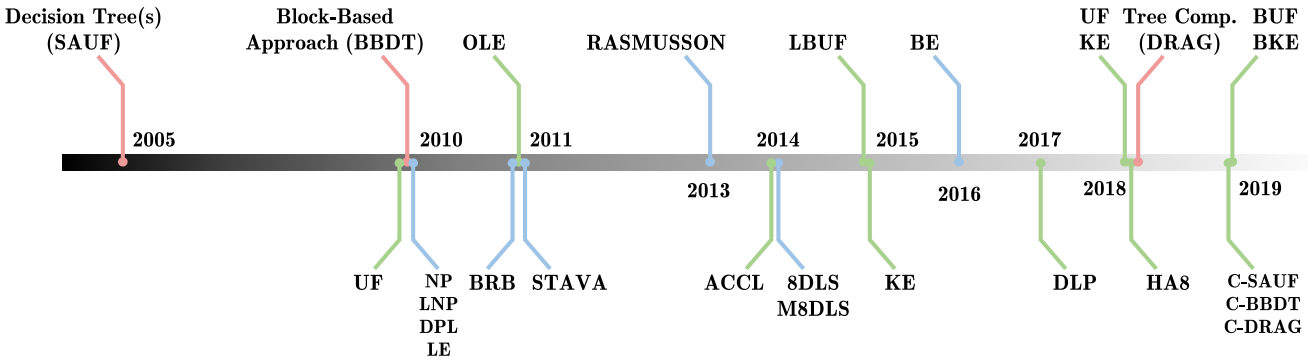


Fig. 4. Timeline of algorithm publications. Red lines represent CPU algorithms that introduced breakthrough changes in the labeling procedures, later included in GPU implementations. Blue and green lines are respectively used to identify iterative and direct solutions. Few algorithms appear twice in the timeline, meaning that the algorithm was initially released for a single connectivity type (e.g., 4-connectivity), and later extended to work with other types of connectivity (e.g., 8-connectivity) or with 3D volumes.

TABLE I
COMPARISON BETWEEN STATE-OF-THE-ART ALGORITHMS, WHICH CONSIDERS PUBLICATION YEAR, AUTHORS, MINIMUM BLOCK OF PIXELS PROCESSED TOGETHER (I.E., SINGLE PIXEL, BLOCK OR RUN), PROCESSING TYPE (I.E., ITERATIVE OR DIRECT), CONNECTIVITY, INPUT TYPE (2D OR 3D) AND DATA STRUCTURES EMPLOYED IN ADDITION TO THE OUTPUT IMAGE L

Name	*Year*	*Authors*	*Block Size*	*Iterative* vs *Direct*	*Connectivity*	*Input*	*Additional Data Structure(s)* [bytes]
NP [51]	2010	Hawick *et al.*	Pixel	Iterative	4, 8	2D	
LNP [51]	2010	Hawick *et al.*	Pixel	Iterative	4, 8	2D	
DPL [51]	2010	Hawick *et al.*	Run	Iterative	4	2D	
LE [51]	2010	Hawick *et al.*	Pixel	Iterative	4, 8	2D	$H = w \times h \times 4$ TB
UF [41]	2010	Oliveira and Lotufo	Pixel	Direct	4	2D	
BRB [54]	2011	Chen *et al.*	Block & Run	Iterative	8	2D	
OLE [52]	2011	Kalentev *et al.*	Pixel	Iterative	8	2D	TB
STAVA [53]	2011	Stava and Benes	Pixel	Iterative	8	2D	
RASMUSSON [56]	2013	Rasmusson *et al.*	Pixel	Iterative	8	2D	
8DLS [55]	2014	Soh *et al.*	Pixel	Iterative	8	2D	TB
M8DLS [55]	2014	Soh *et al.*	Pixel	Iterative	8	2D	
ACCL [59]	2014	Paravecino and Kaeli	Pixel	Direct	4	2D	
LBUF [50]	2015	Yonehara and Aizawa	Pixel	Direct	4, 8	2D	
KE [58]	2015	Komura	Pixel	Direct	4	2D	
BE [46]	2016	Zavalishin *et al.*	Block	Iterative	8	2D, 3D	$BL = \frac{w}{2} \times \frac{h}{2} \times 4$ $BA = \frac{w}{2} \times \frac{h}{2} \times 1$ TB
DLP [25]	2017	Cabaret *et al.*	Pixel	Direct	8	2D	
HA8 [24]	2018	Hennequin *et al.*	Run	Direct	4, 8	2D	
KE [57]	2018	Allegretti *et al.*	Pixel	Direct	8	2D	
UF [57]	2018	Allegretti *et al.*	Pixel	Direct	8	2D, 3D	
C_SAUF [47]	2019	Allegretti *et al.*	Pixel	Direct	8	2D	
C_BBDT [47]	2019	Allegretti *et al.*	Block	Direct	8	2D	
C_DRAG [47]	2019	Allegretti *et al.*	Block	Direct	8	2D	
BUF [36]	2019	Allegretti *et al.*	Block	Direct	8	2D, 3D	
BKE [36]	2019	Allegretti *et al.*	Block	Direct	8	2D, 3D	

For each additional data structure its dimension in bytes is also specified in the form *width* × *height* (×*depth*) × *bytes*. These structures have the same name used in the algorithm description. Some of the iterative algorithms require an additional byte (Termination Byte, TB) to check whether the process is completed or not after each iteration. Please note that bytes are specified considering 32-bit labels. The space complexity of each algorithm can be easily deduced by summing the last table column (additional data structures) to common output requirements, i.e., the size in bytes of the output image L.

divided into two steps: during the first step, each thread loads into shared memory the minimum neighbor label of its pixel, reading from device memory; then, the second step updates each label with the minimum of the neighbors, same as NP, but this time performed locally, in shared memory. This last operation is repeated until block convergence. Then, the content of the shared memory is copied into device memory, and the procedure restarts from the beginning until global convergence is reached. Algorithm 2 details the kernels of the algorithm.

3) Directional Propagation Labeling (DPL): Directional Propagation Labeling (DPL), again proposed in the same paper as NP and LNP [50], tries to overcome the limit of the

Algorithm 2: Neighbor Propagation and Local Neighbor Propagation Kernels. *I* is the Input Image, *L* is the Output Label Image, *id* is the Thread Identifier, Corresponding to a Pixel Raster Index. *r* and *c* Represent the Row and Column Index. *Changed* is an Output Variable to Communicate to the Host if Any Change Occurs. If so, the Kernel Must be Re-Run.

kernel NPI, L, r, c
  $label \leftarrow L(r,c)$
  for all $(n_r, n_c) \in \mathcal{N}(r,c)$ **do**
    if $I(n_r,n_c) = I(r,c) \wedge L(n_r,n_c) < label$ **then**
      $label \leftarrow L(n_r,n_c)$
      $changed \leftarrow$ true
  $L(r,c) \leftarrow label$
kernel LNPI, L, r, c
  $label \leftarrow L(r,c)$
  for all $(n_r, n_c) \in \mathcal{N}(r,c)$ **do**
    if $I(n_r,n_c) = I(r,c) \wedge L(n_r,n_c) < label$ **do**
      $label \leftarrow L(n_r,n_c)$
      $changed \leftarrow$ true
  __shared__ $sL[blockDim.x \times blockDim.y]$
  $r_s \leftarrow$ threadIdx.y
  $c_s \leftarrow$ threadIdx.x
  loop
    $sL(r_s,c_s) \leftarrow label$
    __syncthreads()
    $sChanged \leftarrow$ false
    for all $(n_r, n_c) \in \mathcal{N}(r,c)$ **do**
      if $I(n_r,n_c) = I(r,c) \wedge sL(n_r,n_c) < label$ **then**
        $label \leftarrow sL(n_r,n_c)$
        $sChanged \leftarrow$ true
    __syncthreads()
    if $sChanged =$ false **then**
      break
  $L(r,c) \leftarrow label$

Algorithm 3: Label Equivalence Kernels. *I* is the Input Image, *L* is Both Union-Find Array and Output Label Image, *id* is the Thread Identifier, Corresponding to a Pixel Raster Index. *r* and *c* Represent the Row and Column Index, Respectively.

kernel Init$I$, L, r, c
  if $I(r,c) = 1$ **then**
    $L(r,c) \leftarrow r * w + c + 1$
  else
    $L(r,c) \leftarrow 0$
kernel Scan$L$, r, c, $changes$
  $l \leftarrow L(r,c)$
  if $l > 0$ **then**
    $\min\_l \leftarrow$ `FindMinNeighborLabel(r, c)`
    if $\min\_l < l$ **then**
      $L[l-1] \leftarrow \min(L[l-1], \min\_l)$
      $changes \leftarrow true$
kernel Analysis$L$, r, c
  $l \leftarrow L(r,c)$
  if $l > 0$ **then**
    $L(r,c) \leftarrow Find(L,l)$

neighborhood-confined propagation. In order to accomplish this, it assigns to each thread the task of propagating the minimum label through a row or a column. This operation is performed alternatively in four directions: left to right, bottom to top, right to left, and top to bottom. Unlike the two previous algorithms, this strategy is only compatible with 4-connectivity.

4) Label Equivalence (LE): Label Equivalence is the last and certainly the most interesting proposal of the paper from Hawick et al. [50]. It is also one of the first iterative algorithms to record and solve equivalences using an auxiliary data structure.

The paper does not explicitly refer to the union-find, but the operations performed are the same and can be described by referring to them. In the first kernel, `Init`, the output image L is initialized as usual, with each pixel being given a provisional label equal to its raster index. Moreover, an additional data structure with the same size as the output image, H, is allocated and initialized in the same way. The algorithm then consists of three more kernels that are repeated in sequence until convergence: `Scan`, `Analysis`, and `Labeling`. The first kernel, `Scan`, is responsible for updating equivalences between pixels. Each thread works on a different pixel, finding the smallest of the neighbor labels and updating the corresponding value in H with the minimum label found. This operation does not touch L and only modifies H. The second kernel, `Analysis`, performs the compression of the equivalence tree, i.e., each thread is responsible for replacing a pixel of H with the root of the tree. This operation corresponds to *find*. The third and last kernel, `Labeling`, updates the labels in the output image L with the values contained in H. `Scan` requires atomic operations to avoid lost updates on H, in the case that two threads happen to concurrently update the same pixel.

After its appearance, LE has been improved by many authors. Kalentev et al. [51] noticed that the need for a separate data structure to store label equivalences could be removed through the direct use of the output image L. In this optimization, only two kernels are iterated after the initialization. `Analysis` kernel performs the compression of trees directly on the output image, updating the temporary label of each pixel. Thus, there is no need for a separate kernel devoted to copy the labels. In `Scan` kernel, every thread finds the smallest neighbor label, nl, and assigns it to the father (with label fl) iff $nl < fl$. Kalentev et al. also removed the atomic operations by simply increasing the number of iterations. The authors claim that atomic operations slow down computation more than additional iterations caused by collisions do. Kernels of this optimized version of Label Equivalence are detailed in Algorithm 3. The optimized version is identified as OLE, which stands for Optimized Label Equivalence.

5) Stava and Benes (STAVA): Stava and Benes [52] proposed a solution that can be seen as an improvement of LE, obtained with the use of Tiles Merging. The algorithm is composed of four kernels: `LocalLabeling`, `GlobalLabeling`,

`BorderCompression`, and `PathCompression`. In the first kernel, `LocalLabeling`, 16×16 tiles are labeled with a light version of LE, which does not require any additional union-find structure to store label equivalences, similar to the one proposed by Kalentev et al.. This kernel is detailed in Algorithm 4.

The second kernel, `GlobalLabeling`, merges the labeled tiles, in a hierarchical order: in step n, macro-tiles are obtained by merging groups of 4×4 tiles, the output of step $n-1$. Each group of tiles is processed by a three-dimensional thread block, where x and y components of the thread-id locally identify a tile. A union is performed between each pixel of the South and East border and its neighbors outside the tile. For each tile, the merging process is repeated until no more changes are detected. For optimization purposes, `GlobalLabeling` is alternated with `BorderCompression`, which performs find on border pixels only: the aim is to shorten union-find trees and consequently speed up the next round of global merge.

Finally, the last kernel, `PathCompression`, performs a find on all pixels of the output image, thus completing the CCL process.

6) Block-Run-Based (BRB): Block-Run-Based connected components labeling [53], as the name suggests, combines block-based and run-based strategies in order to simplify the equivalence label-solving process. The whole algorithm is implemented in GPU shared memory, thus minimizing global memory bandwidth consumption. The image, of width W (and half-width HW), is divided into overlapping Block Rows (BR), composed of two consecutive pixel rows, where the upper pixel row is shared with the previous block row. Each thread works on a 2×2 block. The algorithm is composed of two scans, and the first scan is divided into three steps: `BlockRunExtraction`, `PreviousBRUpdate`, and `CurrentBRUpdate`.

