Optimizing GPU-Based Connected Components Labeling Algorithms

Stefano Allegretti, Federico Bolelli, Michele Cancilla, Costantino Grana
Università degli Studi di Modena e Reggio Emilia
Email: {name.surname}@unimore.it

***Abstract*—Connected Components Labeling (CCL) is a fundamental image processing technique, widely used in various application areas. Computational throughput of Graphical Processing Units (GPUs) makes them eligible for such a kind of algorithms. In the last decade, many approaches to compute CCL on GPUs have been proposed. Unfortunately, most of them have focused on 4-way connectivity neglecting the importance of 8-way connectivity. This paper aims to extend state-of-the-art GPU-based algorithms from 4 to 8-way connectivity and to improve them with additional optimizations. Experimental results revealed the effectiveness of the proposed strategies.**

***Index Terms*—Connected Components Labeling, Parallel Computing, GPU**

I. Introduction

Connected Components Labeling (CCL) is a fundamental image processing algorithm that extracts connected components (objects) from an input binary image, transforming it into a symbolic one, in which all pixels of the same object are given the same label, typically an integer number. CCL is required whenever a computer program needs to identify independent components, and it represents the preliminary operation in most of the computer vision research fields *e.g.* text analysis, medical imaging and video surveillance. Differently from other tasks, CCL algorithms should provide an exact solution, and the main difference among them is the execution time. Moreover, given that labeling is the base step of most real time applications, it is required to be as fast as possible. For this reasons, the proposals of the last twenty years focused on the performance optimization of both sequential and parallel algorithms.

For what concerns sequential algorithms, a significant improvement has been introduced with the use of array-based *Union-Find* approach for label equivalences resolution [1], [2]. Further improvements were given by the introduction of block and run-based scans, to reduce the number of memory accesses and thus execution time [3], [4], by the use of pixel prediction to exploit the information provided by already seen pixels, removing the need to check them again [5], [6], and exploiting techniques of footprint code compression [7], [8].

The development of parallel algorithms to solve common problems is of growing interest, lead by the fast development of parallel hardware architectures. A simple process to parallelize sequential algorithm on CPU, consists of dividing the input image into horizontal stripes and computing labeling separately on each of them [9].

Computational throughput of Graphical Processing Units (GPUs) makes them eligible for many image processing methods that can be easily implemented on such architectures. The use of GPUs usually provides good performance, but this does not happen for less regular problems, such as CCL. Indeed, labeling GPU implementations tend to achieve comparable performance with respect to the CPU ones [10]. However, data transfer between device and host is very expensive, usually higher than the cost of labeling procedure. Therefore, all the applications that entirely execute on GPU would benefit from an optimized GPU-based labeling algorithm, removing the need for data transfers.

In the last decade, many approaches to compute GPU-CCL have been proposed. Label Equivalence (LE), introduced by Hawick in [11], is an iterative algorithm that propagates the minimum label through each connected component by means of a parallel procedure repeated until convergence, similarly as the first sequential algorithm proposed by Haralick [12]. Block Equivalence (BE) is a variation of LE that only works for 8-way connectivity, and speeds up the process by implementing the block based approach proposed for sequential algorithms by Grana [3]. The Union Find method (UF) by Oliveira *et al.* adapts the array-based *union-find* approach to a parallel scenario [13], avoiding the necessity of multiple iterations. Komura Equivalence (KE) is a very similar approach to UF, that implements an optimized initialization phase to save work in the following steps [14].

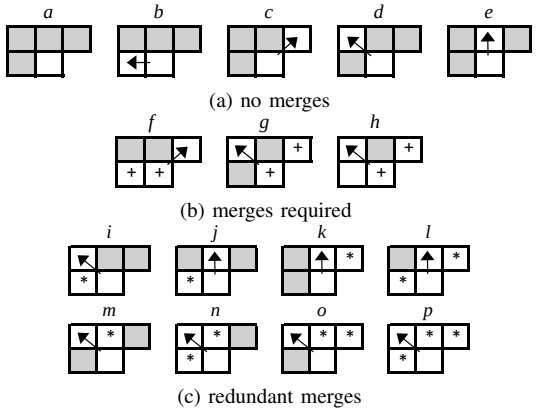


Fig. 1. Possible neighbourhood configuration of a foreground pixel. Grey squares are background and white are foreground. Arrows show connections performed in *initialization* phase, crosses identify mergers that must be performed during *reduction* phase, stars highlight pixels to whom the merging operation can be delegated.

