Quest for Speed: The Epic Saga of Record-Breaking on OpenCV Connected Components Extraction

Federico Bolelli, Stefano Allegretti, and Costantino Grana(✉)

Dipartimento di Ingegneria "Enzo Ferrari",
Università degli Studi di Modena e Reggio Emilia, Modena, Italy
{federico.bolelli,stefano.allegretti,costantino.grana}@unimore.it

Abstract. Connected Components Labeling (CCL) represents an essential part of many Image Processing and Computer Vision pipelines. Given its relevance on the field, it has been part of most cutting-edge Computer Vision libraries. In this paper, all the algorithms included in the OpenCV during the years are reviewed, from sequential to parallel/GPU-based implementations. Our goal is to provide a better understanding of what has changed and why one algorithm should be preferred to another both in terms of memory usage and execution speed.

Keywords: OpenCV · Connected Components Labeling

1 Introduction

OpenCV (Open Source Computer Vision Library) is a software library mainly aimed at real-time computer vision [37]. Originally developed by Intel, it was later supported by Willow Garage and then Itseez. The library is cross-platform and free for use under the open-source Apache 2 License. Starting with 2011, OpenCV features GPU acceleration for real-time operations.

A common basic task in image processing is to produce a description of the objects inside a binary image; this is often done by extracting its connected components. By considering the pixel lattice as a graph in which foreground pixels are nodes connected by edges to their foreground neighbors, a connected component on the graph corresponds to the common definition of an "object of interest". Based on the specific use case, two pixels can be considered connected or not, according to the definition of pixel connectivity: in 2D-images, pixels can be either *4-connected* (sides only) or *8-connected* (sides and corners). A possible solution to extract connected components (objects) is to use a Connected Components Labeling (CCL) algorithm: a procedure which generates a symbolic image in which each pixel of a single connected component is assigned a unique identifier.

The CCL algorithm has an exact output meaning that different algorithmic solutions should be mainly compared in term of speed and memory footprint.

P. L. Mazzeo et al. (Eds.): ICIAP 2022 Workshops, LNCS 13374, pp. 107–118, 2022.
https://doi.org/10.1007/978-3-031-13324-4_10

After the introduction of the task in 1966 [38], several proposals to optimize its computational load have been published for both sequential [7,9,14,15,20,22,24,25,28,31,42] and parallel architectures [1–4,29,35,43], taking into account also 3D volumes [8,27,40].

Connected Components Analysis, or CCA in short, extends CCL by computing some features of the connected components such as their bounding box, their area, or the first moments to compute center of gravity. CCA is basically a voting algorithm like histogram computation or Hough transform [30] and it is a mandatory step for many Computer Vision and Image Processing pipelines [5,13,17,18,23,32,34,36,41].

Connected Components extraction has been available since the early days of OpenCV and has evolved (in speed) with every release. Initially, the implementation available was based on the combination of two different functions: `FindContours` and `DrawContours`, respectively in charge of retrieving contours and the hierarchical information from binary images and drawing them. Since version *3.0.0*, `cv::connectedComponents` and `cv::connectedComponentsWithStats` APIs have been introduced, providing a major speed breakthrough for CCL computation within the library.

The goal of this paper is to review all the algorithms implemented in OpenCV during the years, thus providing the reader with a better understanding of what has changed and why one should choose one algorithm rather than another both in terms of memory usage and execution speed.

2 The First Approach

The extraction of Connected Components (CCs) from a binary image has been available since the first release of the OpenCV with the combination of `findContours` and `drawContours` functions (Listing 1.1).

