A Block-Based Union-Find Algorithm to Label Connected Components on GPUs

Stefano Allegretti, Federico Bolelli, Michele Cancilla, and Costantino Grana

Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia
Via Vivarelli 10, Modena MO 41125, Italy
{name.surname}@unimore.it

Abstract. In this paper, we introduce a novel GPU-based Connected Components Labeling algorithm: the Block-based Union Find. The proposed strategy significantly improves an existing GPU algorithm, taking advantage of a block-based approach. Experimental results on real cases and synthetically generated datasets demonstrate the superiority of the new proposal with respect to state-of-the-art.

Keywords: Connected Components Labeling · Image Processing · GPU · CUDA

1 Introduction

In the last decades, the maturity of Graphic Processing Units (GPUs) encouraged the development of algorithms specifically designed to work in a data-parallel environment [4]. Indeed, applications characterized by irregular control flow and irregular memory access patterns usually break the parallel execution model when ported on GPU: they must be redesigned to take advantage of the GPU architecture [12]. Connected Components Labeling (CCL), an essential image processing algorithm that extracts objects inside binary images, is such a kind of algorithm. The labeling procedure transforms an input binary image into a symbolic one in which all pixels belonging to a connected component are given the same label. Even though labeling has an intrinsically sequential nature [7,19,24], many algorithms exploiting the parallelism of both CPUs and GPUs have been recently proposed [3,11,13,27,38].

CCL, originally introduced by Rosenfeld and Pfaltz in 1966 [35], has an exact solution, and the algorithms are mainly characterized by their execution time. Since labeling represents the base step of many image processing applications [14,15,17,18,28,33,34], it is required to be as fast as possible. Unfortunately, CCL is not as easy to parallelize as other image processing tasks: CPU and GPU algorithms usually have comparable performance [32]. However, efficient data-parallel algorithms are valuable for applications that totally run on GPU, allowing to remove the need for data transfers between CPU and GPU memory.

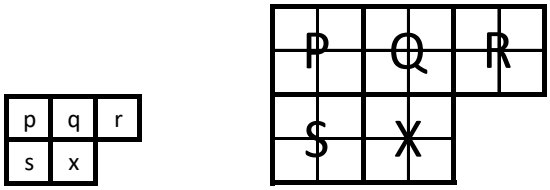
In this paper, we introduce a new 8-connectivity GPU-based connected components labeling algorithm, which improves previously proposed solutions by

taking advantage of the 2×2 block-based approach originally presented in [21] for sequential algorithms. The proposed method reduces the amount of memory accesses, significantly improving state-of-the-art performance in terms of execution time over both real case and synthetically generated datasets. The source code of our proposal is available in [36].

The rest of this paper is organized as follows. In Section 2, the main contributions on parallel CCL are resumed. Section 3 analyzes the Union Find algorithm, which represents the basis of our work, then Section 4 details our proposal. Section 5 demonstrates the effectiveness of our approach in comparison with other state-of-the-art methods, providing an exhaustive evaluation. Finally, conclusions are drawn in Section 6.

2 Related Work

The first work on GPU CCL dates back in 2010, when Hawick *et al.* [22] proposed Label Equivalence (LE). LE is an iterative algorithm that propagates the minimum label through every connected component. The process is sped up by alternating the propagation phase with label equivalences resolution. In 2011, Kalentev *et al.* [26] proposed an optimization of Label Equivalence, which we will call OLE, obtained by removing overabundant operations and memory allocations. Komura Equivalence (KE) [27] was also created as an improvement over Label Equivalence, which removes the need for multiple iterations. The original algorithm employs 4-connectivity, and it has been extended to 8-connectivity in [2]. Zavalishin *et al.* [38] further improved OLE, applying a block-based strategy to reduce the number of temporary labels, and memory accesses. The result is known as Block Equivalence (BE). The benefit introduced by blocks was partially lessened by an increased allocation time, caused by the need for additional data structures to record block labels and connectivity information. Union Find (UF), by Oliveira and Lotufo [31], is a parallel algorithm that employs the *Union-Find* data structure, commonly used to solve labels equivalences by sequential algorithms [10,21,37].