Algorithm 1 Union Find procedures. I is input image, L is both union-find array and output label image.

```
1: function FIND(L, index)
2:     label := L[index]
3:     while label − 1 ≠ index do
4:         index := label − 1
5:         label := L[index]
6:     return index

7: procedure UNION(L, a, b)
8:     done := false
9:     while done = false do
10:        a := Find(L, a)
11:        b := Find(L, b)
12:        done := (a = b)
13:        if a = b then
14:            done := true
15:        else
16:            if done := false and a > b then
17:                Swap(a, b)
18:            old := atomicMin(&L[b], a + 1)
19:            done := (old = b + 1)
20:            b := old − 1
```

Many computer vision tasks require 8-way connectivity CCL. Despite this, most of the aforementioned algorithms have focused on 4-way connectivity only. According to the *Gestalt Theory* of perception, our senses operate the closure property perceiving objects as a whole, even if they are loosely connected as happens in the 8-way connectivity case [3].

This paper aims to improve state-of-the-art GPU algorithms with three main contributions: *(i)* extension from 4 to 8-way connectivity of UF and KE methods *(ii)* optimization of KE 8-connected, and *(iii)* exhaustive evaluation of the proposed strategies and comparison with state-of-the-art using a public benchmark.

The rest of this paper is organized as follows. Section II describes the relevant GPU-based algorithms that will be then extended in Section III. In Section IV, a set of experiments is reported discussing and motivating obtained results. Finally, in Section V we draw conclusions.

II. Related Work

A. Union Find Algorithm

Oliveira *et al.* proposed in [13] a GPU-CCL algorithm based on a *union-find* approach. The *union-find* data structure was firstly applied to CCL problems by Dillencourt *et al.* in [15]. It consists of a forest of trees that supports two basic operations: *find* and *union*.

The *find* function takes a node of a tree as input and returns its root as output.

The *union* procedure takes two nodes as inputs and joins together the trees they belong to, by setting the first tree root as the father of the second one.

When *union-find* is applied to CCL, each pixel in the image matches a node in the data structure. The final goal is to build a single tree for each connected component, starting with every foreground pixel in its own tree and taking advantage of *union* operations for merging trees of connected pixels.

Algorithm 2 Union Find kernels. I is input image, L is both union-find array and output label image. Checks on image borders are not shown.

```
1: kernel INITIALIZATION(I, L, r, c)
2:     if I[r, c] = 1 then
3:         L[r, c] := LinearIndex(r, c) + 1
4:     else
5:         L[r, c] := 0

6: kernel MERGE(I, L, r, c)
7:     if I[r, c] = 1 then
8:         index := LinearIndex(r, c)
9:         for all k_i := Neighbours(index) do
10:            if k_i < index and I[k_i] = 1 then
11:                Union(L, index, k_i)

12: kernel ANALYSIS(I, L, r, c)
13:    if I[r, c] = 1 then
14:        L[r, c] := Find(L, LinearIndex(r, c)) + 1
```

The *union-find* forest can be stored in memory as an array, where each node is represented by an index, and the value stored at that index represents its parent node. Root nodes have their own indexes stored in the array. Values in said array can also be interpreted as pixel labels. This way, when the goal of matching every single connected component to a separate tree has been achieved, a flattening operation that links each node of every tree directly to the root is sufficient to assign the same label to every pixel in the tree, thus completing the CCL task. The *union-find* array can at this point be interpreted as the label image, saving the need for a separate data structure in memory. In order to distinguish between background and foreground pixels, in our implementation we actually write, for each node, the index of its parent + 1. This way, every foreground pixel has a positive label, and 0 is assigned to background ones.

