This is the peer reviewd version of the followng article:

How does Connected Components Labeling with Decision Trees perform on GPUs? / Allegretti, Stefano; Bolelli, Federico; Cancilla, Michele; Pollastri, Federico; Canalini, Laura; Grana, Costantino. - 11678:(2019), pp. 39-51. (Intervento presentato al convegno International Conference on Computer Analysis of Images and Patterns tenutosi a Salerno, Italy nel Sep 3-5) [10.1007/978-3-030-29888-3_4].

Springer, Cham

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

07/05/2024 08:45

(Article begins on next page)

How does Connected Components Labeling with Decision Trees perform on GPUs?

Stefano Allegretti, Federico Bolelli, Michele Cancilla, Federico Pollastri, Laura Canalini, 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 the problem of Connected Components Labeling (CCL) in binary images using Graphic Processing Units (GPUs) is tackled by a different perspective. In the last decade, many novel algorithms have been released, specifically designed for GPUs. Because CCL literature concerning sequential algorithms is very rich, and includes many efficient solutions, designers of parallel algorithms were often inspired by techniques that had already proved successful in a sequential environment, such as the *Union-Find* paradigm for solving equivalences between provisional labels. However, the use of decision trees to minimize memory accesses, which is one of the main feature of the best performing sequential algorithms, was never taken into account when designing parallel CCL solutions. In fact, branches in the code tend to cause thread divergence, which usually leads to inefficiency. Anyway, this consideration does not necessarily apply to every possible scenario. Are we sure that the advantages of decision trees do not compensate for the cost of thread divergence? In order to answer this question, we chose three well-known sequential CCL algorithms, which employ decision trees as the cornerstone of their strategy, and we built a data-parallel version of each of them. Experimental tests on real case datasets show that, in most cases, these solutions outperform state-of-the-art algorithms, thus demonstrating the effectiveness of decision trees also in a parallel environment.

Keywords: Image processing · Connected Components Labeling · Parallel Computing · GPU

1 Introduction

In the last decade, the great advance of Graphic Processing Units (GPUs) pushed the development of algorithms specifically designed for data parallel environments. On GPUs, threads are grouped into packets (warps) and run on *single-instruction, multiple-data* (SIMD) units. This structure, called SIMT (*single-instruction, multiple threads*) by NVIDIA, allows to execute the same instruction on multiple threads in parallel, thus offering a potential efficiency advantage [27]. It is commonly known that most sequential programs, when ported on

GPU, break the parallel execution model. Indeed, only applications characterized by regular control flow and memory access patterns can benefit from this architecture [11].

During execution, each processing element performs the same procedure (kernel) on different data. All cores in a warp run like lock-step at same instruction, but next instruction can be fetched only when the previous one has been completed by all threads. If an instruction requires different amounts of time in different threads, such as when branches cause different execution flows, then all threads have to wait, decreasing the efficiency of the lock-step. This is the reason why intrinsically sequential algorithms must be redesigned to reduce/remove branches and fit GPU logic. But is this always necessary? Would this always improve performance? In this paper, focusing on the *Connected Component Labeling* (CCL) problem, we demonstrate that the best performing sequential algorithms can be easily implemented on GPU without changing their nature, and on real case scenarios they may perform significantly better than state-of-the-art algorithms specifically designed for GPUs. Given that CCL is a well-defined problem that provides an exact solution, the main difference among algorithms is the execution time. This is why the proposals of the last years focused on the performance optimization of both sequential and parallel algorithms.

The rest of this paper is organized as follows. In Section 2 the problem of labeling connected components on binary images is defined and the notation used throughout the paper is introduced. Section 3 resumes state-of-the-art algorithms for both CPUs and GPUs, describing strengths and weaknesses of each proposal. In Section 4 we describe how sequential algorithms have been implemented on GPU using CUDA. Then, to demonstrate the effectiveness of the proposed solution, an exhaustive set of experiments is reported in Section 5. Finally, Section 6 draws some conclusions.