`findContours` operates on a binary image by retrieving objects' contours. The function retrieves contours from the binary image using the algorithm described in [39]. The algorithm follows objects' borders with a sort of topological analysis capability. If one wants to convert a binary picture into the border representation, then they can extract the topological structure of the image with little additional effort by using this function. The information to be extracted

```
void cv::findContours (InputArray image, OutputArrayOfArrays contours,
    OutputArray hierarchy, int mode, int method, Point offset = Point())

void cv::drawContours (InputOutputArray image, InputArrayOfArrays
    contours, int contourIdx, const Scalar & color, int thickness = 1,
    int lineType = LINE_8, InputArray hierarchy = noArray(), int maxLevel
     = INT_MAX, Point offset = Point())
```

Listing 1.1. OpenCV C++ API for *findContours* and *drawContours* functions.

```
[...]
vector<vector<Point>> contours;
vector<Vec4i> hierarchy;
findContours(src, contours, hierarchy, RETR_CCOMP, CHAIN_APPROX_SIMPLE);
for (int idx = 0; idx >= 0; idx = hierarchy[idx][0]) {
    Scalar color(rand() & 255, rand() & 255, rand() & 255);
    drawContours(dst, contours, idx, color, FILLED, 8, hierarchy);
}
[...]
```

Listing 1.2. OpenCV example on how to retrieve connected components from a binary image and fill them with random colors. Tested on version *4.5.5*.

```
int cv::connectedComponents(InputArray image, OutputArray labels, int
    connectivity, int ltype)

int cv::connectedComponentsWithStats(InputArray image, OutputArray labels
    , OutputArray stats, OutputArray centroids, int connectivity, int
    ltype)
```

Listing 1.3. OpenCV C++ API for *connectedComponents* and *connectedComponentsWithStats* functions.

is the inclusion relation among the two types of borders: the *outer borders* and the *hole borders*. Since there exists one-to-one correspondence between an outer border and a 1-component, and between a hole border and a 0-component, the topological structure of a given binary image can be determined.

A topological representation can be mapped into connected components by filling the contours. An example is reported in Listing 1.2.

3 A Novel Interface

Unfortunately, finding the contours and flood filling them is not a smart way of performing CCL. For this reason, researchers and practitioners started using different implementations found online until the release of OpenCV 3.0.0, which introduced two new interfaces (Listing 1.3).

The `connectedComponents` function takes a binary image as input and produces an integer symbolic image in which all the pixels from the same object are assigned the same (unique) number. With the parameter `connectivity` the user can specify whether to use 4- or 8-connectivity to define pixel connectivity (i.e. considering pixel connected only if they share the same border, 4-connectivity, or also if they share vertexes, 8-connectivity). `ltype` specifies whether the output image should use 16- or 32-bit per pixel. The function returns the total number of labels $[0, N-1]$, where 0 represents the background label. In this version only the Scan Array-Based Union Find (SAUF) algorithm by Wu *et al.* was available.

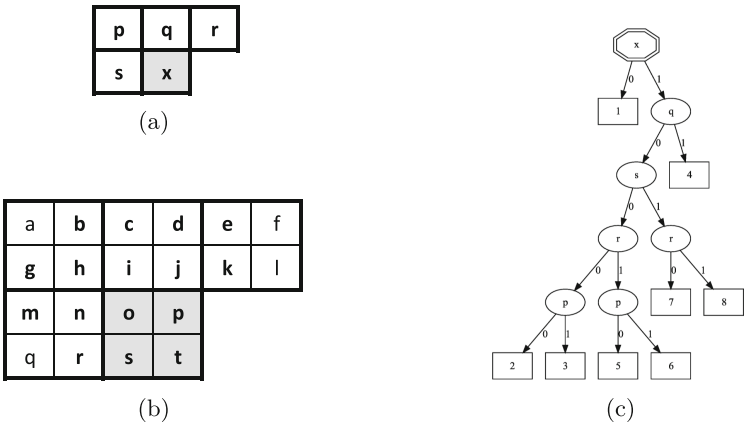

Fig. 1. (a) is the Rosenfeld mask used by SAUF to compute the label of pixel x during the first scan and (b) is the Grana mask used by BBDT to compute the label of pixels o, p, s and t. Finally, (c) is the optimal decision tree proposed in [42]. Internal nodes (ellipsis) represent the conditions to be checked, and leaves (rectangles) contain the actions to be performed, which are identified by integer numbers. The root of the tree, also a condition, is represented by a octagon. Action 1 represents *no action*. Action 2 is *new label*. Action 3 means $x \leftarrow p$, i.e. assign x the label of p. Action 4, 5, and 7 are respectively $x \leftarrow q$, $x \leftarrow r$, and $x \leftarrow s$. Finally, action 6 and 8 require merge between different label classes, specifically, $x \leftarrow r + p$, $x \leftarrow s + r$.