A possible implementation of *find* and *union* procedures is shown in Alg. 1. Atomic operations are used in *union* to avoid problems concerning the simultaneous updates performed by different threads.

UF algorithm is based on the *union-find* data structure and it consists of three kernels: *initialization*, *merge* and *analysis*, described in Alg. 2. Each of them is launched on a number of threads equal to the image size, and each thread is assigned a pixel, which we will refer to as the *active pixel*.

During *initialization* phase, the *union-find* structure is initialized, with each node in its own tree.

During *merge* phase, each thread working on a foreground pixel analyzes its neighbourhood, and for every foreground neighbour performs a *union* with the active pixel. In the pseudo code, the *neighbours* function is used to get the neighbours of a pixel, which depend on the chosen connectivity.

After *merge* phase the *analysis* kernel performs the flatten-

Algorithm 3 4-way connectivity Komura equivalence algorithm kernels. I is input image and L is labels matrix. Checks on image borders are not showed.

```
1:  kernel INITIALIZATION(I, L, r, c)
2:      if I[r, c] = 0 then
3:          L[r, c] := 0
4:      else
5:          index := LinearIndex(r, c)
6:          label := index
7:          for all k_i := Neighbours(index) do
8:              if k_i < label and I[k_i] = 1 then
9:                  label := k_i
10:         L[r, c] := label + 1

11: kernel REDUCTION(I, L, r, c)
12:     if I[r, c] = 1 and I[r, c − 1] = 1 then
13:         a := LinearIndex(r, c)
14:         b := LinearIndex(r, c − 1)
15:         Reduce(L, a, b)

16: procedure REDUCE(L, a, b)
17:     a := Find(L, a)
18:     b := Find(L, b)
19:     done := (a = b)
20:     if a > b then
21:         Swap(a, b)
22:     while done = false do
23:         old := atomicMin(&L[b], a + 1)
24:         if old = a + 1 then
25:             done := true
26:         else if old > a + 1 then
27:             b := old − 1
28:         else if old < a + 1 then
29:             b := a
30:             a := old − 1
```

ing of trees, completing the labeling task. The entire algorithm is first performed on rectangular blocks the image is divided into. Then, *merge* kernel is performed on border pixels only, and a final *analysis* is launched over the whole image. The original algorithm uses 4-way connectivity, and we extended it to 8-way connectivity by simply adding the diagonal directions to the neighbourhood of a pixel, in *merge* kernel.

B. Komura Equivalence Algorithm

The Komura Equivalence algorithm [14], which employs a 4-way connectivity, introduces additional improvements to the UF algorithm. This method consists of four steps: *initialization*, *analysis*, *label reduction*, and *analysis* again, which are detailed in the following:

(a) The *initialization* kernel, shown at line 1 of Alg. 3, sets the label of each pixel with the smallest linear address of its neighbours, avoiding the commonly used write operation which initializes the output image with increasing labels. Since the smallest address is chosen, north pixel is prioritized over west one. Same as Union Find, in our implementation we add 1 to each label, to make sure that foreground pixels always are assigned positive values. Background pixels are given label 0 instead. This small change also slightly affects the other kernels.

(b) The *analysis* kernel is equal to the Union Find procedure of the same name, and achieves the goal of collapsing the trees created in the previous step to their roots.

(c) In *reduction* kernel, shown at line 11 of Alg. 3, every thread merges the active pixel tree with the one containing the pixel to the west, in the case it is foreground. Indeed, if both north and west pixels are foreground, only north has been chosen during *initialization* phase. The merging is performed by the *reduce* procedure, which has the same effect of *union*.