(a) Rosenfeld (b) Block-based

Fig. 1. Neighborhood masks used by the algorithms described in the paper. UF employs the mask in (a), where the central pixel is x. The block-based mask (b) is used by BUF instead. Central block is X.

3 Preliminaries

The proposal of this paper is an optimization of the Union Find algorithm (UF), by Oliveira and Lotufo, which is briefly introduced in this section. UF performs a partitioning of the output image L by creating subsets of connected pixels, and merging together those belonging to the same connected component. To perform this task, it takes advantage of the *Union-Find* paradigm, which represents subsets as directed rooted trees and provides convenient functions to deal with them: *Find* that returns the root of a tree and *Union* that joins together two different trees.

Algorithm 1 Possible implementation of *Union-Find*. L is the *Union-Find* array, a and b are both array indexes and pixel identifiers.

```
1:  function FIND(L, a)
2:      while L[a] ≠ a do
3:          a ← L[a]
4:      return a

5:  procedure UNION(L, a, b)
6:      a ← Find(L, a)
7:      b ← Find(L, b)
8:      if a < b then
9:          L[b] ← a
10:     else if b < a then
11:         L[a] ← b
```

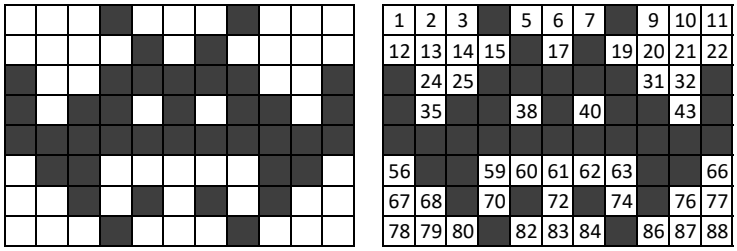
Trees are coded in the output image L, using temporary labels: for a pixel p, with raster index id_p, $L[id_p] = id_f$ is the father node of p. A possible implementation of the *Union-Find* functions is reported in Algorithm 1. A description follows:

- *Find*(L, a) consists of traversing the tree to which a belongs, starting from a up to the root node.
- *Union*(L, a, b) first calls *Find* twice to get the roots of the trees containing a and b, and then sets the smaller root as the father of the other one, thus joining the two trees into a single one. The procedure used in the source code is slightly more complicated, to avoid race hazards in a parallel environment.

An example of execution of the whole algorithm is depicted in Fig. 2. The algorithm consists of three kernels: *Initialization*, *Merge* and *Compression*. During *Initialization*, single-node trees are coded in the output image L, by assigning each foreground pixel its own raster index. All background pixels are set to 0.

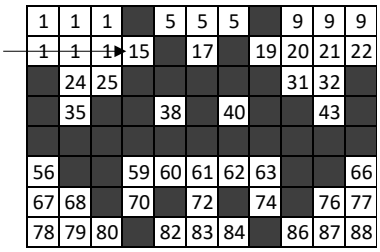
The aim of the *Merge* kernel is to build a single tree for each connected component. To achieve this goal, each thread working on a foreground pixel x joins the tree of x to those of its foreground neighbors, by means of *Union* procedures. Since *Union* is symmetric, checking the whole neighborhood is not necessary. Instead, only half of it is considered, identified by the mask depicted in Fig. 1a. The effects of *Merge* on *Union-Find* trees are shown in Fig. 2c. In this example, the thread operating on pixel 15 performs a *Union* between 15 and 1, and then another *Union* between 15 and 5.

In the *Compression* kernel, every *Union-Find* tree is flattened, by linking every node directly to the root. This process ends the connected components labeling task, because every pixel of the same connected component is given the same value.