A `connectedComponentsWithStats` implementation is also available. This function allows to calculate at the same time the output symbolic image with labeled connected components and their statistics:

- the minimum bounding box containing the connected component;
- the area (in pixels) of the object;
- the centroids (x, y)-coordinates of connected components, including background.

All of this information is stored inside `stats` and *centroids* matrices. Also in this case, the SAUF algorithm is employed to identify connected objects.

The SAUF algorithm itself, introduced by Wu *et al.*, is based on two key elements:

- the use of the Union-Find algorithm to store and handle equivalences between pixel classes[1];

[1] The union-find data structure, first applied to CCL by Dillencourt *et al.* [19], provides two convenient procedures to deal with equivalence classes of labels: *Find*, which retrieves the representative label of an equivalence class, and *Union*, which merges two equivalence classes into one, ensuring that they share the same representative label.

– an optimal strategy, based on a manually identified decision tree, to reduce the average number of load/store operations during the first scan of the input image.

As most of the state-of-the-art algorithms for CCL, SAUF is based on a two scan (or two pass) approach. During the first scan of the image, the algorithm assigns temporary labels to pixels and records equivalences between classes. The second scan, instead, is meant to replace each temporary label with the representative of its equivalence class (usually the smallest one). The scanning approach is led by the Rosenfeld mask reported in Fig. 1a. Indeed, when labeling the current pixel, x, pixels p, q, r, and s are enough to determine the class which x belongs to. Moreover, if q is a foreground pixel, it is already connected with all the other foreground pixels in the "current" mask and this connectivity has already been recorded in the Union-Find data structure. This means that we can simply assign x the same class of q, saving three checks. When q is a background pixel, we can for example check pixel p. In this case, when p is foreground, r must be inspected also, to verify whether p and r are connected through x. If this is the case, a merge between the two classes have to be performed. Moving on with this reasoning, the decision tree depicted in Fig. 1c can be obtained. Other equivalently optimal[2] versions can be generated.

As said, combining the use of the optimal decision tree with the Union-Find algorithm optimized with path compression [42] translates into the SAUF algorithm. A similar approach can be applied to 4-connectivity producing this time a much simpler (and smaller) decision tree.

4 Going Faster with Blocks

In 2010, Grana *et al.* [22] introduced a major breakthrough, consisting in a 2×2 block-based approach denoted as *Block-Based with Decision Tree* algorithm (BBDT). The proposed algorithms make use of an optimal decision tree, generated upon the mask of Fig. 1, and the Union-Find algorithm implemented with Three Table Array (TTA) strategy proposed in [25].

The problem is modeled as a *command execution metaphor*: values of pixels in the scanning mask constitute a *rule* (binary string), which is associated to a set of equivalent actions in an *OR*-decision table. Given this decision table, an algorithm can simply read all the pixels inside the mask, identify the rule, and find the action to be performed in the corresponding column. A dynamic programming approach [24] is then used to convert the *OR*-decision table into an optimal binary decision trees. This approach allows to minimize the average number of conditions to be checked when choosing the correct action to be performed.