In the first step, the current BR is compressed into one-pixel row by applying `OR` operation of its upper and lower row pixel value. Then, a run of contiguous 1 s is detected from the compressed pixel row and the column index of the block containing the run starting pixel is assigned to each block in this run, by means of `__ballot_sync`[1] and `__clz` (Count Leading Zeros) instructions. From another perspective, each block can be considered as a node of a union-find tree, while the label value represents a linkage address pointing to the column location of its parent node. The purpose of the following two iterative steps is to merge union-find trees of current and previous BRs according to the connection relations. The second step, `PreviousBRUpdate`, merges union-find trees of pairs of connected blocks in the current and previous row, by linking one root to the other one. Then, `CurrentBRUpdate` is similar to the previous step, but only updates root nodes in the current BR. These two steps are iterated until convergence.

The second scan, `LabelAssignment`, proceeds row-wise in a reversed direction, and assigns a unique global label to each block recognized as a tree root. Then, child blocks get their label from the parent node.

Algorithm 4: Stava and Benes `LocalLabeling` Kernel. *I* is the Input Image, *L* is Both Union-Find Array and Output Label Image, *id* is the Thread Identifier, Corresponding to a Pixel Raster Index. *r* and *c* Represent the Row and Column Index, Respectively.

```
1: kernel LocalLabeling(I, L, r, c)
2:   __shared__ sI[16 × 16]
3:   __shared__ sL[16 × 16]
4:   __shared__ changes[1]
5:   loc_r ← threadIdx.y
6:   loc_c ← threadIdx.x
7:   sI(loc_r, loc_c) ← I(r, c)              ▷ Load input patch
8:   __syncthreads()
9:   l ← loc_r * blockDim.x + loc_c
10:  loop                                   ▷ Pass 1 of the CCL algorithm
11:    sL(loc_r, loc_c) ← l
12:    if threadIdx.x = 0 ∧ threadIdx.y = 0 then
13:      changes[0] ← false
14:    __syncthreads()
15:    new_l ← l                            ▷ Find the minimal neighbor label
16:    for (n_r, n_c) ∈ N(r, c) do
17:      if sI(loc_r, loc_c) = sI(n_r, n_c) then
18:        new_l ← min(new_l, sL(n_r, n_c))
19:    __syncthreads()
20:    if new_l < l then                    ▷ Merge the equivalence trees
21:      atomicMin(sL[l], new_l)
22:      changes[0] ← true
23:    __syncthreads()
24:    if changes[0] = false then
25:      break                              ▷ Local solution has been found
26:    l ← Find(sL, l)                      ▷ Pass 2
27:    __syncthreads()
28:  g_l ← convertLabelToGlobal(l)
29:  L(r, c) ← g_l                          ▷ Store result to device memory
```

Unfortunately, the implementation available only deals with specific image dimensions (e.g., 512×512 and 1024×1024). Any different size would require a specific implementation or multiple conditional checks that would significantly degrade the performance.

7) 8-Directional Label Selection (8DLS): 8-Directional Label Selection [54] is another iterative algorithm in which, at every step, each pixel is assigned the minimum label found scanning each of the 8 directions radiating from it until a background pixel is encountered. Modified 8-Directional Label Solver (M8DLS) is an improved version of 8DLS, that avoids processing pixels that already have a minimum label. After two iterations of 8DLS, pixels that still have their original label are considered permanent. The specific amount of iterations, 2, is empirically determined, based on exhaustive tests conducted by the authors. Unfortunately, when using such a kind of "optimization", the correctness of the output may not be guaranteed.

[1] `__ballot_sync` is a warp vote function: it evaluates a predicate for all the threads in the warp and returns an integer whose n-th bit is set iff the predicate evaluates to non-zero for the n-th thread.

Algorithm 5: Block Equivalence `Init` Kernel. I is the Input Image, L is Both Union-Find Array and Output Label Image, *id* is the Thread Identifier, Corresponding to a Pixel Raster Index. r and c Represent the Row and Column Index, Respectively.

kernel Init$I$, bL, $bConn$, r, c
  $P \leftarrow 0$
  $P0 \leftarrow 777_{16}$
  if $I[2r, 2c] > 0$ **then**
    $P \leftarrow P \mid P0$
  if $I[2r, 2c{+}1] > 0$ **then**
    $P \leftarrow P \mid (P0 \ll 1)$
  if $I[2r{+}1, 2c] > 0$ **then**
    $P \leftarrow P \mid (P0 \ll 4)$
  if $I[2r{+}1, 2c{+}1] > 0$ **then**
    $P \leftarrow P \mid (P0 \ll 5)$
  if $P > 0$ **then**
    $bL[r, c] \leftarrow r{*}w{+}c{+}1$
    if HasBit$(P, 0) \wedge I[2r{-}1, 2c{-}1] > 0$ **then**
      SetBit$(bConn[r, c], 0)$
    if (HasBit$(P, 1) \wedge I[2r{-}1, 2c] > 0) \vee$
      (HasBit$(P, 2) \wedge I[2r{-}1, 2c{+}1] > 0$) **then**
      SetBit$(bConn[r, c], 1)$
    ...

8) Rasmusson: Rasmusson et al. [55] proposed a method based on early calculation of label propagation sizes, from the connectivity between pixels extracted in a pre-processing step, and reuse of established label propagation routes. As usual, the algorithm starts by initializing the output with provisional labels equal to pixel raster indices. The four most significant bits of each label store connection information about half the neighborhood; because of this trick, the algorithm only works on images with at most $2^{28} \approx 256$ M pixels.

The core of the algorithm is the `Propagate` kernel, which is repeated until convergence. The image is divided into 32×32 tiles, which are copied into shared memory, and a thread is created for each pixel. Then, for each of the 8 possible directions starting from a pixel, the maximum propagation is computed. These propagation sizes are iteratively calculated in a parallel manner: each thread stores the provisional propagation size in a local variable, p, and increments it at each step by the value of the provisional propagation size of the pixel distant p positions. In this way, a maximum propagation of 32 is reached after $\log_2 32 = 5$ iterations. Different threads share the respective values of p by means of shuffle operations, which efficiently exchange a variable between threads within a warp. The pixel-thread association must change for each direction, in order for pixels aligned in a certain orientation to always be in the same warp. The label of each pixel is updated to the minimum of the 8 labels in the maximum propagation sizes, and also a propagation size of 1 is considered for the correctness of corner cases. In order to extend the label propagation to neighbor tiles, actually 33×33 blocks are copied into shared memory.

9) Block Equivalence (BE): Proposed by Zavalishin et al. [45], Block Equivalence (BE in short) is a block-based version of Label Equivalence (Section III-A4), enriched with a novel method for checking block connectivity in a parallel fashion. The algorithm is composed of four kernels: `Init`, `Scan`, `Analysis`, and `FinalLabeling`, each requiring a thread per block. The `Init` kernel is responsible for finding which pairs of neighbor blocks are indeed connected. Decision trees, used by the original block-based CCL algorithm [42], tend to cause thread divergence and inefficiency in a GPU environment; therefore, a different method is proposed, that avoids nested conditional jumps:

1) Each internal pixel of block X is read, in order to find out which neighbor blocks could possibly be connected to X;
2) Internal pixels of the neighbor blocks selected by the previous step are read, and the hypothesis of connectivity with X is confirmed or refused.

At the end of `Init`, block adjacency information is stored in an ad-hoc matrix BA, to be read again in the subsequent phases. Each block has 8 adjacent blocks, so one bitmapped byte per block is sufficient to record the neighbors (adjacent blocks sharing the same value). Kernels `Scan` and `Analysis` behave in the same way as the kernels of Label Equivalence, in the optimized version of Kalentev et al.; the only difference is that they work on blocks instead of pixels, storing block labels in a specific matrix, BL. `Scan` and `Analysis` are iterated until convergence, and then `FinalLabeling` copies block labels into pixel labels, thus composing the output. The pseudo-code is provided in Algorithm 5.

Algorithm 6: `LocalMerge` Kernel of UF. I is Input Image, L is Both Union-Find Array and Output Label Image, *id* is the Thread Identifier, Corresponding to a Pixel Raster Index. r and c Represent the Row and Column Index, Respectively.

kernel Local Merge$I$, L, r, c
  __shared__ $sI[]$
  __shared__ $sL[]$
  $r_s \leftarrow$ threadIdx.y
  $c_s \leftarrow$ threadIdx.x
  $sI(r_s, c_s) \leftarrow L(r, c)$
  __syncthreads()
  for all $(n_r, n_c) \in \mathcal{N}(r, c)$ **do**
    if $I(r, c) = I(n_r, n_c)$ **then**
      `Union`(sL, id_s, id_{sn})
  $l \leftarrow$ `Find`(sL, id_s)
  $L[id] \leftarrow$ `convertLabelToGlobal`(l)

B. Direct Algorithms

1) Union-Find (UF): Union-Find, by Oliveira and Lotufo [41], is chronologically the first non-iterative proposal, and also one of the first GPU CCL algorithms overall. It consists of a parallel version of the common union-find algorithm, largely used in sequential CCL solutions. UF is based on three kernels: `Initialization`, `Merge`, and `Compression`. Each kernel is launched on a number of threads equal to the image size, and each thread is assigned a pixel, x. In order to avoid time-consuming memory allocations, no additional memory is

reserved for the union-find data structure. Instead, the equivalences between labels are stored, during the procedure, in the output image itself, in the sense that the provisional label assigned to a pixel corresponds to the linear index of its father node in the union-find forest. In the `Initialization` kernel, all foreground pixels in the output image are initialized with a provisional label equal to their linear index. From the union-find point of view, this procedure corresponds to the creation of a separate tree for every pixel. In kernel `Merge`, each thread working on a foreground pixel x checks its neighborhood mask, and for every foreground neighbor it performs a union with x. Since union is a symmetric operation, the neighborhood mask only contains half of the neighbors of a pixel. The union operation can involve reading and writing operations on other pixels than x. The possibility that two or more threads read or modify the same pixel is taken into account through the use of atomic operations in union, in order not to lose updates, and avoid the necessity of multiple iterations. After `Merge`, every connection between pixels in the image is reflected in the union-find structure, in the sense that every separate tree represents a connected component. Finally, the `Compression` kernel performs the compression of trees. This operation makes sure that every pixel is assigned a label corresponding to the linear index of the root of its tree. Thus, every pixel in the same CC ends up sharing the same label, and the labeling task is completed. The aforementioned basic structure of the algorithm is enhanced with the addition of another common technique known as *tile merging*. This means that the entire algorithm is first performed on rectangular blocks the image is divided into in a preliminary phase called `Local Merge`. This allows to take full advantage of shared memory. This local phase is detailed in Algorithm 6. Then, the `Merge` kernel is performed on border pixels only, and a final `Compression` is executed over the entire image. The original algorithm uses 4-connectivity, but it was extended to 8-connectivity in [56], by adding in the `Merge` kernel the diagonal directions to the neighborhood of a pixel.

2) Line-Based Union-Find (LBUF): Line-Based Union-Find [49] is a variation of UF that employs single lines as tiles for the tile merging strategy. This choice reduces the neighborhood of every pixel to a subset containing only the two neighbors belonging to the same row, and consequently allows to simplify the logic of `Local Merge`. In fact, because only half the neighborhood needs to be checked (see Section III-B-1), each thread only has to link its pixel to the one on the left, in the case that both are foreground. The remaining kernels of UF are left unchanged.

3) Komura Equivalence (KE): Komura Equivalence [57] can be seen as a variation of UF, which involves a more complex initialization in order to remove the need for one union per pixel later. It consists of four steps: `Initialization`, `Compression`, `Reduction`, and `Compression` again. The main difference between KE and UF lies in the first kernel: differently from UF, KE `Initialization` does not create single-node trees. Instead, every thread checks the neighborhood of x in increasing raster index order, and the smallest foreground neighbor is set as the father node of x. This first phase aims at creating partial union-find trees right from the beginning, that are flattened by the subsequent `Compression` kernel. The `Reduction` kernel is a variation of `Merge`. It only performs a union between x and foreground neighbors that were not chosen during `Initialization`. Analogously to UF, a final `Compression` is required for flattening the forest trees. Same as UF, the original KE is a 4-connectivity algorithm. It is extended to 8-connectivity in [56], by adding the supplementary neighbors in both `Initialization` and `Reduction`.