Algorithm 4 8-way connectivity Komura equivalence reduction kernel - r and c are thread row and column, I is input image and L is labels matrix. Checks on image borders are not showed.

```
1: kernel REDUCTION(I, L, r, c)
2:     if I[r, c] = 1 and I[r − 1, c] = 0 then
3:         k := LinearIndex(r, c)
4:         NW := I[r − 1, c − 1]
5:         if NW = 1 and I[r − 1, c + 1] = 1 then
6:             Union(L, k, LinearIndex(r − 1, c + 1))
7:         if NW = 0 and I[r, c − 1] = 1 then
8:             Union(L, k, LinearIndex(r, c − 1))
```

III. PROPOSED ALGORITHM

Our new algorithm is an adaptation of KE to a 8-way connectivity scenario. It keeps the same structure as the original 4-way variation—*initialization*, *analysis* and *reduction* kernels are preserved. A few changes must be introduced to the operations these kernels perform though.

The *initialization* kernel has the same structure as the 4-way connectivity variation, but the *neighbours* function must also return diagonal neighbours. Since we still assign the smallest neighbour address, the priority order is north-west→north→north-east→west.

The *analysis* kernel is the same as the 4-way connectivity variation. Its job of following the equivalence chains to the root of trees is not indeed tied to pixel connectivity.

The *reduction* kernel is still responsible for merging trees that result separate after the first two phases, but are connected together by at least a couple of foreground pixels nonetheless. This occurrence is rather common, considering that in the *initialization* phase each pixel can only choose one among possibly four different neighbour indexes. Each possible neighbourhood configuration of a foreground pixel is shown in Fig. 1. Configurations in Fig. 1a do not need additional merging between pixel trees. In fact, connected pixels in the mask have already been linked together in *initialization* phase. Conversely, crosses in Fig. 1b identify merging operations that must be performed between the tree containing the active pixel and the one containing the pixel

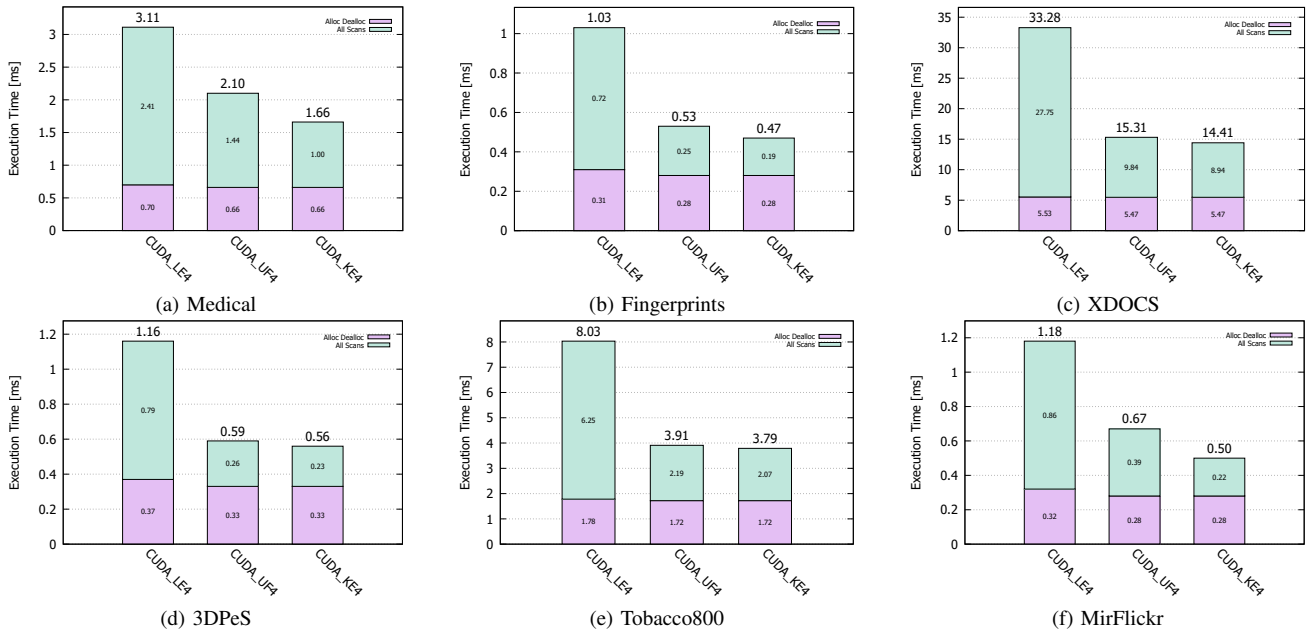


Fig. 2. 4-way connectivity experimental results obtained on a Nvidia Quadro K2200 GPU with the YACCLAB benchmark. For each dataset and algorithm the average execution time in ms is reported (lower is better).