The possible actions are the same mentioned for SAUF algorithms, this time working with blocks: *no action* if the current block is background (i.e., all the

[2] Optimality is related to the number of accesses to the pixels in the scanning mask, *i.e.*, number of memory accesses.

```
int cv::connectedComponents(InputArray image, OutputArray labels, int
    connectivity, int ltype, int ccltype)

int cv::connectedComponentsWithStats(InputArray image, OutputArray labels
    , OutputArray stats, OutputArray centroids, int connectivity, int
    ltype, int ccltype)
```

Listing 1.4. OpenCV C++ API for *connectedComponents* and *connectedComponentsWithStats* functions.

pixels of the block are background), *new label* if it has no foreground neighbors, *assign* or *merge* based on the label of neighboring foreground pixels/blocks.

Since version *3.2.0*, the BBDT algorithm has been introduced in the OpenCV. Two new overloading functions, detailed in Listing 1.4, have been added to introduce the `ccltype` parameter while preserving the Application Binary Interface (ABI compatibility). This parameter makes the user able to select the algorithm to be used. Given that BBDT is only available for 8-connectivity, the SAUF version is always executed when labeling with 4-connectivity.

It is important to notice that, while the SAUF algorithm forces a row major ordering of labels, BBDT does not. This means that label ordering in the output *label* image may be different when executing the two algorithms, but with exactly the same semantic meaning.

5 Spaghetti for All

Many improvements have been proposed since the introduction of BBDT, and some of them introduced significantly novel ideas, in particular:

- realizing that it is possible to use a finite state machine to summarize the value of pixels already inspected by the horizontally moving scan mask [28];
- combining decision trees and configuration transitions in a decision forest, in which each previous pattern allows to "predict" some of the current configuration pixels values, thus allowing automatic code generation [20];
- switching from decision trees to Directed Rooted Acyclic Graphs (DRAGs), to reduce the machine code footprint and lessen its impact on the instruction cache [11].

Prediction, as introduced by He *et al.* [26], has proven to be one of the most useful additions, as it allows to exploit already available information, save expensive load/store operations, and reduce execution time consequently. When the scan mask is shifted along a row of the image it always contains some of the pixels it already contained in the previous step, though in different locations. If those pixels were indeed checked in the previous mask step, a second read of their value can be avoided by their removal from the decision process.

The procedure proposed in [20] was suitable to be automatized, but still a small mask was employed. The reason, in this case, was that the larger the mask

```
void cv::cuda::connectedComponents(InputArray image, OutputArray labels,
    int connectivity, int ltype, cv::cuda::
    ConnectedComponentsAlgorithmsTypes ccltype)
```

Listing 1.5. OpenCV C++ API for *connectedComponents* performed in CUDA.

is, the more decision trees will populate the resulting forest, and the higher every tree will be. The machine code that implements the algorithm resulting from the application of prediction to BBDT would be very large, and may have a negative impact on instruction cache. Therefore, despite load/store operations being less, the overall performance on real case datasets may be worse than that of the single tree variation. For this reason, all works on prediction chose to avoid the complexity of the BBDT mask, and simplified it in various ways.

In [7], the BBDT original mask and the *state prediction* paradigm are combined in the *Spaghetti Labeling* algorithm, by taking advantage of the code compression technique that converts a directed rooted tree into a DRAG [11]. The resulting process is modeled by a directed acyclic graph (DAG) with multiple entry points (roots), which correspond to the knowledge that can be inferred from the previous step. This guarantees a significant reduction of the machine code, even better than that achievable by a compiler, since it can leverage the presence of equivalent actions in the trees leaves, and compress not only equal subtrees, but also equivalent ones.

Spaghetti labeling has been included in OpenCV since version *4.5.2* and *3.4.14*. The signatures are the same as the previous ones, changing only the default value `ccltype = CCL_SPAGHETTI`.

The later introduction of GRAPHGEN [8], a technique for the automatic generation of decision DAGs inspired by Spaghetti, allowed, since version *4.5.5*, to also implement a 4-connected version of Spaghetti, making it the default algorithm for both 8- and 4-connectivity.