4) Accelerated CCL (ACCL): The CUDA compute capability 3.5 introduced several features, among which dynamic parallelism and Hyper-Q are the ones exploited by the Accelerated CCL (ACCL) algorithm, proposed by Paravecino and Kaeli in [58]. ACCL is configured as a run-based algorithm and is composed of two phases. In the first phase, a thread is run for each row of the image, with the aim of finding runs and recording the first and last indices. During this step, each run is assigned a provisional label L, recorded in a separate matrix. During the second scan, connected runs from contiguous rows are joined. This particular operation involves spawning a child kernel for each merge operation, with one thread per provisional label in L. These threads update the respective labels of the added segment. The algorithm makes use of texture memory for read-only memory accesses, which improves the performance when multiple threads from the same block access contiguous memory locations. Finally, the Hyper-Q feature is exploited to process multiple images at the same time with different CUDA streams, thus increasing the total throughput.

5) Distanceless Label Propagation (DLP): Proposed by Cabaret et al. [25], Distanceless Label Propagation is another algorithm based on union-find, characterized by an uncommon gather-scatter procedure used to propagate the minimum label of a CC. It includes four kernels: `DLP-I`, `DLP-SR`, `DLP-R` and `DLP-RUF`. The first kernel, `DLP-I`, initializes foreground pixels as usual, with a provisional label equal to the raster index. The second kernel is `DLP-SR` (SetRoot), and is responsible for a gather-scatter label propagation procedure. The minimum label is found in a mask selecting a reduced neighborhood of x, containing 2×2 pixels (x, right, down, and down-right); then, differently from most algorithms based on minimum label gathering, the minimum is scattered back to the whole mask with the SetRoot procedure: for each involved pixel, the root of its union-find tree is updated with the minimum label just found, by means of the usual atomic operation that characterizes direct algorithms. The third kernel, `DLP-R`, performs the classical compression of union-find trees, replacing provisional labels with the root of their trees. The last kernel is `DLP-RUF`, named after its distinctive algorithm, *recursive* union-find. It is the same as `DLP-SR`, but replaces the SetRoot procedure with a recursive implementation of the union. These four different kernels are used to build three sequential steps that compose the CCL algorithm: *(i) Tile Labeling* - the labeling process is performed locally on tiles, by running in sequence `DLP-I`, `DLP-SR`, `DLP-R`, `DLP-RUF` and finally another `DLP-R`; *(ii) Border Merging* - `DLP-RUF` is applied to border pixels only,

with the aim of merging union-find trees of different tiles; *(iii) Relabeling* - `DLP-R` is applied to the whole image.

6) Hardware Accelerated 4/8 (HA4/HA8): In 2018, Hennequin et al. [24] proposed HA4, a 4-connected run-based algorithm heavily based on CUDA intrinsics. The algorithm can be used to perform either connected components labeling or analysis; the CCL variation is composed of three kernels: `StripLabeling`, `BorderMerging`, and `FinalLabeling`.

The first kernel, which is also the most complex, performs partial labeling on horizontal strips. The block width is 32, the same as the warp size, and each thread is assigned a pixel X so that a line of 32 pixels perfectly matches a warp. The kernel begins with threads reading the values of their pixels and sharing them with the rest of the line (warp) with a `__ballot_sync` instruction, which produces a 32-bit bitmask of the foreground pixels; then, each thread checks the *start* and *end* position of the run X belongs to, by means of efficient `__ffs` (Find First Set) and `__clz` intrinsics. The first pixel of each run is given a temporary label equal to its raster index, while the other pixels are left uninitialized. Then, in order to merge lines vertically, the first thread of the line copies the bitmask generated earlier with `__ballot_sync` in shared memory, so that it can be read from the lower line. Each thread performs a union if X or the pixel above is the start of a run. After this partial labeling of the block, the same threads shift 32 pixels to the right and continue labeling the stripe with the same method, extending runs that overcome the block border. This technique avoids the need for merging vertical borders and minimizes thread creation/destruction.

The second kernel merges the horizontal border, performing a union only on the start of runs, same as in `StripLabeling`. Finally, the last kernel performs the final labeling. Only the run-starting-pixel thread retrieves the final label with a *find*, and then it propagates it to the other threads with a shuffle operation; finally, each thread updates the label image with this value.

Since the original algorithm is 4-connected, we propose an 8-connected variation, which performance can be compared to the other algorithms in this survey. In order to make the algorithm 8-connected, it is sufficient to add the diagonal direction to the horizontal border merging and perform a union if the rightmost of the two adjacent pixels is a start. Specifically, each thread must merge X to the upper-left pixel if X is a start, and to the upper-right pixel if the latter is a start.

Checking and merging in a diagonal direction gets tricky at warp borders since the diagonal neighbor pixels can fall into different warps. To tackle such cases, blocks of the `BorderMerging` kernel encompass 1024 pixel-long lines, where each warp shares the value of the rightmost pixel to the subsequent warp using shared memory.

7) Decision Tree-Based Algorithms: C_SAUF, C_BBDT, and C_DRAG, published in [46], are CUDA adaptations of successful CCL algorithms originally designed for running sequentially on a single CPU thread. The general structure of these algorithms is the following:

- *First scan:* scans the input image using a mask of already visited pixels, and assigns a temporary label to the current pixel(s), recording any equivalence between those found in the mask;
- *Flattening:* analyzes the registered equivalences and establishes the definitive labels to replace the provisional ones;
- *Second scan:* generates the output image by replacing provisional labels with final ones.

The first scan is the most complex and time-consuming, and it has been targeted by several improvements.

Scan Array-based Union-Find (SAUF), proposed by Wu et al. in [59], reduces the number of neighbors visited during the first scan using a *decision tree*. The idea is that if two already visited pixels are connected, their labels have already been marked as equivalent in the union-find data structure, so it is not necessary to check their values.

This algorithm was extended in [42] with the introduction of 2×2 blocks, in which all foreground pixels share the same label. Since the neighboring mask becomes larger, the authors propose an optimal strategy to automatically build the decision tree by means of a dynamic programming approach [60]. The resulting algorithm is known as BBDT.

In [61], authors noticed the existence of identical and equivalent subtrees in the BBDT decision tree. *Identical* subtrees were merged together by the compiler optimizer, with the introduction of jumps in machine code, but *equivalent* ones were not. By also taking into account equivalent subtrees they converted the decision tree into a Directed Rooted Acyclic Graph, which they called DRAG. The code compression obtained does not impact neither on the memory accesses, nor on the number of comparisons, but allows a significant reduction of the machine code footprint. This heavily reduces the memory requirements increasing the instruction cache hit rate and the overall run-time performance.

The adaptation of these algorithms to the CUDA domain, as proposed in [46], consists in translating each step into an appropriate kernel, plus an additional starting one for initializing provisional labels. Each kernel uses one thread per pixel (C_SAUF) or per block (C_BBDT, C_DRAG). The first kernel, `Initialization`, just sets the label of X to its raster index, the same as in many other algorithms. The second kernel, `Merge`, deals with the recording of equivalences between labels. During execution, thread t_x traverses a decision tree in order to decide which action needs to be performed, while minimizing the average amount of memory accesses. When no neighbors in the scanning mask are foreground, nothing needs to be done. In all other cases, the current label must be merged to those of connected pixels, with a union. Then, it is easy to parallelize the flattening step: in the third kernel, `Compression`, thread t_x performs $L[id_x] \leftarrow Find(L, id_x)$ to link each provisional label to the representative of its union-find tree. This is equivalent to the homonymous kernel in UF (Section III-B-1).

The last step of the sequential algorithms is the *second scan*, which updates labels in the output image L. A large part of the job of *second scan* is not necessary in the parallel implementations, because `Compression` already solves label equivalences in the output image. For C_BBDT and C_DRAG, a final processing of L is required to copy the label assigned to each block into

its foreground pixels. This job is performed by the last kernel, `FinalLabeling`.

Although branches in the code are known to cause thread divergence and slow down performances when dealing with CUDA parallel architectures, the authors of [46] demonstrated that in real case scenarios, where the foreground density of input images is low ($< 10\%$) or high ($> 90\%$), the benefits introduced by the use of decision trees do compensate for the cost of thread divergence. Indeed, when foreground density satisfies such conditions the decision is usually taken in the first levels of the decision trees employed for the scanning phase, saving many memory accesses without breaking the thread execution flow.

8) Block-Based Union-Find (BUF): Block-based Union-Find (BUF) [36] is the first of the direct algorithms which employ a block-based approach. It shares the same kernels as UF (`Initialization`, `Merge`, and `Compression`), but each phase works on blocks instead of single pixels, and there is an additional final kernel, `FinalLabeling`, to copy block labels into pixels ones. Until the end of the algorithm, block labels are stored in the output image, in the upper-left pixel of every block. In this way, it avoids the allocation of unnecessary device memory. The method used to check connectivity between 2×2 blocks is borrowed from BE (Section III-A-9), and is performed in the `Merge` kernel.

9) Block-Based Komura Equivalence (BKE): Block-based Komura Equivalence (BKE), introduced along with BUF in [36], is another direct block-based algorithm, this time obtained by modifying KE (Section III-B-3). BKE is quite similar to BUF in what it does; the major difference between the two is that BKE, the same as KE, starts linking together connected blocks directly in the initialization phase. This means that the task of finding what blocks are connected to the active one must be anticipated in the first phase. The method is the same one used for BUF and BE, but in this case, as in KE, only the connected neighbor block with the smallest linear index is linked to X. The remaining connectivity information is stored in a variable, to be used later in `Reduce`, where the remaining neighbor blocks are merged to X. The neighbors that can be merged in `Reduce` are the ones to the North, North-East, and West. Being it the first one considered for the linking performed during `Initialization`, North-West is excluded. So, 3 bits are enough to store connectivity information for the `Reduce` phase. In order to avoid the allocation of additional memory, these bits are stored in the output image, in the top-right pixel label slot, that would otherwise be left unused until `FinalLabeling`, when connectivity information is not necessary anymore. The `Reduce` kernel then reads this connectivity information and performs the necessary union operations.

Both BKE and BUF make use of the *inline compression*, an optimization designed for the use of union-find in data parallel environments. Previously introduced and described in Section II-A, it aims at reducing the size of equivalence trees created and updated during `Initialization` and `Merge`/`Reduction` phases. Thus, possible concurrent threads performing the find on the same node may save memory accesses.

IV. Benchmarking

The term *reproducible research* is usually referred to as the approach of presenting scientific claims together with all information needed to reproduce the results, so that others may verify the findings and build upon them. As researchers, we strongly believe in and support reproducibility in scientific research. For this reason, in 2016 we publicly released the first version of YACCLAB [62] —Yet Another Connected Components LAbeling Benchmark—, an open-source benchmarking system that allows to fairly compare CCL algorithms on different environments. YACCLAB has been originally described in [47] and later extended in [63] that included more literature algorithms and additional evaluation criteria, making it more general and flexible. In 2019, a new version of the benchmark including the possibility of evaluating 3D algorithms and GPU-based implementations has been released [36]. The benchmark represents nowadays the *de-facto* standard for evaluating CCL proposals, being it employed by several authors [33], [64], [65], [66].

When measuring the performance of an algorithm, many different subtleties must be taken into account. Although datasets have been blamed for narrowing the focus of research and reducing it to a single benchmark performance number, it is clear that the ability to compare different techniques on the same data is essential to choose which algorithm suits specific needs best [67]. In order to ensure a common and standard test set, YACCLAB is provided with an open-source binary dataset, which covers most of the CCL application scenarios, including both 2D images and 3D volumes. Since the whole project is open source, anyone can contribute by increasing and improving the YACCLAB dataset with different application environments. The current set of images available in the benchmark and used to compare algorithm performance in this survey is described in Section IV-A. Before the publication of YACCLAB, many novel proposals have been published and almost none of them compared on the same data [27], [68] making performance comparison almost impossible.

Benchmarking may be a problem by itself, because measuring performance may not be obvious, but CCL has an exact result and this reduces the burden of the evaluator to the measurement of how fast an algorithm is. However, setting the correct parameters when reproducing other people tests may be a problem, changing the final figures by orders of magnitude. This is why publishing the source code together with a scientific paper or, at the very least, an executable should be mandatory. YACCLAB tackles these aspects by including the source code of all the most significant proposals of the last twenty years, considering both CPU- and GPU-based algorithms. All the implementations have been taken from the original publication, whenever available, or produced by following the paper pseudo-code and description. Being the code open source, anyone can check whether implementation details are correctly handled or not, and test algorithms on their own setting (operating system, compiler, dataset, etc.), verifying any claim found in the literature and selecting the algorithm that performs the best on their specific environment.

With this paper, YACCLAB has been further extended to include many of the above-described algorithms that were previously

Fig. 5. Samples of the YACCLAB 2D datasets. From left to right, top to bottom: 3DPeS, Fingerprints, Medical, MIRflickr, Tobacco800, XDOCS, and Hamlet.

missing in the benchmarking suites. In the following of this Section, datasets, and evaluation criteria employed to later compare the algorithms are summarized.

A. Evaluation Dataset

Following a common practice in literature [25], [26], [45], the YACCLAB dataset includes both synthetic and real images, addressing both 2D and 3D scenarios. All images are provided in 1 bit per pixel PNG format, with 0 being background and 1 being foreground. The dataset can be automatically downloaded during the setup of the YACCLAB benchmark.