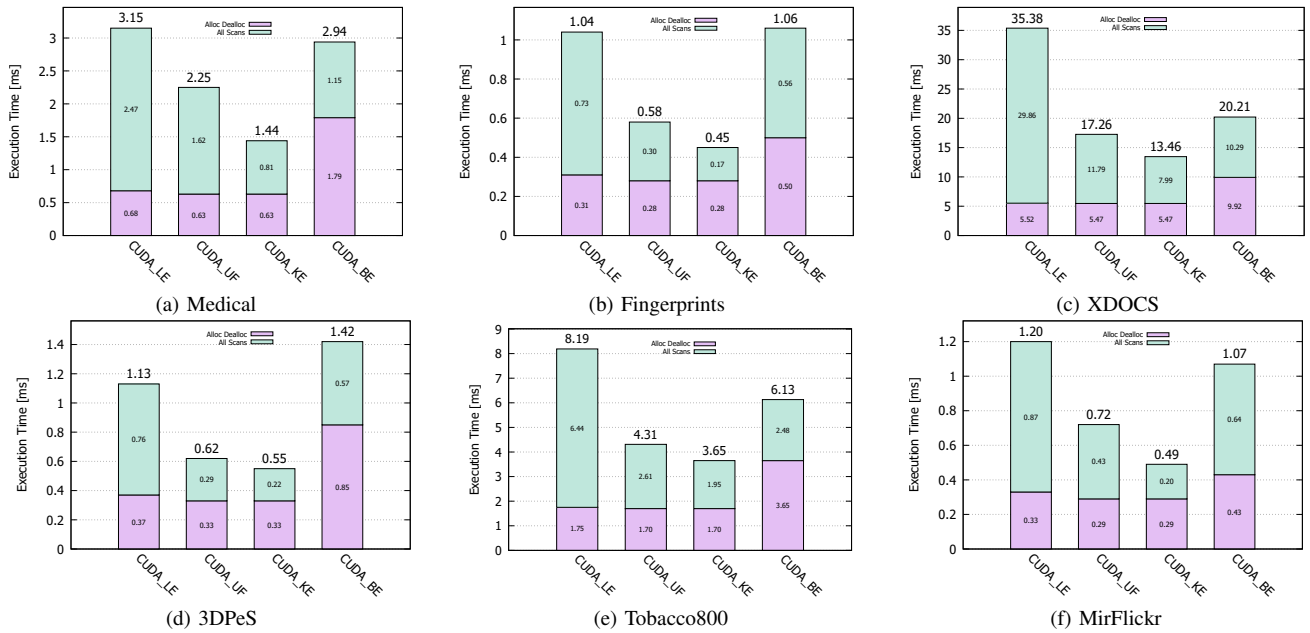
to the west (f) or to the north-east (g and h). Those two trees may have already been joined together by another pixel outside from the neighbourhood mask, but since we do not know it, we must join them anyway. A further exploration outside of the mask is not worth the effort, because of the large amount of memory accesses it would introduce. Each configuration in Fig. 1c exhibit one or more couple of trees that need to be merged as well. Nevertheless, each of those tree connections also show up in the neighbourhood mask of at least another foreground pixel, marked with a star (*e.g.* in i, the connection between the west pixel and the north-west pixel also appears in the neighbourhood mask of the west pixel). This means that, when we face a mask configuration included in the aforementioned set, we are not forced to join neighbour trees. Indeed, other threads can perform the same operation. Consequently, to save unnecessary calls to the *reduce* function, each thread in *reduction* kernel only joins together connected trees if the neighbour configuration of the active pixel belongs to the *merge required* set, because those are the only cases we cannot delegate the merging operation to other threads. We use a small decision tree to identify said configurations while limiting the number of memory accesses to a minimum. Actual memory accesses range from a minimum of 1 to a maximum of 4 per thread, but their cost is compensated by the fact that *reduce* function, which in turn performs a variable number of memory accesses, is only called on 3 neighbourhood configurations out of 16 possibilities.