OpenCV aims at maximizing speed, thus parallelization is heavily employed throughout all library and a specifically developed framework is available. At the moment, following the embarrassingly parallel approach of [12], labeling algorithms are run on image stripes and a further joining stage is added. The parallel version of the algorithms is automatically employed if at least one of the allowed parallel frameworks is enabled and if the rows of the image are at least twice the number returned by `getNumberOfCPUs`.

6 GPU Implementation

Starting from the 4th major release of OpenCV, all CUDA modules are located in *opencv_contrib*,[3] an additional public repository containing extra modules that can be optionally added to the installation of the library. The CUDA version of

[3] https://github.com/opencv/opencv_contrib.

CCL has been recently added to *opencv_contrib*, and will be included in release *4.5.6*. Its signature, reported in Listing 1.5, was chosen to be as close as possible to the CPU version, with the only difference being the return type. This function, in fact, does not return the amount of labels assigned: the additional task of counting labels, which is trivial for most sequential algorithms, is instead considerably time consuming when performed in a massively parallel fashion, and for this reason it is excluded from the workload of CUDA CCL algorithms.

So far, the only available CUDA algorithm is Block-Based Komura Equivalence (BKE) [4], which takes advantages of both the Union-Find algorithm and the Block-Based approach and represents the current state of the art. In this proposal, the Union-Find structure is directly coded in the output image, in the sense that the provisional label assigned to each block doubles its meaning as the memory address of the parent in the Union-Find tree. This particular choice of provisional labels allows to avoid a specific data structure for the Union-Find.

Like all CUDA algorithms, BKE is composed of *kernels*, i.e. functions executed at the same time by a high number of threads. The kernels composing the algorithm are *Initialization*, *Compression*, *Reduction* and *FinalLabeling*, and are described in the following. Each uses a number of threads equal to the blocks in the image, so that each thread is responsible for labeling its own block, which will be referred to as X.

Initialization. Each thread looks at the neighborhood in order to find out which blocks are connected to X, then takes the smallest of their raster addresses and sets it as the initial label of X. From the Union-Find point of view, this means that X is assigned a parent in the forest. Finally, an information bitset detailing with pixels of the block are foreground and which blocks are connected to X is stored in the output image, along with the provisional label; it will be used again in subsequent kernels. In this case, the output image is used as a temporary buffer: this information bitset is only useful for the algorithm, and will not be present in the final output.

Compression. This kernel flattens the Union-Find trees coded in the image, by means of the *Find* operation: each thread reads the parent label of X, then the parent of the parent, and repeats the process until it reaches the root; then, it assigns the root label to X. After this compression, all trees have height 1.

Reduction. Each thread reads the information bitset stored in Initialization in order to find out which blocks are connected to X, and then proceeds to make sure that all of them are indeed in the same Union-Find tree. This is accomplished by means of the *Union* procedure, which takes two nodes as input, traces back their trees until the roots and finally links one root to the other. Of course, the neighbor blocks with the smallest address is excluded, since X has already been connected to it in *Initialization*. From the Union-Find point of view, the Reduction kernel completes the CCL task: each block is put in the same tree as all of its neighbor, and therefore each tree in the forest completely

corresponds to a CC in the image. After Reduction, a second Compression is performed, again to flatten trees to height 1. This time, however, it also means that each block in the same tree has the same parent, and thus the same label.

FinalLabeling. The only remaining operation to perform at this point is to assign block labels to single pixels. Each thread reads the information bitset again, this time to check which pixels of the blocks are foreground; then, it assigns the label of X to all of them, and label 0 to the remaining pixels. This is the final rewriting of the output image, and overwrites the information bitset previously stored. After FinalLabeling, each pixel in the same CC has the same unique label, and thus the labeling task is completed.

7 Discussion

The inclusion of algorithms in OpenCV has been done after a careful and really open comparison of the execution times, evaluated using YACCLAB [11,21], a widely used [16,33] open source *C++* benchmarking framework for CCL algorithms. YACCLAB allows researchers to test state-of-the-art algorithms on real and synthetic generated datasets. The fairness of the comparison is guaranteed by compiling the algorithms with the same optimizations and by running them on the same data and over the same machine.