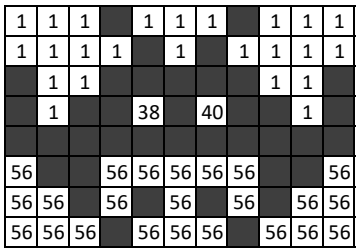


(a) Input

(b) Initialization



(c) Temporary Labels



(d) Final Labels

Fig. 2. Example of Union Find execution. (b) is the expected labels image after *Initialization*. (c) is a temporary result of *Merge* kernel, under the assumption that threads run in raster scan order, and the execution reached thread 15. (d) is the final labels image after the execution of the *Compression* kernel.

4 Proposed Algorithm

Grana *et al.* noticed in [21] that, in the case of a two-dimensional image and 8-connectivity, all foreground pixels within 2×2 blocks always share the same label. Consequently, they designed a CCL algorithm that uses block labels instead of pixel labels throughout the process, to greatly reduce the total amount of memory accesses and speed up performance consequently.

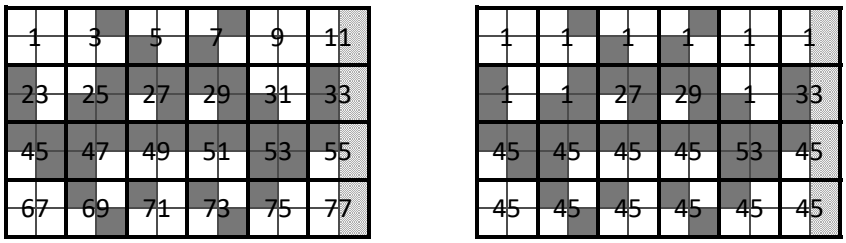
We propose a new GPU CCL 8-connectivity algorithm, which is an optimized variation of UF obtained through the application of 2×2 blocks. Our proposal, named Block-based Union Find (BUF), inherits the base structure of Union Find (Sec. 3). The difference resides in the use of block labels. In fact, every thread works on a 2×2 block, which we will refer to as the X block. The algorithm implements the same kernels as UF, plus the additional *FinalLabeling*, which is needed to copy block labels into pixels. Differently from the work by Zavalishin *et al.* [38], we do not allocate memory for block labels. Instead, until the end of the algorithm, we store them directly in the output image: the label assigned to a block is stored in its top-left pixel, whose raster index is also used as the block *id*.

The first kernel of the algorithm, *Initialization*, creates the starting *Union-Find* trees. At the beginning, one separate tree is built for each block X, by

Algorithm 2 Block-based Union Find *Merge* kernel. I and L are input and output images, linearly stored in memory. A padding can be added at the end of rows for alignment purpose, so *step* stores their total size in memory. A thread is identified by $(t_x, t_y) \in \mathcal{N}^2$, with $t_x \in [0, \lceil cols/2 \rceil]$ and $t_y \in [0, \lceil rows/2 \rceil]$.

1: **kernel** MERGE(I, $step_I$, L, $step_L$)
2: $x_I \leftarrow 2 \times t_y \times step_I + t_x \times 2$
3: $x_L \leftarrow 2 \times t_y \times step_L + t_x \times 2$
4: $\mathcal{BS} \leftarrow 0$
5: **if** $I[x_I] = 1$ **then** $\mathcal{BS}$ |= `0x777`
6: **if** $I[x_I + 1] = 1$ **then** $\mathcal{BS}$ |= (`0x777` << 1)
7: **if** $I[x_I + step_I] = 1$ **then** $\mathcal{BS}$ |= (`0x777` << 4)
8: **if** $\mathcal{BS} > 0$ **then**
9: **if** `HasBit`$(\mathcal{BS}, 0)$ **and** $I[x_I - step_I - 1]$ **then**
10: `Union`$(L, x_L, x_L - 2 \times step_L - 2)$
11: **if** (`HasBit`$(\mathcal{BS}, 1)$ **and** $I[x_I - step_I]$) **or**
12: (`HasBit`$(\mathcal{BS}, 2)$ **and** $I[x_I - step_I + 1]$) **then**
13: `Union`$(L, x_L, x_L - 2 \times step_L)$
14: **if** `HasBit`$(\mathcal{BS}, 3)$ **and** $I[x_I - step_I + 2]$ **then**
15: `Union`$(L, x_L, x_L - 2 \times step_L + 2)$
16: **if** (`HasBit`$(\mathcal{BS}, 4)$ **and** $I[x_I - 1]$) **or**
17: (`HasBit`$(\mathcal{BS}, 8)$ **and** $I[x_I + step_I - 1]$) **then**
18: `Union`$(L, x_L, x_L - 2)$