1) 2D Datasets (Fig. 5): MIRflickr [69]. This is the Otsu-binarized [70] version of the MIRflickr dataset. It contains a set of 25 000 natural images taken from Flickr with few connected components and an average density of 0.4459 foreground pixels.

Medical [71]. This dataset covers applications of CCL on medical data. It is composed of 343 binary histological images with an average amount of 1.21 million pixels to analyze and 484 components to label. This dataset is intended to cover applications of CCL related to medical image analysis.

Hamlet. A set of 104 images, scanned from the Hamlet provided by the Gutenberg Project [72]. Images have an average amount of 1 447 components to label and an average foreground density of 0.0789.

Tobacco800. Composed of 1 290 document images, it has been collected and scanned using a wide variety of equipment over time. Image sizes range from $1\,200 \times 1\,600$ to $2\,500 \times 3\,200$ pixels [73], [74], [75].

XDOCS. A collection of Italian civil registries recently digitalized [17], [76], [77]. This dataset is composed of 1 677 high resolution images with an average size of $4\,853 \times 3\,387$ and 15 282 components to analyze. The average foreground density is quite low: 0.0918.

Fingerprints. This dataset counts 960 fingerprint images taken from three fingerprint verification competitions (FCV2000, FCV2002, and FCV2004) [78]. Images were collected using low-cost optical sensors or synthetically generated, binarized using an adaptive threshold [79], and finally negated. Their size varies from 240×320 up to 640×480 pixels.

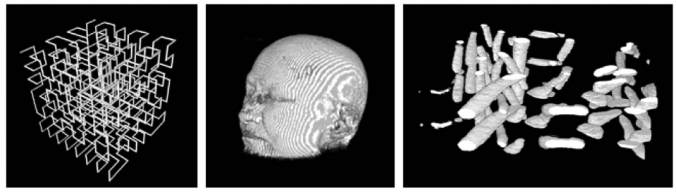


Fig. 6. Samples of the YACCLAB 3D datasets. From left to right: Hilbert space-filling curve, OASIS, and Mitochondria medical imaging data.

3DPes. This is a set of images coming from the surveillance dataset 3DPeS (3D People Surveillance Dataset) [80]. The background models for all cameras are provided by the authors, so a very basic technique of motion segmentation has been applied to generate the foreground binary masks, i.e., background subtraction and Otsu thresholding [70]. The analysis of the foreground masks to remove small connected components and to match the nearest neighbors is a common application for CCL.

2) 3D Datasets (Fig. 6): OASIS. A MRI data collection taken from the Open Access Series of Imaging Studies (OASIS) project [81]. The 373 volumes of $128 \times 256 \times 256$ voxels were binarized with the Otsu threshold [70] calculated over the entire volume.

Mitochondria. This is the Electron Microscopy Dataset available in [82] and originally published in [83]. The original dataset contains sections taken from the CA1 hippocampus region of the brain. All the binary volumes provided by the authors have been included in YACCLAB, for a total of three volumes composed by 165 slices with a resolution of $1\,024 \times 768$ pixels.

Hilbert. The Hilbert curve is a fractal space-filling curve described by David Hilbert. This curve represents a challenging test case for the labeling algorithms and YACCLAB 3D-dataset includes six volumes filled with the 3D Hilbert curve obtained at different iterations (1 to 6) of the construction method. Final binary volumes have a size of $128 \times 128 \times 128$.

3) Random Synthetic Images: A set of black and white random noise images to stress how the behavior of algorithms varies with respect to the *density* (percentage of foreground pixels) and *granularity* (minimum size of foreground blocks). Specifically, two different test sets are available in YACCLAB, one for 2D images and one for 3D volumes. The former contains $2\,048 \times 2\,048$ images, while the latter has $256 \times 256 \times 256$ volumes. Images and volumes have been generated with the Mersenne Twister MT19937 random number generator, included in the *C++* standard library [84]. Density ranges from 0% to 100% with a step of 1%. For every density value, each integer granularity $g \in [1, 16]$ has been considered. Ten images have been generated for every couple of density-granularity values, for a total of 16 160 images, both for 2D and 3D. Samples from the YACCLAB 2D synthetic dataset are reported in Fig. 7, considering different levels of granularity. In other words, granularity corresponds to the minimum size of (non-overlapped) foreground blocks that appear in an image. When changing the granularity, it also changes the shape that foreground objects may assume, introducing additional difficulties (that vary from algorithm to algorithm) in the labeling procedure.

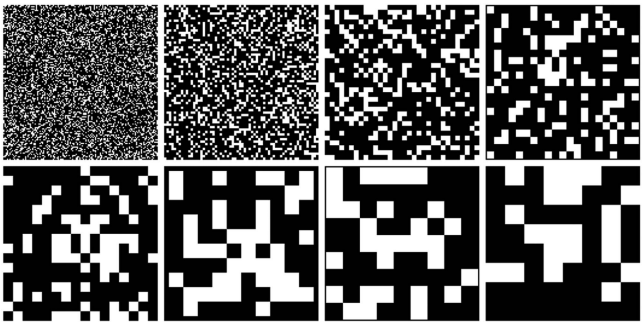


Fig. 7. Samples of the YACCLAB 2D synthetic dataset. Example images have a foreground density of 30% and, from left to right, top to bottom, granularities are 1, 2, 4, 6, 8, 12, 14, 16.

B. Assessment Strategies

The YACCLAB benchmark provides different tests that can be used to stress algorithm behavior under different perspectives. The simplest evaluation mechanism consists of a direct comparison of execution times, including the time for allocating data structures. Algorithms are run on all the available images, and the average execution time per dataset is recorded.

The benchmark also allows to separately evaluate the different phases composing the algorithms, separating the memory allocation time from the core execution of the labeling procedure. Two-stage algorithms can also distinguish between the *Local Labeling* and the *Tiles Merging* steps. The impact of each phase can thus be stressed and analyzed through such experiments, highlighting strengths and weaknesses.

Granularity is the last experiment available and consists of measuring the execution time on synthetic images when foreground pixel density and granularity change. This approach allows us to draw additional conclusions, highlighting behaviors that may not emerge from the previous tests.

All the experiments carried out with YACCLAB on GPU algorithms assume that the input image is already available in the GPU memory before the algorithm begins. The rationale behind such an evaluation choice is that the algorithms' execution time does not depend on the data transfer between the host and the GPU. Comparing performance by including the data transfer time will simply introduce a fixed equal "delay", distracting from the CCL execution time. In the same way, YACCLAB assumes that the output is required in the device memory and no additional data transfer is needed. The allocation of the output GPU image and of any other data structure potentially required by the algorithm itself is instead considered in the total execution time.

V. Evaluation

According to the *Gestalt psychology* of perception, our senses operate the law of closure, perceiving objects as a whole even if they are loosely connected as for the 8-connectivity [42]. This is the reason why many of the latest CCL proposals focus on 8-connectivity. Based on this observation, and considering the amount of environments already covered by our analysis, we focus our experiments on the 8-connectivity for 2D images. On 3D volumes, 8-connectivity maps to 26-connectivity, which is employed in our analysis of 3D implementations.

As previously mentioned, many literature solutions considered in this survey have no associate public source code, forcing us to re-implement them. All the algorithms are identified by their acronym already mentioned in Section III.

A. Hardware and Software

The comparative evaluation is carried out on two different machines, using different experimental setups.

Setup A. This test machine is equipped with an Intel(R) Core(TM) i7-4790 CPU @ 3.60 GHz (with 4×32 KB L1 cache, 4×256 KB L2 cache, 8 MB L3 cache) and two GPUs: an NVIDIA Quadro P1000 and an NVIDIA Quadro K2200, both with 4 GB of GDDR5 GPU Memory and 640 CUDA Cores. The former has a *Pascal* architecture, while the latter is based on *Maxwell* architecture. In such environments, two different operating systems have been employed: Windows 19.11.25508.2 and Ubuntu 22.04 LTS, respectively combined with MSVC 19.11.25508.2 and GCC 9.3.0 compilers. In both cases, NVCC 11.0 has been used to compile CUDA files. NVIDIA Driver versions 522.06 and 450.51 have been employed on Windows and Linux respectively.

Setup B. The second experimental setup is a server machine equipped with 52 Intel(R) Xeon(R) Gold 5320 CPU @ 2.20 GHz (with 48 KB L1d cache, 32 KB L1i cache, 1 280 KB L2 cache, and 40 MB L3 cache) and an NVIDIA A100 80 GB PCIe GPU based on *Ampere* architecture with 80 GB of HBM2 with ECC GPU Memory and 6 912 CUDA Cores (NVIDIA Drivers version 525.60.13). In this case, Ubuntu 22.04 LTS has been combined with GCC 8.5.0 and NVCC 12.0 compilers.

The kernel sensitivity to run-time parameters such as thread block and grid dimensions is not discussed. Although these parameters can significantly affect performance in CUDA programs, we stick to those provided in the original papers by verifying that such configurations provide the best results also on our environments.

By targeting different operating systems and different GPU architectures, our analysis aims to demonstrate how these variations affect algorithm performance.

B. Comparative Evaluation

Overall Comparison. To begin with, Tables II and III provide an overall comparison based on the average total execution time and the average throughput in MPixel/ms, targeting all the considered environments. Some algorithms lack a 3D implementation, so results on 3D volumes cannot be reported.

As previously noticed in [63] for CPU-based CCL implementations, the performance of the same algorithm run on different operating systems may be drastically different, especially when memory allocation becomes important, i.e., when a dataset contains high-resolution images, or the algorithm requires larger data structures. There might be plenty of reasons behind this performance gap, including the difference in virtual page commits that Windows and Linux have, or the fact that under Windows it is almost impossible to prevent the GUI from accessing the

TABLE II
RESULTS OBTAINED UNDER WINDOWS 10 RUNNING A QUADRO K2200 AND A QUADRO P1000 NVIDIA GPUS (SETUP A)