The algorithms provided by YACCLAB cover most of the paradigms for CCL explored in the past, along with a lower bound limit for all CCL algorithms over a specific dataset/image, obtained by reading once the input image and writing it on the output again.

The benchmark provides a template implementation of the algorithms over the labels solving strategy. Using different label solvers can significantly change the performance of a specific combination of dataset, algorithm and operating system.

The YACCLAB dataset covers most applications in which CCL may be useful, and features a significant variability in terms of resolution, image density, variance of density, and number of components. It includes six real-world datasets, and specifically: *3DPeS* [6], *Fingerprints*, *Medical*, *MIRflickr*, *Tobacco-800*, *XDOCS* [10].

A clear result is that, on average, Spaghetti Labeling is the optimal choice. In very specific corner cases, such as when the order of labels needs to be sorted by rows, or when the instruction cache is extremely small, other techniques could be employed. The combination of `FindContours` and `DrawContours` is a viable solution if your aim is to obtain the contours, because the connected components are an additional bonus. If you just need the connected components, these should be definitely avoided. The GPU version is now available and it makes sense if your images are already in GPU, allowing you to stay in GPU without moving back and forth from main memory to device memory. Even if the GPU version is faster than Spaghetti Labeling, the total amount of time required to move data between host and device plus the CCL procedure is higher than running in CPU directly.

8 Conclusion

With this paper we provided a review of the sequential and parallel implementation of CCL algorithms included in the OpenCV library. The open source nature of OpenCV allowed to spot numerous and subtle bugs, and it is always incredible how many small details may be overlooked in real world usage of code.

All the additions to OpenCV, not only for CCL, have been strongly motivated by independent performance evaluations, in terms of effectiveness, or (as for this specific case) speed. Every alternative proposal should be openly evaluated and the source code needs to be released publicly, in order to avoid contrasting claims of "I'm better than you". We want the user to git-pull our code and check if it really is the best for his use case, or not.

References

1. Allegretti, S., Bolelli, F., Cancilla, M., Grana, C.: Optimizing GPU-based connected components labeling algorithms. In: IPAS, pp. 175–180 (2018)
2. Allegretti, S., Bolelli, F., Cancilla, M., Grana, C.: A block-based union-find algorithm to label connected components on GPUs. In: Ricci, E., Rota Bulò, S., Snoek, C., Lanz, O., Messelodi, S., Sebe, N. (eds.) ICIAP 2019. LNCS, vol. 11752, pp. 271–281. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30645-8_25
3. Allegretti, S., Bolelli, F., Cancilla, M., Pollastri, F., Canalini, L., Grana, C.: How does connected components labeling with decision trees perform on GPUs? In: Vento, M., Percannella, G. (eds.) CAIP 2019. LNCS, vol. 11678, pp. 39–51. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29888-3_4
4. Allegretti, S., Bolelli, F., Grana, C.: Optimized block-based algorithms to label connected components on GPUs. IEEE Trans. Parallel Distrib. Syst. **31**, 423–438 (2019). https://doi.org/10.1109/TPDS.2019.2934683
5. Allegretti, S., Bolelli, F., Pollastri, F., Longhitano, S., Pellacani, G., Grana, C.: Supporting skin lesion diagnosis with content-based image retrieval. In: 2020 25th International Conference on Pattern Recognition (ICPR). IEEE, January 2021
6. Baltieri, D., Vezzani, R., Cucchiara, R.: 3DPeS: 3D people dataset for surveillance and forensics. In: Proceedings of the 2011 Joint ACM Workshop on Human Gesture and Behavior Understanding, pp. 59–64. ACM (2011)
7. Bolelli, F., Allegretti, S., Baraldi, L., Grana, C.: Spaghetti labeling: directed acyclic graphs for block-based connected components labeling. IEEE Trans. Image Process. **29**(1), 1999–2012 (2019)
8. Bolelli, F., Allegretti, S., Grana, C.: One DAG to rule them all. IEEE Trans. Pattern Anal. Mach. Intell., 1–12 (2021). https://doi.org/10.1109/TPAMI.2021.3055337
9. Bolelli, F., Baraldi, L., Cancilla, M., Grana, C.: Connected components labeling on DRAGs. In: 2018 24th International Conference on Pattern Recognition (ICPR), pp. 121–126 (2018)
10. Bolelli, F., Borghi, G., Grana, C.: XDOCS: an application to index historical documents. In: Serra, G., Tasso, C. (eds.) IRCDL 2018. CCIS, vol. 806, pp. 151–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73165-0_15
11. Bolelli, F., Cancilla, M., Baraldi, L., Grana, C.: Towards reliable experiments on the performance of Connected Components Labeling algorithms. J. Real-Time Image Proc. **17**(2), 229–244 (2018)