2 Problem Definition

Given I, an image defined over a two dimensional rectangular lattice $\mathcal{L}$, and $I(p)$ the value of pixel $p \in \mathcal{L}$, with $p = (p_x, p_y)$, we define the *neighborhood* of a pixel as follows:

$$\mathcal{N}(p) = \{q \in \mathcal{L} \mid \max(|p_x - q_x|, |p_y - q_y|) \leq 1\} \quad (1)$$

Two pixels, p and q, are said to be *neighbors* if $q \in \mathcal{N}(p)$, that implies $p \in \mathcal{N}(q)$. From a visual perspective, p and q are *neighbors* if they share an edge *or* a vertex. The set defined in Eq. 1 is called 8-neighborhood of p.

In a binary image, meaningful regions are called *foreground* ($\mathcal{F}$), and the rest of the image is the *background* ($\mathcal{B}$). Following a common convention, we will assign value 1 to foreground pixels, and value 0 to background.

The aim of connected components labeling is to identify disjoint objects, composed of foreground pixels. So, given two foreground pixels $p, q \in \mathcal{F}$, the relation of *connectivity* $\diamond$ can be defined as:

$$p \diamond q \Leftrightarrow \exists \{s_i \in \mathcal{F} \mid s_1 = p, \ s_{n+1} = q, \ s_{i+1} \in \mathcal{N}(s_i), \ i = 1, \ldots, n\} \quad (2)$$

We say that two pixels p, q are *connected* if the condition $p \diamond q$ is true. The above definition means that a path of connected foreground pixels exists, from p to q. Moreover, since pixel connectivity satisfies the properties of *reflexivity*, *symmetry* and *transitivity*, $\diamond$ is an equivalence relation. Equivalence classes based on $\diamond$ relationship are called *Connected Components* (CC).

Algorithm 1 *Union-Find* functions. P is the *Union-Find* array, a and b are provisional labels.

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

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

The pixel connectivity associated to the neighborhood of Eq. 1 is usually called 8-connectivity, in contrast to the 4-connectivity, in which only pixels sharing an edge are considered neighbors and then connected. According to the *Gestalt Psychology*, CCL usually requires 8-connectivity, that is why we focus on it.

CCL algorithms assign a positive unique label (integer number) to all foreground pixels of a connected component of the image, marking background pixels with 0. Depending on the strategy used to perform the task, it can happen, at some point of the algorithm, that two connected pixels p, q are assigned different provisional labels. In that case, the two labels are said to be *equivalent*. Equivalent labels must be eventually modified, so that they result equal to a certain representative one, through a process of equivalence resolution.

Several strategies exist to solve equivalences, among which one of the most efficient exploits *Union-Find*, firstly applied to CCL by Dillencourt *et al.* [14]. The *Union-Find* data structure provides convenient procedures to keep track of a partition $\mathcal{P}$ of the set $\mathcal{S}$ of provisional labels. Two basic functions are defined on labels $a, b \in \mathcal{S}$:

- $Union(a, b)$: merges the subsets containing a and b.
- $Find(a)$: returns the representative label of the subset containing a.

The recording of an equivalence between labels is performed through a call to $Union$, while $Find$ eases the resolution. The partition is represented as a forest of rooted trees, with a tree for every subset of $\mathcal{S}$. The forest can be represented as an array P, where $P[a]$ is the father node of a. A possible implementation of *Union-Find* functions is given in Algorithm 1.

3 Connected Components Labeling Algorithms

Efficiency of connected components labeling is critical in many real-time applications [13, 29, 30], and this is the reason why many strategies have been proposed for efficiently addressing the problem [10].