We also introduced another slight improvement to *reduction* kernel. As Playne noticed in [10], the original *reduce* function always performs at least two *atomicMin*, even in the case no other thread interfered. We thus replaced the *reduce* procedure with *union*, which produces the same result without employing unnecessary atomic operations. The pseudo code of *reduction* is showed in Alg. 4. After the *reduction* step, equally to the 4-way connectivity variation, a final *analysis* is needed to assign every pixel the root label of its tree.

IV. Experimental Results

In this Section, the performance of the proposed strategies are evaluated and compared with other state-of-the-art GPU-CCL algorithms. There are many variables that could influence the performance of an algorithm in terms of execution time: the machine architecture and the operative system on which test are performed, the adopted compiler, code implementation and last but not least the data on which algorithms are tested. In order to produce a fair comparison, our proposals have been evaluated with YACCLAB [16], [17]: an open source *C++* benchmarking system, based on OpenCV and available in [18], which lets researchers test CCL algorithms under variable points of view and over a common background. The benchmark provides a set of datasets covering real case scenarios for CCL, from which we select the most significant ones: *MIRflickr* [19], *Medical* [20], *Tobacco800* [21], *XDOCS* [22], [23], *Fingerprints* [24], and *3DPeS* [25]. A complete description of these datasets can be found in the original paper.

The algorithms have been implemented with Visual Studio 2017 using CUDA 9.2 and compiled for x64 architectures employing MSVC 19.13.26132 and NVCC V9.2.148 compilers with optimizations enabled. Tests are performed on an Intel Core i7-4770 CPU @ 3,40 GHz (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache) with 16 GB of RAM available, and with a Quadro K2200 NVIDIA GPU with Maxwell architecture, 640 CUDA cores and 4 GB of memory. Both 4-way and 8-way connectivity implementations



(a) Medical (b) Fingerprints (c) XDOCS

(d) 3DPeS (e) Tobacco800 (f) MirFlickr

Fig. 3. 8-way connectivity experimental results obtained on a Nvidia Quadro K2200 GPU with the YACCLAB benchmark. For each dataset and algorithm the average execution time in ms is reported (lower is better).

are provided for each algorithm except for BE, which employs a block-based strategy suitable for 8-way connectivity only.

In Fig. 2 and Fig. 3 the average execution time for each algorithm and dataset is reported, respectively for 4-way and 8-way connectivity. Bar charts report separately the time needed for allocating data structures and the execution time required by the labeling procedure. According to [17], the allocation time reported in this charts is an upper bound of the real allocation time required by an algorithm on a given environment.

Focusing on Fig. 2, the allocation time is the same for each strategy, but for LE. Indeed, all strategies must allocate memory for the output image and LE requires an additional boolean flag to stop iteration when no change on the output image occurs. A similar conclusion can be drawn for Fig. 3, but the allocation time of BE is almost twice the others, since it relies on additional matrices to store equivalences between blocks and their labels.

The execution time of the LE core phases is always the worst, given that it requires multiple iterations over the input image to update the output one until convergence. The block scan approach introduced by BE allows to reduce by a factor of four the operations required by LE, thus reducing the labeling time at the expense of allocation step. In most cases, using blocks to scan images improve the performance, except for images with very low foreground density such as in 3DPeS.