performing $L[id_X] \leftarrow id_X$. Then, the *Merge* kernel joins the trees of connected blocks, as illustrated in Algorithm 2. The block neighborhood mask, which contains half the neighborhood, is depicted in Fig. 1b. Since blocks connections are determined by lower level pixel connections, for every neighbor block of the mask we must check whether some of its pixels are connected to some internal pixels of block X. A naive approach, which just checks each adjacent block one by one, would require multiple readings of internal pixels. So, it is better to find a more efficient way. We adopted a strategy based on the work by Zavalishin *et al.* [38], which involves a preliminary scan of pixels inside the block: for each foreground one, its external neighbors are added to a set of pixels that will be checked subsequently. The aforementioned set of pixels is represented as a bitset that contains a bit for each pixel in a 4×4 square that encloses the X block, as reported in Fig. 4. Initially, every bit is set to 0. When an internal pixel a is read and recognized as foreground, each external pixel e neighbor to a must have its corresponding bit set to 1. To conveniently achieve this goal, the whole 3×3 square centered on a is set accordingly, by means of a bitmask (Fig. 4b). Bitmask `0x777` is required to set neighbors of the top-left pixel inside block X. The bitmasks of other pixels can be obtained in the following way: if the pixel is in the right column of the block, `0x777` is shifted one bit left. If the pixel is in the bottom row, the bitmask is shifted four bits left. The bottom-right pixel of X is never responsible for connections between blocks inside the mask, so it



(a) Temporary Labels (b) Final Labels

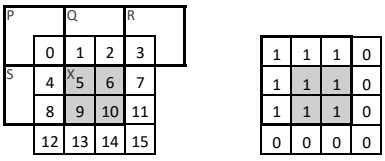
Fig. 3. Example of Block-based Union Find execution. (a) are the labels after *Initialization*. Every block has its own label, equal to the raster index of its top-left pixel. (b) are final block labels, after *Compression*. Blocks in the same connected component shares the same label, and the only remaining thing to do is to copy block labels into internal foreground pixels.

is never used. To find out which neighbor blocks are connected to X, the *Merge* kernel must then check which pixels of the bitset are set, and read their values. A *Union* is performed between X and connected blocks, as it happens for single pixels in UF.

The BUF *Compression* kernel then performs the flattening of *Union-Find* trees, by linking each block directly to the result of the *Find*. The effects of *Merge* and *Compression* on an input image are depicted in Fig. 3. Eventually, *FinalLabeling* copies the label of each block into its internal foreground pixels, thus producing the final output.

5 Comparative Evaluation

The proposed strategy is evaluated by comparing its performance with state-of-the-art algorithms. Experimental results reported and discussed in this Section are obtained running the YACCLAB benchmark [10,20] on an Intel Core i7-



(a) (b)

Fig. 4. (a) shows how pixels in a 4×4 square centered on the X block are numerated. These numbers correspond to the pixel position in the associated bitset. Bits 0, 1, 2, 3, 4, and 8 are used to record whether the corresponding pixel is to be checked for determining blocks connectivity or not. The other bits are stored for convenience. (b) depicts the 3×3 bitmask (`0x777`) corresponding to the neighbors of the top-left internal pixel.

Table 1. Average run-time results in ms obtained under Windows (64 bit) OS with MSVC 19.15.26730 and NVCC V10.0.130 compilers using a Quadro K2200 NVIDIA GPU. The bold values represent the best performing CCL algorithm on a given dataset. Our proposals are identified with *.