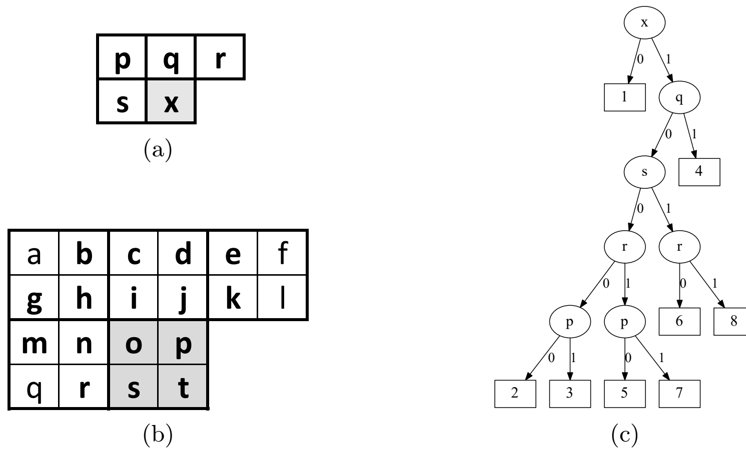


Fig. 1. (a) is the Rosenfeld mask used by SAUF to compute the label of pixel x during the first scan and (b) is the Grana mask used by BBDT and DRAG to compute the label of pixels o, p, s and t. Finally, (c) is one of the optimal decision trees associated to the Rosenfeld mask. Here ellipses (nodes) represent conditions to check and rectangles (leaves) are the actions to perform.

Traditionally, on sequential machines a two scan algorithm is employed. It is composed of three steps:

– *First scan*: scans the input image using a mask of already visited pixels, such as the one in Fig. 1a, 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 replacing provisional with final labels.

When statistics about connected components are required (e.g. area, perimeter, circularity, centroid), the second scan can be avoided, reducing the total execution time.

Different solutions allowed performance improvements by avoiding redundant memory accesses [19, 21, 34]. One of the first improvements is the Scan Array-based Union-Find (SAUF) proposed by Wu *et al.* in [34]. This is a reference algorithm because of its very good performance and ease of understanding. The optimization introduced with SAUF reduces the number of neighbors visited during the first scan using a decision tree, such as the one shown in Fig. 1c. 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 we do not even need to check their values.

Since 8-connectivity is usually employed to describe foreground objects, this algorithm was extended in [18, 19] with the introduction of 2×2 blocks, in which

all foreground pixels share the same label. In this case, the scanning mask is bigger (Fig. 1b), leading to a large number of combinations that produces a complex decision tree, whose construction is much harder. In [20] an optimal strategy to automatically build the decision tree by means of a dynamic programming approach has been proposed and demonstrated. This approach is commonly known as Block Based Decision Tree scanning (BBDT).

He *et al.* [21] were the first to realize that, thanks to the sequential approach taken, when the mask shifts horizontally through the image, it contains some pixels that were already inside the mask in the previous iteration. If those pixels were checked in the previous step, a repeated reading can be avoided. They addressed this problem condensing the information provided by the values of already seen pixels in a configuration state, and modeled the transition with a finite state machine. In [16] a general paradigm to leverage already seen pixels, which combines configuration transitions with decision trees, was proposed. This approach has again the advantage of saving memory accesses.

In [5, 8], 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 thus 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 run-time performance.

When moving to parallel architectures, CCL can be easily obtained by repeatedly propagating the minimum label to every pixel neighbor. Nevertheless, much better alternatives exist. Oliveira *et al.* [28] were the first to make use of the *Union-Find* approach in GPU. In their algorithm, the so called Union Find (UF), the output image is initialized with sequential values. Then, *Union-Find* primitives are used to join together trees of neighbor pixels. Finally, a flattening of trees updates the output image, completing the task. The algorithm is firstly performed on rectangular tiles, and then large connected components are merged in a subsequent step.

Optimized Label Equivalence (OLE) [23] is an iterative algorithm that records *Union-Find* trees in the output image itself. The algorithm consists of three kernels that are repeated in sequence until convergence. They aim at propagating the minimum label through each connected component, flattening equivalence trees at every step.

Zavalishin *et al.* [35] proposed Block Equivalence (BE), introducing the block-based strategy into a data-parallel algorithm. They make use of two additional data structures besides the output image: a block label map and a connectivity map, respectively to contain block labels and to record which blocks are connected together. The structure of the algorithm is the same as OLE, with the exception that it operates on blocks instead of single pixels. When convergence

is met, a final kernel is responsible for copying block labels into pixels of the output image.