12. Bolelli, F., Cancilla, M., Grana, C.: Two more strategies to speed up connected components labeling algorithms. In: Battiato, S., Gallo, G., Schettini, R., Stanco, F. (eds.) ICIAP 2017. LNCS, vol. 10485, pp. 48–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68548-9_5
13. Canalini, L., Pollastri, F., Bolelli, F., Cancilla, M., Allegretti, S., Grana, C.: Skin lesion segmentation ensemble with diverse training strategies. In: Vento, M., Percannella, G. (eds.) CAIP 2019. LNCS, vol. 11678, pp. 89–101. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29888-3_8
14. Chang, W.Y., Chiu, C.C.: An efficient scan algorithm for block-based connected component labeling. In: 22nd Mediterranean Conference on Control and Automation, pp. 1008–1013 (2014)
15. Chang, W.Y., Chiu, C.C., Yang, J.H.: Block-based connected-component labeling algorithm using binary decision trees. Sensors **15**(9), 23763–23787 (2015)
16. Chen, J., Nonaka, K., Sankoh, H., Watanabe, R., Sabirin, H., Naito, S.: Efficient parallel connected component labeling with a coarse-to-fine strategy. IEEE Access **6**, 55731–55740 (2018)
17. Cipriano, M., et al.: Deep segmentation of the mandibular canal: a new 3D annotated dataset of CBCT volumes. IEEE Access **10**, 11500–11510 (2022)
18. Cipriano, M., Allegretti, S., Bolelli, F., Pollastri, F., Grana, C.: Improving segmentation of the inferior alveolar nerve through deep label propagation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–10. IEEE (2022)
19. Dillencourt, M.B., Samet, H., Tamminen, M.: A general approach to connected-component labeling for arbitrary image representations. J. ACM **39**(2), 253–280 (1992)
20. Grana, C., Baraldi, L., Bolelli, F.: Optimized connected components labeling with pixel prediction. In: Blanc-Talon, J., Distante, C., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2016. LNCS, vol. 10016, pp. 431–440. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48680-2_38
21. Grana, C., Bolelli, F., Baraldi, L., Vezzani, R.: YACCLAB - yet another connected components labeling benchmark. In: 2016 23rd International Conference on Pattern Recognition (ICPR), pp. 3109–3114 (2016)
22. Grana, C., Borghesani, D., Cucchiara, R.: Optimized block-based connected components labeling with decision trees. IEEE Trans. Image Process. **19**(6), 1596–1609 (2010)
23. Grana, C., Borghesani, D., Cucchiara, R.: Automatic segmentation of digitalized historical manuscripts. Multimedia Tools Appl. **55**(3), 483–506 (2011)
24. Grana, C., Montangero, M., Borghesani, D.: Optimal decision trees for local image processing algorithms. Pattern Recogn. Lett. **33**(16), 2302–2310 (2012)
25. He, L., Chao, Y., Suzuki, K.: A linear-time two-scan labeling algorithm. In: 2007 IEEE International Conference on Image Processing, pp. 241–244 (2007)
26. He, L., Chao, Y., Suzuki, K.: An efficient first-scan method for label-equivalence-based labeling algorithms. Pattern Recogn. Lett. **31**(1), 28–35 (2010)
27. He, L., Chao, Y., Suzuki, K.: Two efficient label-equivalence-based connected-component labeling algorithms for 3-D binary images. IEEE Trans. Image Process. **20**(8), 2122–2134 (2011)
28. He, L., Zhao, X., Chao, Y., Suzuki, K.: Configuration-transition-based connected-component labeling. IEEE Trans. Image Process. **23**(2), 943–951 (2014)