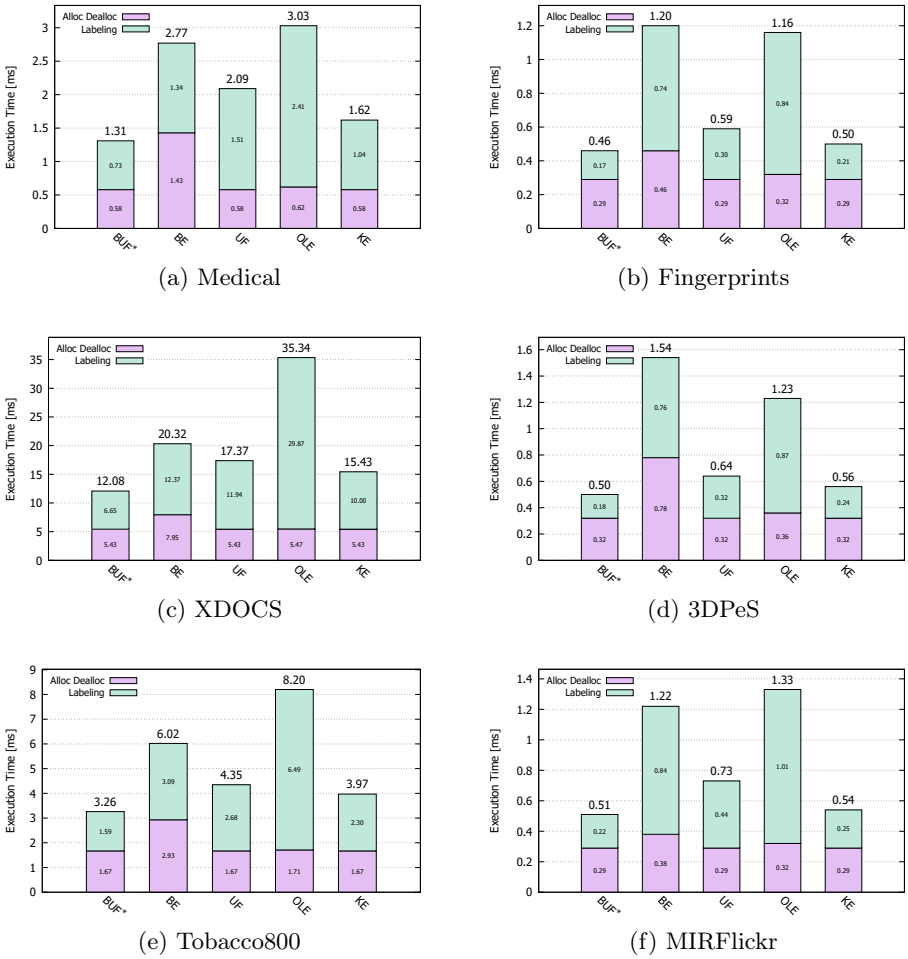
	3DPeS	*Fingerprints*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
BUF*	**0.512**	**0.441**	**1.313**	**0.495**	**3.268**	**12.088**
BE [38]	1.517	1.164	2.730	1.165	5.966	20.278
UF [31]	0.594	0.529	2.040	0.659	4.304	17.316
OLE [26]	1.211	1.128	3.013	1.281	8.173	35.242
KE [2]	0.568	0.481	1.622	0.526	3.978	15.432

4770 CPU (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), and using a Quadro K2200 NVIDIA GPU with Maxwell architecture, 640 CUDA cores and 4 GB of memory. All the compared algorithms have been implemented using CUDA 10.0 and compiled for x64 architectures, employing MSVC 19.15.26730 and NVCC V10.0.130 compilers with optimizations enabled. The benchmark provides a set of datasets covering real case scenarios for CCL, among which we selected the most significant ones: *MIRflickr* [25], *Medical* [16], *Tobacco800* [1,29], *XDOCS* [6,8,9], *Fingerprints* [30], and *3DPeS* [5]. A complete description of these datasets can be found in [10]. The first experiment carried out is the comparison between algorithms in terms of average execution time over real datasets (Table 1). Our proposal outperforms state-of-the-art on all test collections. The speed-up between BUF and KE, the best among competitors, varies from 1.1 (*MIRflickr*) to 1.3 (*XDOCS*).