Komura Equivalence (KE) [24] was released as an improvement over Label Equivalence. Anyway, it has more in common with the Union Find algorithm. Indeed, their structures are almost equivalent. The main difference is the initialization step, which starts building *Union-Find* trees while assigning the initial values to the output image. The original version of the algorithm employs 4-connectivity. An 8-connectivity variation has been presented in [2].

Finally, Distanceless Label Propagation (DLP) [12] tries to put together positive aspects of both UF and LE. The general structure is similar to that of UF, with the difference that the *Union* operation is performed between each pixel and the minimum value found in a 2×2 square. Moreover, the *Union* procedure is implemented in an original and recursive manner.

4 Adapting Tree-Based Algorithms to GPUs

Algorithm 2 Summary of algorithms kernels. I and L are input and output images. The pixel (or block) on which a thread works is denoted as x.

1: **kernel** Initialization(L)
2: $L[id_x] \leftarrow id_x$

3: **kernel** Merge(I, L)
4: `DecisionTree(Mask(x))`

5: **kernel** Compression(L)
6: $L[id_x] \leftarrow$ `Find`(L, id_x)

7: **kernel** FinalLabeling(L)
8: $label \leftarrow L[id_x]$
9: **for all** $a \in$ `Block`(x) **do**
10: **if** $I(a) = 1$ **then**
11: $L(a) \leftarrow label$
12: **else**
13: $L(a) \leftarrow 0$

We adapt SAUF, BBDT and DRAG to a parallel environment, thus producing CUDA based CCL algorithms, that we call C-SAUF, C-BBDT and C-DRAG.

A GPU algorithm consists of a sequence of kernels, *i.e.*, procedures run by multiple threads of execution at the same time. In order to transform the aforementioned sequential algorithms into parallel ones, the three steps of which they are composed (*first scan*, *flattening*, and *second scan*) must be translated into appropriate kernels. In each of those steps, a certain operation is repeated over every element of a sequence. Thus, a naive parallel version consists in a concurrent execution of the same operation over the whole sequence. Unfortunately, the *first scan* cannot be translated in such a simple way, because of its inherently sequential nature: when thread t_x runs, working on pixel x, every foreground pixel in the neighborhood mask must already have a label. To address this issue, we assign an initial label to each foreground pixel, equal to its raster index (id_x). This choice has two important consequences. First, we can observe that there is no need to store provisional labels in the output image L, because calculating them is trivial. So, until *second scan*, L can be used as the *Union-Find* structure, thus removing the need to allocate additional memory for P. In fact, in our parallel

Table 1. Kernel composition of the proposed CUDA algorithms.

	C-SAUF	C-BBDT	C-DRAG	
Initialization	✓	✓	✓	Creates starting *Union-Find* trees
Merge	✓	✓	✓	Merges trees of equivalent labels
Compression	✓	✓	✓	Flattens trees
FinalLabeling		✓	✓	Copies block labels into pixels

algorithms, $L \equiv P$. The second consequence is that the *first scan* loses the aim of assigning provisional labels, and it is only required to record equivalences. Its job is performed by two different kernels: *Initialization* and *Merge*.

The first one initializes the *Union-Find* array L. Of course, at the beginning, every label is the root of a distinct tree. Thus, in this kernel, thread t_x performs $L[id_x] \leftarrow id_x$.

The second kernel, instead, 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 of the scanning mask are foreground, nothing needs to be done. In all other cases, the current label needs to be merged with those of connected pixels, with the *Union* procedure. Moreover, the implementation of *Union* proposed in Algorithm 1 requires to introduce atomic operations to deal with the concurrent execution.

Then, it is easy to parallelize the *flattening* step: it translates into a kernel (*Compression*) in which 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.

The last step of 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 our parallel algorithms, because *Compression* kernel already solves label equivalences directly in the output image. In the case of C-SAUF, increasing foreground labels by one is the only remaining operation to perform, in order to ensure that connected components labels are positive numbers different from background. We avoid a specific kernel for this, shifting labels of foreground pixels by 1 since the beginning of the algorithm. This trick requires small changes to *Union-Find* functions. 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 in *FinalLabeling* kernel. Table 1 sums up the structure of the proposed algorithms, while Algorithm 2 provides a possible implementation of the described kernels.