		2D Images														3D Volumes					
		3DPeS		*Fingerprints*		*Hamlet*		*Medical*		*MIRflickr*		*Tobacco800*		*XDOCS*		*Hilbert*		*Oasis*		*Mitochondria*	
(a) *Quadro K2200*	M8DLS	21.757	0.019	14.245	0.010	127.800	0.021	38.200	0.032	20.180	0.009	331.300	0.014	3150.550	0.005						
	RASMUSS.	5.537	0.074	4.428	0.032	117.100	0.023	9.169	0.132	5.463	0.033	249.293	0.018	2987.948	0.006						
	8DLS	3.341	0.123	7.139	0.020	37.988	0.071	18.028	0.067	34.826	0.005	185.789	0.025	4732.362	0.003						
	OLE	0.958	0.428	0.612	0.229	5.215	0.520	2.806	0.431	0.698	0.258	7.985	0.576	34.102	0.484						
	STAVA	0.829	0.495	0.620	0.226	4.077	0.665	2.682	0.451	0.916	0.197	5.532	0.832	21.853	0.755						
	LBUF*	0.602	0.681	0.306	0.457	3.214	0.843	1.978	0.612	0.293	0.613	4.423	1.040	18.105	0.911						
	DLP*	0.662	0.620	0.260	0.538	3.325	0.815	1.807	0.670	0.328	0.549	5.375	0.856	19.542	0.844						
	HA8*	0.599	0.685	0.228	0.615	2.746	0.987	1.493	0.810	0.223	0.806	4.160	1.106	14.976	1.101						
	KE*	0.565	0.726	0.254	0.551	2.929	0.925	1.735	0.697	0.260	0.693	4.207	1.093	16.954	0.973						
	C_SAUF*	0.531	0.772	0.234	0.598	2.834	0.956	1.585	0.763	0.254	0.708	3.891	1.182	15.522	1.062						
	C_BBDT*	0.514	0.798	0.224	0.624	2.424	1.118	1.318	0.918	0.240	0.749	3.461	1.329	13.087	1.260						
	C_DRAG*	0.506	0.810	0.215	0.652	2.402	1.128	1.263	0.958	0.209	0.862	3.407	1.350	12.797	1.289						
	BE*	1.020	0.402	0.694	0.202	3.781	0.717	2.029	0.596	0.924	0.195	5.283	0.871	18.765	0.879	4.786	2.622	9.540	327.982	90.104	4.320
	UF*	0.598	0.686	0.307	0.457	3.312	0.818	2.213	0.547	0.391	0.461	4.684	0.982	19.115	0.863	3.492	3.594	17.798	175.803	132.286	2.943
	BUF*	0.494	0.830	0.207	0.677	2.252	1.203	1.373	0.881	0.223	0.806	3.300	1.394	12.883	1.280	**2.105**	**5.962**	**6.481**	**482.788**	**67.326**	**5.782**
	BKE*	**0.487**	**0.842**	**0.196**	**0.714**	**2.127**	**1.274**	**1.216**	**0.995**	**0.178**	**1.010**	**3.214**	**1.431**	**11.688**	**1.411**	2.714	4.624	8.259	378.853	102.297	3.805
(b) *Quadro P1000*	M8DLS	13.109	0.031	8.583	0.016	77.058	0.035	23.067	0.052	12.163	0.015	199.626	0.023	1898.248	0.009						
	RASMUSS.	3.569	0.115	2.888	0.048	67.363	0.040	5.387	0.225	3.467	0.052	140.871	0.033	1667.878	0.010						
	8DLS	2.995	0.137	5.361	0.026	30.183	0.090	13.684	0.088	23.522	0.008	149.902	0.031	3888.757	0.004						
	OLE	0.991	0.414	0.482	0.290	4.390	0.617	2.244	0.539	0.568	0.317	6.718	0.685	28.536	0.578						
	STAVA	0.689	0.595	0.405	0.346	2.881	0.941	1.966	0.615	0.676	0.266	3.749	1.227	14.483	1.139						
	LBUF*	0.515	0.796	0.179	0.782	2.386	1.136	1.532	0.790	0.192	0.938	3.077	1.495	12.718	1.297						
	DLP*	0.544	0.754	0.120	1.167	2.183	1.241	1.228	0.985	0.177	1.017	3.344	1.376	12.023	1.372						
	HA8*	0.481	0.852	0.107	1.308	1.930	1.404	1.135	1.066	0.133	1.353	2.692	1.709	9.765	1.689						
	KE*	0.511	0.802	0.137	1.022	2.320	1.168	1.408	0.859	0.169	1.065	3.182	1.446	12.886	1.280						
	C_SAUF*	0.480	0.854	0.131	1.069	2.262	1.198	1.310	0.924	0.182	0.989	2.911	1.580	11.727	1.406						
	C_BBDT*	0.468	0.876	0.106	1.321	1.927	1.406	1.091	1.109	0.145	1.241	2.591	1.775	9.748	1.692						
	C_DRAG*	0.466	0.880	0.102	1.373	1.927	1.406	1.049	1.153	0.130	1.385	2.559	1.798	9.581	1.721						
	BE*	0.775	0.529	0.465	0.301	2.952	0.918	1.703	0.711	0.481	0.374	4.151	1.108	14.324	1.151	3.459	3.628	7.023	445.529	58.013	6.710
	UF*	0.483	0.849	0.156	0.897	2.311	1.173	1.568	0.772	0.237	0.759	3.083	1.492	12.498	1.319	2.379	5.275	13.939	224.474	95.871	4.060
	BUF*	0.450	0.911	0.101	1.386	1.766	1.535	1.120	1.080	0.146	1.233	2.433	1.891	9.496	1.737	**1.391**	**9.022**	**4.656**	**672.025**	**43.923**	**8.863**
	BKE*	**0.442**	**0.928**	**0.089**	**1.573**	**1.672**	**1.621**	**0.997**	**1.214**	**0.104**	**1.731**	**2.394**	**1.921**	**8.595**	**1.919**	1.753	7.159	5.605	558.243	65.939	5.904

White columns report average (per image) run-time results in ms (↓) while gray columns represent the average throughput of each algorithm in MPixel/ms (↑). The bold values represent the best algorithm on the specific environment/dataset. The * identifies direct algorithms.

GPU during tests. A thorough explanation of this phenomenon would require deep experiments and goes beyond the goal of the paper. The conclusion that deserves to be raised is that the execution time ratio Linux/Windows is almost constant for different algorithms, meaning that changing the operating system does not impact on algorithm behavior.

When focusing on 2D images, the overall performance of the algorithms and their mutual ranking are not affected by dataset intrinsic characteristics. Of course, the total execution time is deeply influenced by image size and complexity (i.e., the number of connected components it contains and their geometry). XDOCS is certainly the hardest dataset to label and it holds the higher average execution time.

Experimental results also reveal that the advantages of the "M" variation proposed for 8DLS are negligible on real-case datasets, unless images are very large and the majority of connected components is small, as it happens within XDOCS. In all the other cases, the additional checks required by M8DLS ruin the overall algorithm performance.

Another conclusion that can be drawn is that (M)8DLS and RASMUSSON algorithms are significantly slower than others, so they can be ignored in the rest of the analysis.

Different GPU architectures may also impact algorithm performance, changing the figures by an order of magnitudes, as it can be appreciated in Table III when comparing algorithms execution on Quadro GPUs with the newer A100 80 GB. Apart from the overall performance, related to computational capabilities, memory hierarchies, and bandwidths, each GPU architecture may have specific optimizations that can be applied to improve performance. However, none of the implementations analyzed with this paper and provided in YACCLAB explicitly leverage on GPU specific optimization and this is confirmed also by our experiments: selected a dataset, the execution time ratio of the algorithms run on different GPUs is constant. However, the performance gap between different algorithmic proposals is negligible when using newer architecture and smaller datasets. A deeper analysis is hence provided in the following of this section.

The abovementioned tables also suggest that, regardless of the operating system and the GPU architecture (at least for those considered in our experiments), BKE, currently the only GPU-based algorithm available in the OpenCV library [85], is always the best-performing algorithm on 2D datasets. For what concerns 3D volumes, BUF usually demonstrates better results, except when running on the A100 GPU where BKE is still competitive. The advantages provided by the use of a block-based approach, which allows for a considerable reduction in the number of memory accesses, are clearly demonstrated by the superior performance of BKE and BUF.

On the Role of Inline Compression (IC). The better results obtained by BUF compared to BKE on 3D datasets are explained by considering the IC optimization both the algorithms make use of (see Section II-A and Algorithm 1 as a reference). This optimization is intended for reducing the number of parent nodes a thread has to traverse when searching for the root of union-find equivalence trees, saving memory accesses and time. These savings come at the expense of additional write operations, which could nullify the improvement. The net improvement of the IC optimization strongly depends on the shape and dimension of the objects the input images contain, which affect the complexity of

TABLE III
RESULTS OBTAINED UNDER UBUNTU 22.04 LTS RUNNING A QUADRO K2200 AND A QUADRO P1000 NVIDIA GPUS (SETUP A) AND A NVIDIA A100 80 GB (SETUP B)

		2D Images														3D Volumes					
		3DPeS		*Fingerprints*		*Hamlet*		*Medical*		*MIRflickr*		*Tobacco800*		*XDOCS*		*Hilbert*		*Oasis*		*Mitochondria*	
(a) *Quadro K2200*	M8DLS	11.841	0.035	7.783	0.018	70.481	0.038	22.600	0.054	12.903	0.014	187.245	0.025	1764.260	0.009						
	RASMUSS.	5.197	0.079	4.078	0.034	118.739	0.023	9.010	0.134	5.171	0.035	256.770	0.018	3070.204	0.005						
	8DLS	2.825	0.145	6.616	0.021	37.695	0.072	17.632	0.069	34.233	0.005	185.426	0.025	4829.739	0.003						
	OLE	0.624	0.657	0.451	0.310	5.062	0.535	2.656	0.456	0.549	0.328	7.527	0.611	35.453	0.465						
	STAVA	0.620	0.661	0.556	0.252	3.780	0.717	2.393	0.506	0.866	0.208	4.945	0.930	19.490	0.846						
	LBUF*	0.391	1.049	0.250	0.560	2.910	0.931	1.712	0.707	0.260	0.692	3.855	1.193	15.815	1.043						
	DLP*	0.443	0.926	0.203	0.690	3.056	0.887	1.573	0.769	0.299	0.602	4.778	0.963	17.118	0.963						
	HA8*	0.388	1.057	0.170	0.824	2.412	1.124	1.218	0.993	0.187	0.963	3.588	1.282	12.445	1.325						
	KE*	0.346	1.185	0.198	0.707	2.590	1.046	1.454	0.832	0.228	0.789	3.642	1.263	14.388	1.146						
	C_SAUF*	0.315	1.302	0.180	0.778	2.487	1.090	1.312	0.922	0.222	0.811	3.264	1.409	12.830	1.285						
	C_BBDT*	0.315	1.302	0.182	0.769	2.134	1.270	1.040	1.163	0.218	0.826	2.901	1.586	11.021	1.496						
	C_DRAG*	0.305	1.344	0.171	0.819	2.092	1.295	1.014	1.193	0.187	0.963	2.859	1.609	10.580	1.559						
	BE*	0.682	0.601	0.431	0.325	3.305	0.820	1.644	0.736	0.563	0.320	4.324	1.064	16.366	1.008	3.703	3.389	7.941	394.025	49.011	7.943
	UF*	0.382	1.073	0.252	0.556	3.037	0.892	1.966	0.615	0.364	0.495	4.135	1.112	16.852	0.979	3.868	3.245	23.245	134.607	152.400	2.554
	BUF*	0.289	1.419	0.152	0.921	1.936	1.400	1.120	1.080	0.187	0.963	2.709	1.698	10.321	1.598	**1.601**	**7.839**	**5.300**	**590.368**	**29.098**	**13.378**
	BKE*	**0.279**	**1.470**	**0.140**	**1.000**	**1.807**	**1.500**	**0.947**	**1.278**	**0.143**	**1.259**	**2.609**	**1.763**	**9.153**	**1.802**	2.251	5.575	7.102	440.573	79.816	4.877
(b) *Quadro P1000*	M8DLS	12.077	0.034	7.764	0.018	69.362	0.039	20.556	0.059	11.196	0.016	179.319	0.026	1725.116	0.010						
	RASMUSS.	3.057	0.134	2.425	0.058	65.936	0.041	4.976	0.243	3.025	0.060	141.906	0.032	1713.080	0.010						
	8DLS	2.441	0.168	4.754	0.029	28.978	0.094	13.060	0.093	22.715	0.008	148.790	0.031	3938.968	0.004						
	OLE	0.667	0.615	0.345	0.406	4.159	0.652	2.051	0.590	0.427	0.422	6.330	0.727	29.798	0.553						
	STAVA	0.498	0.823	0.376	0.372	2.495	1.086	1.648	0.734	0.653	0.276	3.249	1.416	13.140	1.255						
	LBUF*	0.330	1.242	0.163	0.859	2.016	1.344	1.225	0.988	0.175	1.029	2.638	1.744	11.518	1.432						
	DLP*	0.359	1.142	0.102	1.373	1.817	1.491	0.942	1.285	0.165	1.091	2.903	1.585	10.804	1.526						
	HA8*	0.295	1.390	0.085	1.647	1.542	1.757	0.836	1.447	0.110	1.636	2.221	2.071	8.413	1.960						
	KE*	0.322	1.273	0.118	1.186	1.927	1.406	1.106	1.094	0.150	1.200	2.679	1.717	11.578	1.424						
	C_SAUF*	0.291	1.409	0.112	1.250	1.849	1.466	0.990	1.222	0.163	1.104	2.409	1.910	10.201	1.617						
	C_BBDT*	0.282	1.454	0.092	1.522	1.531	1.770	0.774	1.563	0.130	1.385	2.137	2.153	8.451	1.951						
	C_DRAG*	0.278	1.475	0.087	1.609	1.524	1.778	0.754	1.605	0.115	1.565	2.094	2.197	8.384	1.967						
	BE*	0.436	0.940	0.277	0.505	2.441	1.110	1.311	0.923	0.304	0.592	3.431	1.341	13.549	1.217	2.853	4.399	6.560	476.974	38.951	9.994
	UF*	0.293	1.399	0.140	1.000	1.937	1.399	1.280	0.945	0.224	0.804	2.651	1.735	11.238	1.467	2.950	4.254	21.114	148.193	124.371	3.130
	BUF*	0.258	1.589	0.081	1.728	1.356	1.999	0.821	1.474	0.125	1.440	1.978	2.326	8.248	1.999	**1.013**	**12.389**	**4.094**	**764.277**	**25.386**	**15.334**
	BKE*	**0.253**	**1.621**	**0.067**	**2.090**	**1.266**	**2.141**	**0.689**	**1.756**	**0.081**	**2.222**	**1.920**	**2.396**	**7.204**	**2.289**	1.414	8.876	5.256	595.310	50.803	7.663
(c) *A100 80GB*	M8DLS	1.950	0.210	1.865	0.075	13.292	0.204	3.386	0.357	2.798	0.064	21.286	0.216	303.517	0.054						
	RASMUSS.	0.519	0.790	0.600	0.233	5.759	0.471	0.593	2.040	0.732	0.246	15.913	0.289	121.323	0.136						
	8DLS	0.835	0.491	1.302	0.108	2.997	0.904	0.806	1.501	3.015	0.060	11.397	0.404	161.845	0.102						
	OLE	0.202	2.030	0.145	0.966	0.628	4.315	0.319	3.793	0.160	1.125	0.885	5.198	2.750	5.996						
	STAVA	0.211	1.943	0.200	0.700	0.755	3.589	0.552	2.192	0.327	0.550	0.971	4.737	2.404	6.859						
	LBUF*	0.099	4.141	0.041	3.415	0.377	7.188	0.213	5.681	0.044	4.091	0.533	8.630	0.933	17.674						
	DLP*	0.103	3.981	0.041	3.415	0.330	8.212	0.198	6.111	0.092	1.957	0.546	8.425	1.378	11.967						
	HA8*	0.117	3.504	0.051	2.745	0.403	6.725	0.219	5.525	0.068	2.647	0.567	8.113	0.906	18.201						
	KE*	0.096	4.271	0.038	3.684	0.347	7.810	0.204	5.931	0.048	3.750	0.539	8.534	0.957	17.231						
	C_SAUF*	0.098	4.184	0.038	3.684	0.347	7.810	0.188	6.436	0.056	3.214	0.499	9.218	0.793	20.794						
	C_BBDT*	0.103	3.981	0.041	3.415	0.316	8.576	0.181	6.685	0.056	3.214	0.502	9.163	0.751	21.957						
	C_DRAG*	0.103	3.981	0.040	3.500	0.336	8.065	0.181	6.685	0.053	3.396	0.487	9.446	0.680	24.250						
	BE*	0.210	1.952	0.165	0.848	0.681	3.979	0.421	2.874	0.170	1.059	0.890	5.169	2.194	7.516	0.731	17.168	0.968	3232.386	4.520	86.124
	UF*	0.100	4.100	0.039	3.590	0.381	7.113	0.222	5.450	0.046	3.913	0.551	8.348	1.041	15.841	0.345	36.377	1.339	2336.781	6.500	59.889
	BUF*	0.102	4.020	0.041	3.415	0.335	8.090	0.197	6.142	0.061	2.951	0.515	8.932	0.740	22.284	0.209	60.048	0.593	5276.476	**2.020**	**192.713**
	BKE*	**0.094**	**4.362**	**0.035**	**4.000**	**0.331**	**8.187**	**0.184**	**6.576**	**0.038**	**4.737**	**0.465**	**9.892**	**0.621**	**26.554**	**0.161**	**77.950**	**0.558**	**5607.437**	3.700	105.211