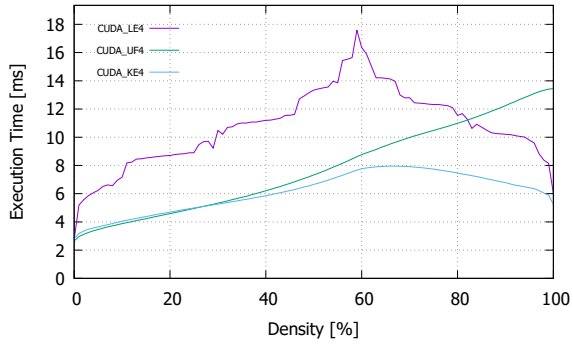
Anyway, multiple scan approaches are always worse than others. As can be seen from charts in Fig. 3, the reduction in the number of iterations allowed by UF always improves performance. Moreover, KE allows the removal of the initialization phase of UF, always providing the best performance on real cases datasets.

In order to highlight strengths and weaknesses of the algorithms, and following a common approach in literature [1]–[3], an additional test has been performed on images with increasing foreground density (Fig. 4). These images are pseudo-randomly generated using the Mersenne Twister MT19937 [26], as described in [17]. Resulting images have a size of 2048×2048 and a density varying from 0% to 100% with step of 1%.

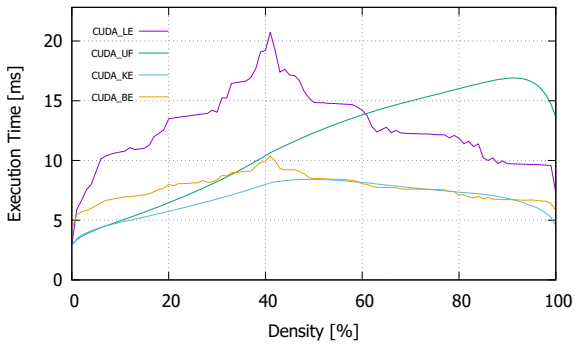
Considering both 4 and 8-way connectivity, the LE approach has an increasing trend in the execution time up to 40% of foreground density, and a decreasing one after 60%. In the middle densities the execution time reaches the maximum value. This is linked to the number of iterations required by the labeling procedure to converge, as shown in Tab. I. More specifically, the pixels patterns which cause the highest number of iterations appear near 60% of foreground density for 4-way connectivity variation and near 40% for the 8-way one. The BE algorithm has a similar behavior, albeit with better performance. This confirms results on real cases datasets described before.

On the other hand, the execution time of the UF approach grows with foreground density. This is because each pixel thread has to perform one *merge* operation for each connected pixel, and the number of those pixels is linked to density. The more merges there are, the more memory accesses and atomic operations are performed.

Performance gap between UF and KE significantly increases from 4 to 8-way connectivity. This can be easily explained considering the optimization we introduced on the 8-connected variation of the algorithm. Indeed, thanks to *reduction* procedure we are able to remove unnecessary operations, thus reducing overall execution time.



(a) 4-way connectivity



(b) 8-way connectivity

Fig. 4. Execution time of (a) 4-way and (b) 8-way connectivity algorithms on images of increasing density.

TABLE I
AVERAGE NUMBER OF ITERATIONS REQUIRED BY LE IMPLEMENTING 8-WAY CONNECTIVITY ON IMAGES OF INCREASING DENSITY.

density (%)	0	10	20	30	40	50	60	70	80	90	100
iterations	1.0	4.0	5.0	5.0	7.2	5.0	5.0	4.0	3.9	3.0	2.0

V. CONCLUSIONS

In this paper the problem of connected components labeling on GPUs is explored. Many approaches have been introduced over the years but we focused our study on four main methods: Label Equivalence, Block Equivalence, Union Find and Komura Equivalence. All this algorithms have been reimplemented in their original 4-way connectivity version, but BE that is suitable for 8-connected applications only. Moreover, we adapted both the Union Find and Komura Equivalence strategies to the 8-way connectivity, and introduced an optimization on the second one which reduces the memory accesses, improving the performance. Experimental results obtained with an open-source benchmark for CCL, revealed the effectiveness of our proposals. The source code of described algorithms is available at [18], so anyone can download, test it on his own setup, and verify our claims.