To better investigate the algorithms behavior, Fig. 5 is also reported, where bar charts report separately the time needed for allocating data structures and the time required by the labeling procedure. The allocation time is the same for each strategy, but for BE. Indeed, all the algorithms must only allocate memory for the output image. BE always requires a higher allocation time, since it relies on additional matrices to store equivalences between blocks and their labels. Obviously, this additional time is data dependent. We can notice that OLE always has the highest execution time. The main drawback of the algorithm is its iterative nature, which is inherited by its block-based variation, BE. In fact, the benefits introduced by blocks allow BE to only have comparable performance to UF, which employs a direct, non iterative approach. Moreover, BE is partially hindered by its increased allocation time.

With our approach, we greatly improve the performance of UF. In fact, the use of block labels allows to divide by four the initial number of *Union-Find* trees. Consequently, the amount of *Union* operations required to merge trees in the same connected component drastically decreases, and the lessened average depth of trees allows to simplify *Find* calls. Besides, BUF, while benefiting from the advantages of blocks, avoids the main flaw of BE, namely the allocation of additional memory.

Following a common approach in literature [21,23,37], additional tests have been performed on images with increasing foreground density, in order to high-



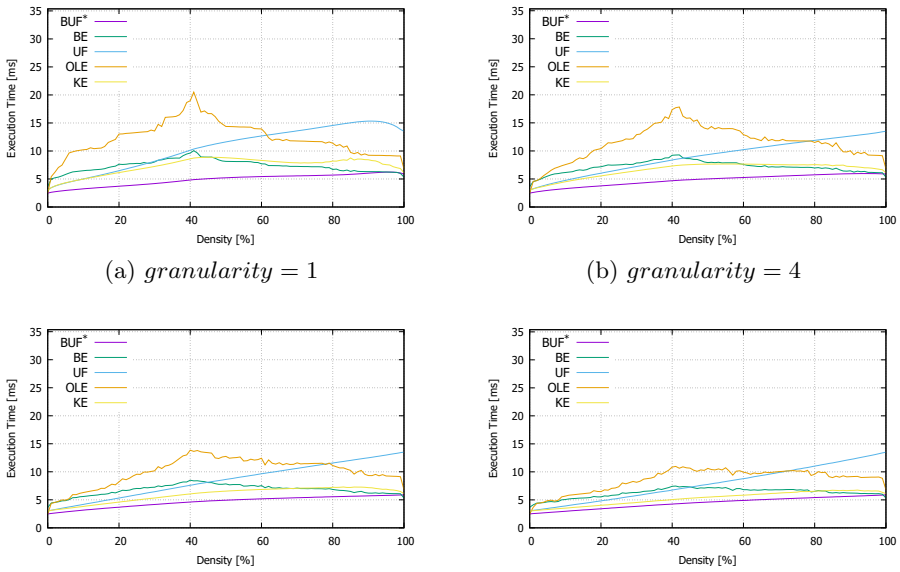
(a) Medical (b) Fingerprints

(c) XDOCS (d) 3DPeS

(e) Tobacco800 (f) MIRFlickr

Fig. 5. Average run-time results with steps in ms. Lower is better.

light strengths and weaknesses of the algorithms (Fig. 6). OLE has an increasing trend in the execution time up to 40% of foreground density, and then a decreasing one after this value. Indeed, the number of iterations required by the labeling procedure reaches the highest value when foreground density is about 40%. BE has a similar behavior, albeit with better performance. The execution time of UF grows with foreground density. The reason is that each pixel thread has to perform one *Union* for each connected neighbor, and the number of those pixels depends on image density. BUF has a similar trend to UF, since it inherits its basic behavior. The adoption of a block-based approach, anyway, allows to decrease the amount of operations, drastically reducing the total execution time. At 80% density and above, the high number ot *Union* operations makes BUF slower than BE. Anyway, such density values are rather uncommon in real cases.