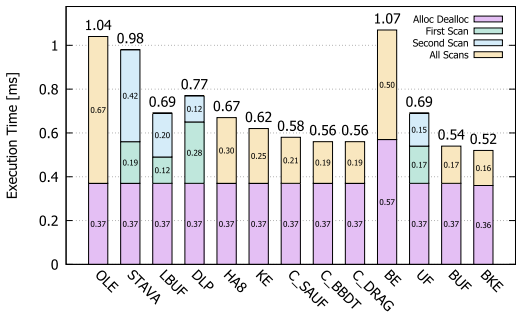
White columns report average (per image) run-time results in ms (↓) while gray columns represent the average throughput of each algorithm in MPixel/ms (↑). The bold values represent the best algorithm on the specific environment/dataset. The * identifies direct algorithms.

the equivalence trees created and updated during *Initialization* and *Merge/Reduction* phases. As already proven in [36], the use of IC always improves BUF and BKE performance on 2D datasets. Instead, when dealing with 3D real-case volumes, the nature of the data usually translates into shorter equivalence trees, causing the IC to hardly save any memory read. In this scenario, the IC may increase the total execution time due to the additional writes. Since BKE performs the `Compression` kernel twice, this limitation has a greater (negative) impact on such an algorithm, making it slower than BUF on the 3D datasets, at least on older GPUs.

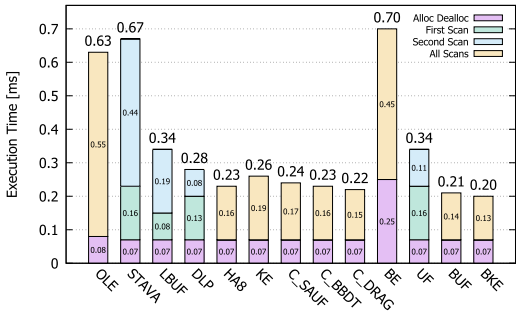
The Effects of Memory Allocation. The remaining algorithms have more similar execution times. For this reason, in order to have a deeper understanding of their behavior, Figs. 8 and 9 are reported for 2D and 3D datasets respectively. In these charts, the time needed for allocating data structures is separated from the one required by the global labeling procedure. Moreover, if an algorithm employs two clearly distinct phases to compute local labeling and perform tiles merging, the time required by each of them is displayed separately. The allocation time is the same for each strategy, except for BE and OLE which require additional data structures to the output image. OLE requires an additional byte to check whether the convergence has been reached or not. The time required by `cudaMalloc` is not perfectly linear with respect to the amount of memory requested: in some cases, the additional allocation required for this single byte considerably increases the elapsed time. On the other hand, BE relies on additional matrices to store block adjacency and labels, always requiring a data-dependent higher allocation time.

The experiments clearly demonstrate that allocating and freeing device memory represents a significant portion of the CCL total elapsed time, and therefore it should be avoided whenever possible, e.g., in embedded applications or whenever the input image size is fixed.

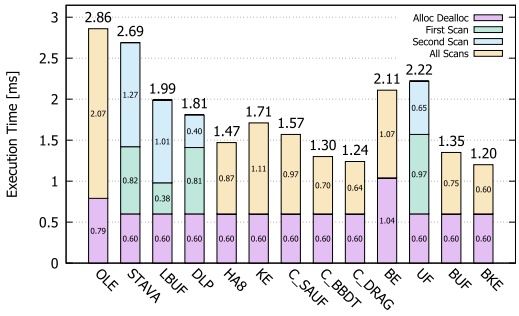
Iterative versus Direct Algorithms. The multiple iterations over the input image required by both OLE and STAVA make them the worst algorithms in most scenarios. The block scan approach proposed by BE reduces the operations (and thus the time) required by the global labeling with respect to OLE at



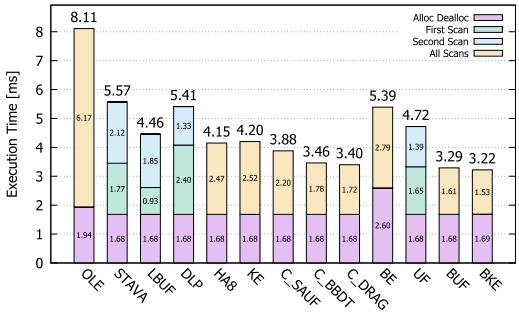
(a) 3DPeS



(b) Fingerprints



(c) Medical



(d) Tobacco800

Fig. 8. Average run-time with steps test on 2D datasets under Windows using NVIDIA QUADRO K2200 (Setup A). Numbers are given in ms. Lower is better. Best viewed online.

the expense of a longer allocation step. When the number of connected components is small, as it happens in 3DPeS and Fingerprints, the benefit of using additional data structures is annihilated by the allocation time.

Direct algorithms, which perform a fixed number of kernel calls, tend to have better performance than iterative ones. UF is faster than OLE, STAVA, and BE on all datasets but Medical, and the same goes for its optimization, LBUF and KE, the former using better block size and the latter with a more efficient initialization phase. DLP has a very similar approach to UF, since it also heavily relies on union-find, even implementing it slightly differently; unsurprisingly, its performance is comparable to UF.

The block-based approach proposed with BUF and BKE allows to leverage the advantages of blocks to further improve UF and KE, without introducing allocation drawbacks, making the two algorithms the fastest available in literature for both 2D and 3D scenarios. As already mentioned, BKE has been available in OpenCV since version 4.6.0 for labeling connected components on GPU devices.

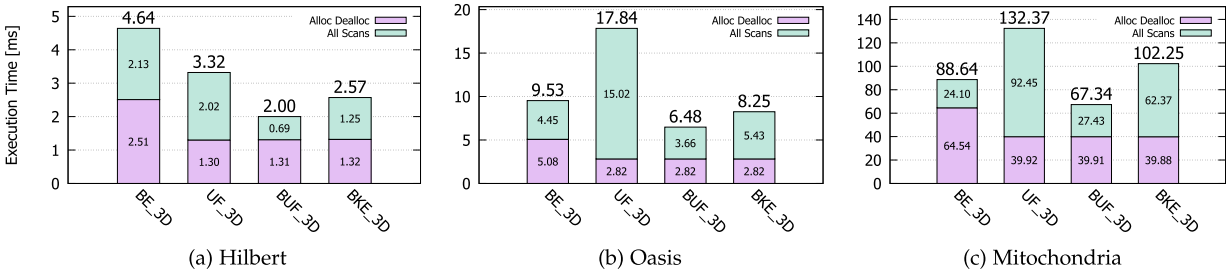
How Does Decision Trees Perform on GPUs? The algorithms based on decision trees, i.e., C_SAUF, C_BBDT, and C_DRAG, have performance close to the state of the art, with C_DRAG always being the best among the three. Although irregular patterns of execution slow down the thread scheduling and make the decision tree the worst thing to use in GPUs, when working with "natural" images the patterns within blocks are more or less uniform, making the thread divergence not really frequent. This is also confirmed by the granularity experiments reported in Fig. 10. When granularity is 1, the computational time of both C_BBDT and C_DRAG explodes around 50% of density, in accordance with expectations related to branch divergence. These algorithms have performance close to state-of-the-art GPU-specific designs only when density is below 10% or over 90%. Indeed, when foreground densities are in such ranges the action to be performed on the current pixel is selected in the first levels of the decision trees employed by the algorithms, saving many memory accesses without ruining the thread execution flow. Since images on which CCL is usually applied are in that range of densities, the performance of C_SAUF, C_BBDT, and C_DRAG highlighted in the previous charts are easily explainable taking into account granularity tests. Additionally, it can be noticed that as the granularity increases (Fig. 10(b), 10(c), and (d)), smaller portions of the image have irregular patterns and require to explore deeper tree levels, while the vast majority tends to be all white or all black, again minimizing thread divergence and maximizing the tree performance.

About Foreground Density. Thanks to an efficient combination of CUDA intrinsics to find runs of foreground pixels and union-find operations to merge runs of the same CC, HA8 performs close to state of the art on many datasets, especially when connected components extend in the horizontal direction (Fingerprints).

Taking into account Fig. 10, we can also notice that the performance trend of UF and BUF when varying the foreground density is similar although shifted along the y-axis since they share the same basic structure. A similar conclusion can be drawn when comparing KE and BKE.

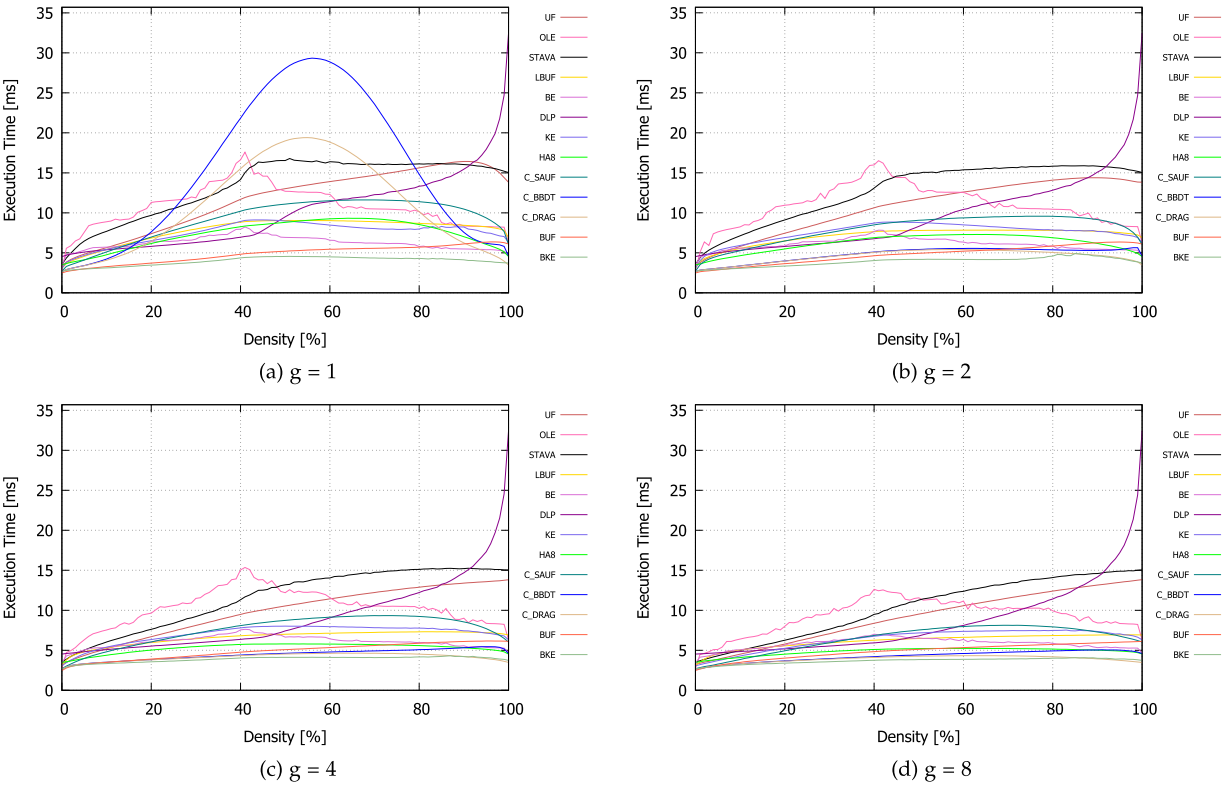
The disadvantages of iterative approaches that do not leverage "auxiliary" data structures to record and store label equivalences are clearly visible with DLP. Independently of the granularity, the execution time of DLP (almost) exponentially increases with the number of foreground pixels.