REFERENCES

[1] K. Wu, E. Otoo, and K. Suzuki, "Two Strategies to Speed up Connected Component Labeling Algorithms," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-59102, 2005.

[2] L. He, Y. Chao, and K. Suzuki, "A Linear-Time Two-Scan Labeling Algorithm," in *International Conference on Image Processing*, vol. 5, 2007, pp. 241–244.

[3] 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, 2010.

[4] L. He, Y. Chao, and K. Suzuki, "A Run-Based Two-Scan Labeling Algorithm," *IEEE Trans. Image Process.*, vol. 17, no. 5, pp. 749–756, 2008.

[5] 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, 2014.

[6] C. Grana, L. Baraldi, and F. Bolelli, "Optimized Connected Components Labeling with Pixel Prediction," in *Advanced Concepts for Intelligent Vision Systems*, 2016.

[7] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected Components Labeling on DRAGs," in *International Conference on Pattern Recognition*, 2018.

[8] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes," in *24rd International Conference on Pattern Recognition Workshops*, 2018.

[9] F. Bolelli, M. Cancilla, and C. Grana, "Two More Strategies to Speed Up Connected Components Labeling Algorithms," in *International Conference on Image Analysis and Processing*. Springer, 2017, pp. 48–58.

[10] 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, June 2018.

[11] K. Hawick, A. Leist, and D. Playne, "Parallel graph component labelling with gpus and cuda," *Parallel Computing*, vol. 36, no. 12, pp. 655 – 678, 2010.

[12] R. Haralick, "Some neighborhood operators," in *Real-Time Parallel Computing*. Springer, 1981, pp. 11–35.

[13] V. M. Oliveira and R. A. Lotufo, "A study on connected components labeling algorithms using GPUs," in *SIBGRAPI*, vol. 3, 2010, p. 4.

[14] Y. Komura, "GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen–Wang multi-cluster spin flip algorithm," *Computer Physics Communications*, vol. 194, pp. 54 – 58, 2015.

[15] M. B. Dillencourt, H. Samet, and M. Tamminen, "A General Approach to Connected-Component Labeling for Arbitrary Image Representations," *Journal of the ACM*, vol. 39, no. 2, pp. 253–280, 1992.

[16] C. Grana, F. Bolelli, L. Baraldi, and R. Vezzani, "YACCLAB - Yet Another Connected Components Labeling Benchmark," in *23rd International Conference on Pattern Recognition*. ICPR, 2016.

[17] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Toward reliable experiments on the performance of Connected Components Labeling algorithms," *Journal of Real-Time Image Processing*, pp. 1–16, 2018.

[18] The YACCLAB Benchmark. [Online]. Available: https://github.com/prittt/YACCLAB

[19] M. J. Huiskes and M. S. Lew, "The MIR Flickr Retrieval Evaluation," in *MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval*. New York, NY, USA: ACM, 2008.

[20] F. Dong, H. Irshad, E.-Y. Oh *et al.*, "Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast," *PloS one*, vol. 9, no. 12, p. e114885, 2014.

[21] G. Agam, S. Argamon, O. Frieder, D. Grossman, and D. Lewis, "The Complex Document Image Processing (CDIP) Test Collection Project," Illinois Institute of Technology, 2006.

[22] F. Bolelli, G. Borghi, and C. Grana, "Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features," in *19th International Conference on Image Analysis and Processing*, 2017.

[23] F. Bolelli, "Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text," in *Italian Research Conference on Digital Libraries*. Springer, 2017, pp. 45–55.

[24] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of fingerprint recognition*. Springer Science & Business Media, 2009.

[25] D. Baltieri, R. Vezzani, and R. Cucchiara, "3DPeS: 3D People Dataset for Surveillance and Forensics," in *Proceedings of the 2011 joint ACM workshop on Human gesture and behavior understanding*. ACM, 2011, pp. 59–64.

[26] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.