(a) $granularity = 1$ (b) $granularity = 4$

(c) $granularity = 8$ (d) $granularity = 16$

Fig. 6. Granularity results in ms on images at various densities. Lower is better.

6 Conclusion

In this paper, the problem of GPU-based Connected Components Labeling in binary images has been addressed. A new algorithm has been proposed, Block-based Union Find, which was obtained by combining an existing strategy with a block-based approach. This allows to considerably lessen the number of memory accesses and consequently reduce execution time. Experimental tests on a wide selection of real case datasets, covering most of the fields where CCL is commonly used, confirm that our proposal represents the state-of-the-art for GPU-based Connected Components Labeling.

References

1. Agam, G., Argamon, S., Frieder, O., Grossman, D., Lewis, D.: The Complex Document Image Processing (CDIP) Test Collection Project. Illinois Institute of Technology (2006)
2. Allegretti, S., Bolelli, F., Cancilla, M., Grana, C.: Optimizing GPU-Based Connected Components Labeling Algorithms. In: Third IEEE International Conference on Image Processing, Applications and Systems (IPAS). pp. 175–180. IEEE (2018)
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: 18th International Conference on Computer Analysis of Images and Patterns (CAIP). Springer (2019)

4. Andrecut, M.: Parallel GPU Implementation of Iterative PCA Algorithms. Journal of Computational Biology **16**(11), 1593–1599 (2009)
5. 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)
6. Bolelli, F.: Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text. In: Italian Research Conference on Digital Libraries (IRCDL). pp. 45–55. Springer (2017)
7. Bolelli, F., Baraldi, L., Cancilla, M., Grana, C.: Connected Components Labeling on DRAGs. In: International Conference on Pattern Recognition (ICPR). pp. 121–126. IEEE (2018)
8. Bolelli, F., Borghi, G., Grana, C.: Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features. In: 19th International Conference on Image Analysis and Processing (ICIAP). pp. 729–738. Springer (2017)
9. Bolelli, F., Borghi, G., Grana, C.: XDOCS: An Application to Index Historical Documents. In: Italian Research Conference on Digital Libraries (IRCDL). pp. 151–162. Springer (2018)
10. Bolelli, F., Cancilla, M., Baraldi, L., Grana, C.: Toward reliable experiments on the performance of Connected Components Labeling algorithms. Journal of Real-Time Image Processing pp. 1–16 (2018)
11. Bolelli, F., Cancilla, M., Grana, C.: Two More Strategies to Speed Up Connected Components Labeling Algorithms. In: International Conference on Image Analysis and Processing (ICIAP). pp. 48–58. Springer (2017)
12. Brunie, N., Collange, S., Diamos, G.: Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In: 39th Annual International Symposium on Computer Architecture (ISCA). pp. 49–60 (2012)
13. Cabaret, L., Lacassagne, L., Etiemble, D.: Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs. In: Seventh International Conference on Image Processing Theory, Tools and Applications (IPTA). pp. 1–6. IEEE (2017)
14. Canalini, L., Pollastri, F., Bolelli, F., Cancilla, M., Allegretti, S., Grana, C.: Skin Lesion Segmentation Ensemble with Diverse Training Strategies. In: 18th International Conference on Computer Analysis of Images and Patterns (CAIP). Springer (2019)
15. Cucchiara, R., Grana, C., Prati, A., Vezzani, R.: Computer vision techniques for PDA accessibility of in-house video surveillance. In: First ACM SIGMM international workshop on Video surveillance. pp. 87–97. ACM (2003)
16. Dong, F., Irshad, H., Oh, E.Y., et al.: Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast. PloS one **9**(12), e114885 (2014)
17. Dubois, A., Charpillet, F.: Tracking Mobile Objects with Several Kinects using HMMs and Component Labelling. In: Workshop Assistance and Service Robotics in a human environment, International Conference on Intelligent Robots and Systems. pp. 7–13 (2012)
18. Eklund, A., Dufort, P., Villani, M., LaConte, S.: BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs. Frontiers in neuroinformatics **8**, 24 (2014)
19. Grana, C., Baraldi, L., Bolelli, F.: Optimized Connected Components Labeling with Pixel Prediction. In: Advanced Concepts for Intelligent Vision Systems (ACIVS). pp. 431–440. Springer (2016)