For the sake of completeness, Fig. 11 depicts the performance of the 3D algorithms when tested over randomly generated



(a) Hilbert (b) Oasis (c) Mitochondria

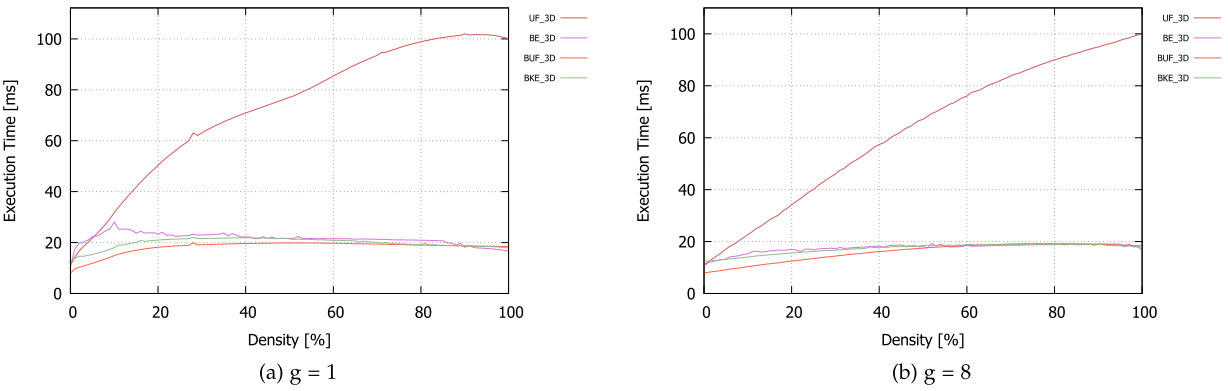
Fig. 9. Average run-time with steps test on 3D datasets. The experiment ran under Windows with NVIDIA QUADRO K2200 (Setup A). Lower is better. Best viewed online.



(a) g = 1 (b) g = 2

(c) g = 4 (d) g = 8

Fig. 10. Granularity tests on 2D randomly generated images. Numbers are given in ms. Lower is better. Best viewed online.



(a) g = 1 (b) g = 8

Fig. 11. Granularity tests on 3D randomly generated images (Setup A). Numbers are given in ms. Lower is better. Best viewed online.

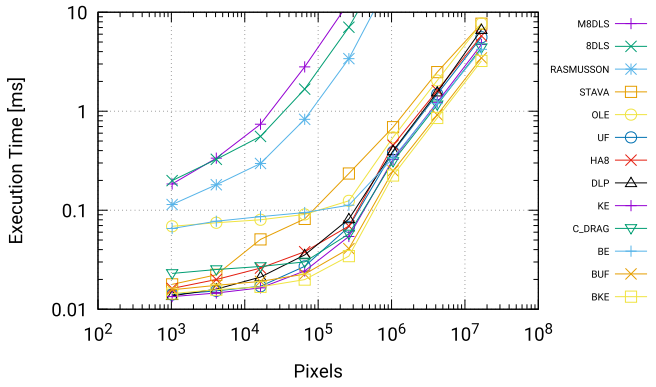


Fig. 12. Correlation between algorithm execution time and the growth of input size (Setup B). Lower is better. Best viewed online.

three-dimensional volumes. The same considerations drawn for the two-dimensional cases can be applied. Moreover, it can be highlighted that the previously discussed negative effects of IC on BKE become less relevant as density increases. The performance of BKE and BUF are almost the same above $\sim$70% and $\sim$50% of foreground density, respectively with granularity 1 and 8. When granularity is 1, storing the block adjacency information in a separate data structure becomes valuable —i.e., the additional memory allocation time is compensated by the faster iterative scan approach— only if foreground density is higher than 90%, making BE slightly faster than both BUF and BKE in such scenario.

On the Effects of the Input Size. To complete our analysis, the effect of input size on execution time is depicted in Fig. 12. In this experiment, each algorithm is tested on random images with a foreground density of 0.5 and a size increasing from 10^3 to 10^7 pixels. The granularity is set to 1. In order to simplify the chart, C_DRAG has been chosen as the representative of C_BBDT and C_SAUF; the three algorithms employ the same core approach and have comparable performances. To focus on interesting algorithms, the y-axis has been cropped.

Confirming previous conclusions, 8DLS, M8DLS, and RASMUSSON perform much worse than other algorithms, regardless of the input dimension. These algorithms require multiple iterations that are inefficient.

When the input image contains few pixels (e.g., $< 10^5$), the execution time is dominated by the allocation time of the output image and auxiliary data structures. In this regard, we can say that all the algorithms have similar performance, except for BE and OLE which need to allocate more memory, degrading the overall performance.

However, with larger input sizes the running time of all the algorithms grows (almost) linearly. As already noticed in Fig. 11, BE becomes more competitive as the complexity (the size in this case) of the input image increases, compensating for the additional allocation time.

In this experiment, C_DRAG is significantly slower than other direct algorithms, because the input image density of 0.5 causes the maximum branch divergence, as already discussed with granularity tests.

VI. Concluding Remarks and Future Work

This article reviewed the state-of-the-art connected components labeling algorithms designed for GPUs and published in the last decade, focusing on the main strategies adopted and analyzing their performance in different scenarios. Experimental results conducted on different operating systems, using different compilers and GPU architectures revealed that BKE is the best-performing algorithm for 2D datasets including text images, fingerprints, medical images, and surveillance datasets. When working on 3D volumes the performance of BKE is still impressive, but BUF demonstrates generally better capabilities, especially when working with hard-to-label volumes such as the Mitochondria dataset. We also showed that algorithms specifically designed for sequential architecture can be extremely effective when adapted to GPUs.

The source code of all the analyzed algorithms is collected in the YACCLAB benchmark and is provided along with this article [62]. It is worth mentioning that we have been forced to re-implement many of the algorithm solutions published in literature since the original implementation was not publicly available and some authors did not answer to our explicit requests. In any case, the code is now public so anyone can verify that our implementations are aligned with the description in the related publications.

If we consider that the allocation time is "fixed" for a given environment and that the bare minimum of any CCL algorithm is the copy of the input into the output image (an operation that is not enough to complete the task, but strictly required) the reader can certainly spot that space for further optimization is really small, extremely so.

Future work should focus on the Connected Components Analysis (CCA) by integrating the calculation of connected components statistics in state-of-the-art solutions.

Another open research direction is the optimization of CCL on embedded systems, including FPGA. Although specific algorithmic solutions should be devised, reusing the core innovation of GPU-based solutions could benefit also such scenarios.

References

[1] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors," *J. Real-Time Image Process.*, vol. 15, no. 1, pp. 3486–3489, 2018.

[2] P. Asad, R. Marroquim, and A. L. E. L. Souza, "On GPU connected components and properties: A systematic evaluation of connected component labeling algorithms and their extension for property extraction," *IEEE Trans. Image Process.*, vol. 28, no. 1, pp. 17–31, Jan. 2019.

[3] A. Dubois and F. Charpillet, "Tracking mobile objects with several kinects using HMMs and component labelling," in *Proc. Workshop Assistance Serv. Robot. Hum. Environ. Int. Conf. Intell. Robots Syst.*, 2012.

[4] C. Zhan, X. Duan, S. Xu, Z. Song, and M. Luo, "An improved moving object detection algorithm based on frame difference and edge detection," in *Proc. 4th Int. Conf. Image Graph.*, 2007, pp. 519–523.

[5] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-time image segmentation on a GPU," in *Facing the Multicore-Challenge*. Berlin, Germany: Springer, 2010.

[6] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Improving skin lesion segmentation with generative adversarial networks," in *Proc. IEEE 31st Int. Symp. Comput.-Based Med. Syst.*, 2018, pp. 442–443.

[7] A. Körbes, G. B. Vitor, R. de Alencar Lotufo, and J. V. Ferreira, "Advances on watershed processing on GPU architecture," in *Proc. Int. Symp. Math. Morphol. Appl. Signal Image Process.*, Springer, 2011, pp. 260–271.

[8] E. Gomel, T. Shaharbany, and L. Wolf, "Box-based refinement for weakly supervised and unsupervised localization tasks," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2023, pp. 15998–16008.

[9] Y. Zhang, S. Wu, N. Snavely, and J. Wu, "Seeing a rose in five thousand ways," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2023, pp. 962–971.

[10] I. H. Laradji, N. Rostamzadeh, P. O. Pinheiro, D. Vazquez, and M. Schmidt, "Where are the blobs: Counting by localization with point supervision," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 560–576.

[11] A. Eklund, P. Dufort, M. Villani, and S. LaConte, "BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs," *Front. Neuroinform.*, vol. 8, 2014, Art. no. 24.

[12] H. V. Pham, B. Bhaduri, K. Tangella, C. Best-Popescu, and G. Popescu, "Real time blood testing using quantitative phase imaging," *PLoS One*, vol. 8, no. 2, 2013, Art. no. e55676.

[13] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Augmenting data with GANs to segment melanoma skin lesions," *Multimedia Tools Appl.*, vol. 79, pp. 15575–15592, 2020.

[14] G. Bontempo, A. Porrello, F. Bolelli, S. Calderara, and E. Ficarra, "DAS-MIL: Distilling across scales for MIL classification of histological WSIs," in *Proc. Int. Conf. Med. Image Comput. Comput. Assist. Interv.*, 2023, pp. 248–258.

[15] B. E. Bejnordi et al., "Diagnostic assessment of deep learning algorithms for detection of lymph node metastases in women with breast cancer," *JAMA*, vol. 318, no. 22, pp. 2199–2210, 2017.

[16] T. Lelore and F. Bouchara, "FAIR: A fast algorithm for document image restoration," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 2039–2048, Aug. 2013.

[17] F. Bolelli, "Indexing of historical document images: Ad Hoc Dewarping technique for handwritten text," in *Proc. Italian Res. Conf. Digit. Libraries*, Springer, 2017, pp. 45–55.

[18] T. Berka, "The generalized feed-forward loop motif: Definition, detection and statistical significance," *Procedia Comput. Sci.*, vol. 11, pp. 75–87, 2012.

[19] M. J. Dinneen, M. Khosravani, and A. Probert, "Using OpenCL for implementing simple parallel graph algorithms," in *Proc. Int. Conf. Parallel Distrib. Process. Techn. Appl.*, 2011.

[20] C. Grana, L. Baraldi, and F. Bolelli, "Optimized connected components labeling with pixel prediction," in *Proc. 17th Int. Conf. Adv. Concepts Intell. Vis. Syst.*, 2016, pp. 431–440.

[21] F. Bolelli, M. Cancilla, and C. Grana, "Two more strategies to speed up connected components labeling algorithms," in *Proc. Int. Conf. Image Anal. Process.*, 2017, pp. 48–58.

[22] W. Lee, S. Allegretti, F. Bolelli, and C. Grana, "Fast run-based connected components labeling for bitonal images," in *Proc. Joint 10th Int. Conf. Inform. Electron. Vis. 5th Int. Conf. Imag. Vis. Pattern Recognit.*, 2021, pp. 1–8.

[23] F. Bolelli, S. Allegretti, and C. Grana, "Connected components labeling on bitonal images," in *Proc. Int. Conf. Image Anal. Process.*, 2022, pp. 347–357.

[24] A. Hennequin, L. Lacassagne, L. Cabaret, and Q. Meunier, "A new direct connected component labeling and analysis algorithms for GPUs," in *Proc. Conf. Des. Architectures Signal Image Process.*, 2018, pp. 76–81.

[25] L. Cabaret, L. Lacassagne, and D. Etiemble, "Distanceless label propagation: An efficient direct connected component labeling algorithm for GPUs," in *Proc. 7th Int. Conf. Image Process. Theory Tools Appl.*, 2017, pp. 1–6.