29. Hennequin, A., Lacassagne, L., Cabaret, L., Meunier, Q.: A new direct connected component labeling and analysis algorithms for GPUs. In: 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 76–81. IEEE (2018)
30. Illingworth, J., Kittler, J.: A survey of the Hough transform. Comput. Vis. Graph. Image Process. **44**(1), 87–116 (1988)
31. Lacassagne, L., Zavidovique, B.: Light speed labeling: efficient connected component labeling on RISC architectures. J. Real-Time Image Proc. **6**(2), 117–135 (2011). https://doi.org/10.1007/s11554-009-0134-0
32. Laradji, I.H., Rostamzadeh, N., Pinheiro, P.O., Vazquez, D., Schmidt, M.: Where are the Blobs: counting by localization with point supervision. In: Computer Vision – ECCV 2018, pp. 547–562 (2018)
33. Ma, D., Liu, S., Liao, Q.: Run-based connected components labeling using double-row scan. In: Zhao, Y., Kong, X., Taubman, D. (eds.) ICIG 2017. LNCS, vol. 10668, pp. 264–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71598-8_24
34. Pham, H.V., Bhaduri, B., Tangella, K., Best-Popescu, C., Popescu, G.: Real time blood testing using quantitative phase imaging. PLoS ONE **8**(2), e55676 (2013)
35. Playne, D., Hawick, K.: A new algorithm for parallel connected-component labelling on GPUs. IEEE Trans. Parallel Distrib. Syst. **29**(6), 1217–1230 (2018). https://doi.org/10.1109/TPDS.2018.2799216
36. Pollastri, F., Bolelli, F., Paredes, R., Grana, C.: Augmenting data with GANs to segment melanoma skin lesions. Multimedia Tools Appl. **79**(21–22), 15575–15592 (2019)
37. Pulli, K., Baksheev, A., Kornyakov, K., Eruhimov, V.: Realtime computer vision with OpenCV: mobile computer-vision technology will soon become as ubiquitous as touch interfaces. Queue **10**(4), 40–56 (2012). https://doi.org/10.1145/2181796.2206309
38. Rosenfeld, A., Pfaltz, J.L.: Sequential operations in digital picture processing. J. ACM **13**(4), 471–494 (1966)
39. Suzuki, S., Abe, K.: Topological structural analysis of digitized binary images by border following. Comput. Vis. Graph. Image Process. **30**(1), 32–46 (1985). https://doi.org/10.1016/0734-189X(85)90016-7
40. Söchting, M., Allegretti, S., Bolelli, F., Grana, C.: A heuristic-based decision tree for connected components labeling of 3D volumes. In: 2020 25th International Conference on Pattern Recognition (ICPR), pp. 7751–7758. IEEE, January 2021. https://doi.org/10.1109/ICPR48806.2021.9413096
41. Uslu, F., Bharath, A.A.: A recursive Bayesian approach to describe retinal vasculature geometry. Pattern Recogn. **87**, 157–169 (2019)
42. Wu, K., Otoo, E., Suzuki, K.: Two strategies to speed up connected component labeling algorithms. Pattern Analysis Application 0(LBNL-59102) (2005)
43. Zavalishin, S., Safonov, I., Bekhtin, Y., Kurilin, I.: Block equivalence algorithm for labeling 2D and 3D images on GPU. EI **2016**(2), 1–7 (2016)