20. Grana, C., Bolelli, F., Baraldi, L., Vezzani, R.: YACCLAB - Yet Another Connected Components Labeling Benchmark. In: 23rd International Conference on Pattern Recognition (ICPR). pp. 3109–3114. IEEE (2016)
21. Grana, C., Borghesani, D., Cucchiara, R.: Optimized Block-based Connected Components Labeling with Decision Trees. IEEE Transactions on Image Processing **19**(6), 1596–1609 (2010)
22. Hawick, K.A., Leist, A., Playne, D.P.: Parallel graph component labelling with GPUs and CUDA. Parallel Computing **36**(12), 655–678 (2010)
23. He, L., Chao, Y., Suzuki, K.: A Linear-Time Two-Scan Labeling Algorithm. In: International Conference on Image Processing. vol. 5, pp. 241–244 (2007)
24. He, L., Zhao, X., Chao, Y., Suzuki, K.: Configuration-Transition-Based Connected-Component Labeling. IEEE Transactions on Image Processing **23**(2), 943–951 (2014)
25. Huiskes, M.J., Lew, M.S.: The MIR Flickr Retrieval Evaluation. In: MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval. ACM, New York, NY, USA (2008)
26. Kalentev, O., Rai, A., Kemnitz, S., Schneider, R.: Connected component labeling on a 2D grid using CUDA. Journal of Parallel and Distributed Computing **71**(4), 615–620 (2011)
27. Komura, Y.: GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen–Wang multi-cluster spin flip algorithm. Computer Physics Communications **194**, 54–58 (2015)
28. Lelore, T., Bouchara, F.: FAIR: A Fast Algorithm for Document Image Restoration. IEEE transactions on pattern analysis and machine intelligence **35**(8), 2039–2048 (2013)
29. Lewis, D., Agam, G., Argamon, S., Frieder, O., Grossman, D., J.Heard: Building a Test Collection for Complex Document Information Processing. In: Proc. 29th Annual Int. ACM SIGIR Conference. pp. 665–666 (2006)
30. Maltoni, D., Maio, D., Jain, A., Prabhakar, S.: Handbook of Fingerprint Recognition. Springer Science & Business Media (2009)
31. Oliveira, V.M., Lotufo, R.A.: A study on connected components labeling algorithms using GPUs. In: SIBGRAPI. vol. 3, p. 4 (2010)
32. Playne, D.P., Hawick, K.: A New Algorithm for Parallel Connected-Component Labelling on GPUs. IEEE Transactions on Parallel and Distributed Systems **29**(6), 1217–1230 (2018)
33. Pollastri, F., Bolelli, F., Paredes, R., Grana, C.: Improving Skin Lesion Segmentation with Generative Adversarial Networks. In: 2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS). pp. 442–443. IEEE (2018)
34. Pollastri, F., Bolelli, F., Paredes, R., Grana, C.: Augmenting data with GANs to segment melanoma skin lesions. Multimedia Tools and Applications pp. 1–18 (2019)
35. Rosenfeld, A., Pfaltz, J.L.: Sequential Operations in Digital Picture Processing. Journal of the ACM **13**(4), 471–494 (1966)
36. Source Code of the Proposed Strategy, https://github.com/prittt/YACCLAB, accessed on 2019-05-06
37. Wu, K., Otoo, E., Suzuki, K.: Two Strategies to Speed up Connected Component Labeling Algorithms. Tech. Rep. LBNL-59102, Lawrence Berkeley National Laboratory (2005)
38. Zavalishin, S., Safonov, I., Bekhtin, Y., Kurilin, I.: Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU. Electronic Imaging **2016**(2), 1–7 (2016)