[26] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1217–1230, Jun. 2018.

[27] L. He, X. Zhao, Y. Chao, and K. Suzuki, "Configuration-transition-based connected-component labeling," *IEEE Trans. Image Process.*, vol. 23, no. 2, pp. 943–951, Feb. 2014s.

[28] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognit.*, vol. 42, no. 9, pp. 1977–1987, 2009.

[29] D. Windisch, C. Kaever, G. Juckeland, and A. Bieberle, "Parallel algorithm for connected-component analysis using CUDA," *Algorithms*, vol. 16, no. 2, 2023, Art. no. 80.

[30] M. Kowalczyk, P. Ciarach, D. Przewlocka-Rus, H. Szolc, and T. Kryjak, "Real-time FPGA implementation of parallel connected component labelling for a 4K video stream," *J. Signal Process. Syst.*, vol. 93, no. 5, pp. 481–498, 2021.

[31] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognit.*, vol. 70, pp. 25–43, 2017.

[32] A. Rosenfeld and A. Kak, *Digital Picture Processing*. New York, NY, USA: Academic, 1982.

[33] D. Zhang, H. Ma, and L. Pan, "A gamma-signal-regulated connected components labeling algorithm," *Pattern Recognit.*, vol. 91, pp. 281–290, 2019.

[34] F. Bolelli, S. Allegretti, and C. Grana, "One DAG to rule them all," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 7, pp. 3647–3658, Jul. 2022.

[35] F. Lemaitre, A. Hennequin, and L. Lacassagne, "How to speed connected component labeling up with SIMD RLE algorithms," in *Proc. 6th Workshop Program. Models SIMD/Vector Process.*, 2020, Art. no. 2.

[36] S. Allegretti, F. Bolelli, and C. Grana, "Optimized block-based algorithms to label connected components on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 423–438, Feb. 2020.

[37] H. Zhou, R. Dou, L. Cheng, J. Liu, and N. Wu, "A provisional labels-reduced, real-time connected component labeling algorithm for edge hardware," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 6, pp. 2997–3001, Jun. 2022.

[38] Z. Li, Q. Zhang, T. Long, and B. Zhao, "A parallel pipeline connected-component labeling method for on-orbit space target monitoring," *J. Syst. Eng. Electron.*, vol. 33, no. 5, pp. 1095–1107, Oct. 2022.

[39] K. Bok, N. Kim, D. Choi, J. Lim, and J. Yoo, "Incremental connected component detection for graph streams on GPU," *Electronics*, vol. 12, no. 6, 2023, Art. no. 1465.

[40] H. Samet, "Connected component labeling using quadtrees," *J. ACM*, vol. 28, no. 3, pp. 487–501, 1981.

[41] V. M. Oliveira and R. A. Lotufo, "A study on connected components labeling algorithms using GPUs," in *Proc. 23rd SIBGRAPI Conf. Graph. Patterns Images*, 2010.

[42] C. Grana, D. Borghesani, and R. Cucchiara, "Optimized block-based connected components labeling with decision trees," *IEEE Trans. Image Process.*, vol. 19, no. 6, pp. 1596–1609, Jun. 2010.

[43] D. J. C. Santiago, T. I. Ren, G. D. Cavalcanti, and T. I. Jyh, "Efficient 2×2 block-based connected components labeling algorithms," in *Proc. IEEE Int. Conf. Image Process.*, 2015, pp. 4818–4822.

[44] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti labeling: Directed acyclic graphs for block-based connected components labeling," *IEEE Trans. Image Process.*, vol. 29, pp. 1999–2012, 2020.

[45] S. Zavalishin, I. Safonov, Y. Bekhtin, and I. Kurilin, "Block equivalence algorithm for labeling 2D and 3D images on GPU," *Electron. Imag.*, vol. 2016, no. 2, pp. 1–7, 2016.

[46] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, "How does connected components labeling with decision trees perform on GPUs?," in *Proc. Int. Conf. Comput. Anal. Images Patterns*, 2019, pp. 39–51.

[47] C. Grana, F. Bolelli, L. Baraldi, and R. Vezzani, "YACCLAB - Yet another connected components labeling benchmark," in *Proc. 23rd Int. Conf. Pattern Recognit.*, 2016, pp. 3109–3114.

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

[49] K. Yonehara and K. Aizawa, "A line-based connected component labeling algorithm using GPUs," in *Proc. 3rd Int. Symp. Comput. Netw.*, 2015, pp. 341–345.

[50] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with GPUs and CUDA," *Parallel Comput.*, vol. 36, no. 12, pp. 655–678, 2010.

[51] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2D grid using CUDA," *J. Parallel Distrib. Comput.*, vol. 71, no. 4, pp. 615–620, 2011.

[52] O. Šťava and B. Beneš, "Connected component labeling in CUDA," in *GPU Computing Gems Emerald Edition*, W.-M. W. Hwu Ed., Amsterdam, The Netherlands: Elsevier, 2011, ch. 35.

[53] P. Chen, H. Zhao, C. Tao, and H. Sang, "Block-run-based connected component labelling algorithm for GPGPU using shared memory," *Electron. Lett.*, vol. 47, pp. 1309–1311, Nov. 2011.

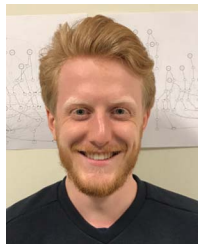
[54] Y. Soh, H. Raja, Y. Hae, and I. Kim, "Fast parallel connected component labeling algorithms using CUDA based on 8-directional label selection," *Int. J. Latest Res. Sci. Technol.*, vol. 3, pp. 187–190, Mar. 2014.

[55] A. Rasmusson, T. Sørensen, and G. Ziegler, "Connected components labeling on the GPU with generalization to Voronoi diagrams and signed distance fields," in *Proc. Int. Symp. Vis. Comput.*, 2013, pp. 206–215.

[56] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, "Optimizing GPU-Based connected components labeling algorithms," in *Proc. IEEE Int. Conf. Image Process. Appl. Syst.*, 2018, pp. 175–180.

[57] Y. Komura, "GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen–Wang multi-cluster spin flip algorithm," *Comput. Phys. Commun.*, vol. 194, pp. 54–58, 2015.
[58] F. Nina Paravecino and D. Kaeli, "Accelerated connected component labeling using CUDA framework," in *Proc. Int. Conf. Comput. Vis. Graph.*, 2014, pp. 502–509.
[59] K. Wu, E. Otoo, and K. Suzuki, "Two strategies to speed up connected component labeling algorithms," Lawrence Berkeley Nat. Lab., Berkeley, CA, USA, Tech. Rep. LBNL-59102, 2005.
[60] C. Grana, M. Montangero, and D. Borghesani, "Optimal decision trees for local image processing algorithms," *Pattern Recognit. Lett.*, vol. 33, no. 16, pp. 2302–2310, 2012.
[61] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected components labeling on DRAGs," in *Proc. 24th Int. Conf. Pattern Recognit.*, 2018, pp. 121–126.
[62] The YACCLAB Benchmark. Accessed: Dec. 21, 2022. [Online]. Available: https://github.com/prittt/YACCLAB
[63] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Toward reliable experiments on the performance of connected components labeling algorithms," *J. Real-Time Image Process.*, vol. 17, pp. 229–244, 2020.
[64] J. Chen, K. Nonaka, H. Sankoh, R. Watanabe, H. Sabirin, and S. Naito, "Efficient parallel connected component labeling with a coarse-to-fine strategy," *IEEE Access*, vol. 6, pp. 55731–55740, 2018.
[65] T. Chabardès, P. Dokládal, and M. Bilodeau, "A labeling algorithm based on a forest of decision trees," *J. Real-Time Image Process.*, vol. 17, no. 5, pp. 1527–1545, 2020.
[66] F. Diaz-del Rio, P. Sanchez-Cuevas, H. Molina-Abril, and P. Real, "Parallel connected-component-labeling based on homotopy trees," *Pattern Recognit. Lett.*, vol. 131, pp. 71–78, 2020.
[67] A. Torralba and A. A. Efros, "Unbiased look at dataset bias," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2011, pp. 1521–1528.
[68] W.-Y. Chang and C.-C. Chiu, "An efficient scan algorithm for block-based connected component labeling," in *Proc. 22nd Mediterranean Conf. Control Automat.*, 2014, pp. 1008–1013.
[69] M. J. Huiskes and M. S. Lew, "The MIR Flickr retrieval evaluation," in *Proc. ACM Int. Conf. Multimedia Inf. Retrieval*, 2008, pp. 39–43.
[70] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Trans. Syst., Man, Cybern.*, vol. 9, no. 1, pp. 62–66, Jan. 1979.
[71] F. Dong et al., "Computational pathology to discriminate benign from malignant intraductal proliferations of the breast," *PLoS One*, vol. 9, no. 12, 2014, Art. no. e114885.
[72] The Hamlet Dataset. Accessed: Mar. 21, 2019. [Online]. Available: http://www.gutenberg.org
[73] G. Agam, S. Argamon, O. Frieder, D. Grossman, and D. Lewis, "The complex document image processing (CDIP) test collection project," Illinois Inst. Technol., Chicago, IL, USA, 2006.
[74] D. Lewis, G. Agam, S. Argamon, O. Frieder, D. Grossman, and J. Heard, "Building a test collection for complex document information processing," in *Proc. 29th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2006, pp. 665–666.
[75] The Legacy Tobacco Document Library (LTDL), Univ. California, San Francisco, CA, USA, 2007.
[76] F. Bolelli, G. Borghi, and C. Grana, "Historical handwritten text images word spotting through sliding window hog features," in *Proc. 19th Int. Conf. Image Anal. Process.*, 2017, pp. 729–738.
[77] F. Bolelli, G. Borghi, and C. Grana, "XDOCS: An application to index historical documents," in *Proc. 14th Ital. Res. Conf. Digit. Libraries*, 2018, pp. 151–162.
[78] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*. Berlin, Germany: Springer, 2009.
[79] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern Recognit.*, vol. 33, no. 2, pp. 225–236, 2000.
[80] D. Baltieri, R. Vezzani, and R. Cucchiara, "3DPeS: 3D people dataset for surveillance and forensics," in *Proc. Joint ACM Workshop Hum. Gesture Behav. Understanding*, 2011, pp. 59–64.
[81] D. S. Marcus, A. F. Fotenos, J. G. Csernansky, J. C. Morris, and R. L. Buckner, "Open access series of imaging studies (OASIS): Longitudinal MRI data in nondemented and demented older adults," *J. Cogn. Neurosci.*, vol. 22, no. 12, pp. 2677–2684, 2010.
[82] The Electron Microscopy Dataset. Accessed: Mar. 21, 2019. [Online]. Available: https://cvlab.epfl.ch/data/data-em/
[83] A. Lucchi, Y. Li, and P. Fua, "Learning for structured prediction using approximate subgradient descent with working sets," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2013, pp. 1987–1994.
[84] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.
[85] G. Bradski, "The OpenCV library," *Dr. Dobb's J. Softw. Tools Professional Programmer*, vol. 25, pp. 120–123, 2000.

Federico Bolelli (Member, IEEE) received the BSc and MSc degrees in computer engineering from the Università degli Studi di Modena e Reggio Emilia, Italy, and the PhD degree from the Università degli Studi di Modena e Reggio Emilia, Italy, where he is currently working as a tenure track assistant professor within the AImageLab Group, Dipartimento di Ingegneria "Enzo Ferrari". His research interests include image processing, algorithms and optimization, and medical imaging. He is currently involved in a H2020 European Projects.



Stefano Allegretti received the BSc and MSc degrees in computer engineering from the Università degli Studi di Modena e Reggio Emilia, Italy. He is currently working toward the PhD degree with the AImagelab Laboratory, Dipartimento di Ingegneria "Enzo Ferrari", Università degli Studi di Modena e Reggio Emilia. His research interests include algorithm optimization and image processing.

Luca Lumetti received the BSc and MSc degrees in computer engineering from the Università degli Studi di Modena e Reggio Emilia, Italy. He is currently working toward the PhD degree with the AImageLab Group, Università degli Studi di Modena e Reggio Emilia, Italy. His research interests regard artificial intelligence, computer vision, and medical imaging.

Costantino Grana (Member, IEEE) received the graduate degree from the Università degli Studi di Modena e Reggio Emilia, Italy, in 2000, and the PhD degree in computer science and engineering, in 2004. He is currently full professor with the Dipartimento di Ingegneria "Enzo Ferrari", Università degli Studi di Modena e Reggio Emilia, Italy. His research interests are mainly in medical imaging, optimization of image processing algorithms, and computer vision applications. He published six book chapters, 47 papers on international peer reviewed journals and more than 130 papers on international conferences.

Open Access funding provided by 'Università degli Studi di Modena e Reggio Emilia' within the CRUI-CARE Agreement