5 Experimental Results

In order to produce a fair comparison, algorithms are tested and compared with state-of-the-art GPU implementations using the YACCLAB open-source benchmarking framework [9, 17]. Since this tool has been originally developed for se-

Table 2. Average run-time results in ms. The bold values represent the best performing CCL algorithm. Our proposals are identified with *.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
C-SAUF*	0.560	0.487	2.867	1.699	0.571	3.846	14.870
C-BBDT*	0.535	0.472	2.444	1.249	0.529	3.374	12.305
C-DRAG*	**0.526**	**0.460**	**2.423**	**1.220**	**0.496**	**3.322**	**12.012**
OLE [23]	1.111	1.031	5.572	2.996	1.174	8.152	35.245
BE [35]	1.401	1.056	4.714	2.849	1.053	6.120	20.314
UF [28]	0.593	0.527	3.243	2.062	0.656	4.332	17.333
DLP [12]	0.657	0.484	3.323	1.719	0.597	5.031	18.182
KE [2]	0.565	0.478	2.893	1.644	0.523	4.007	15.445

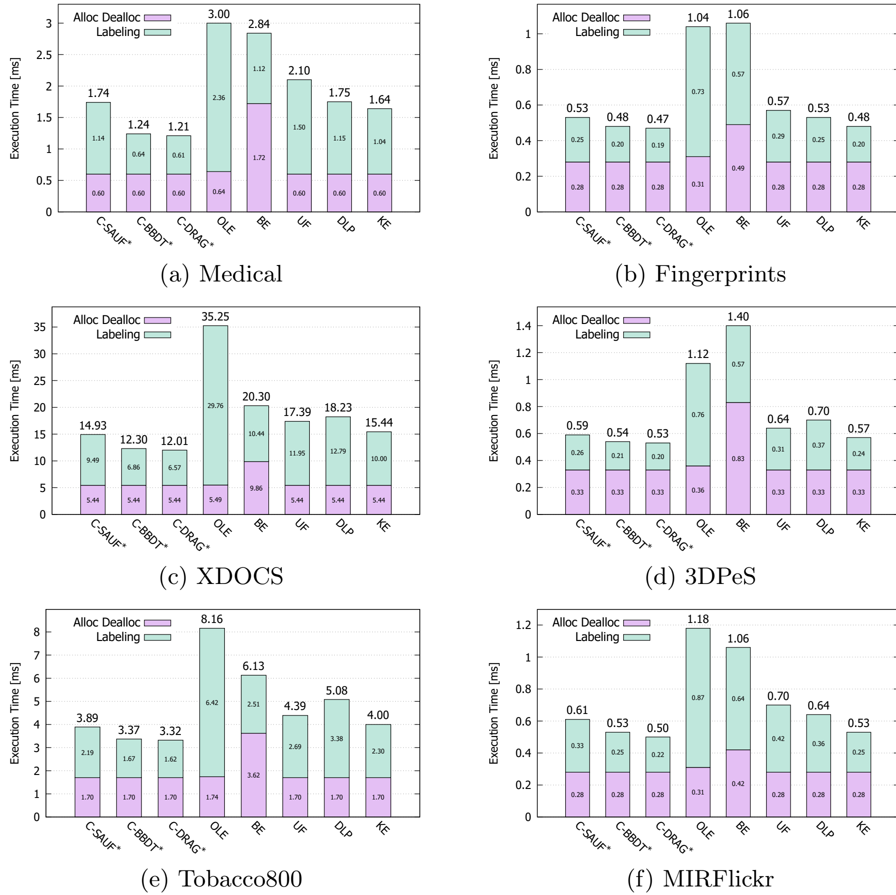
quential algorithms, we have enriched its capabilities to run also GPU-based CCL algorithms.

Experiments are performed on a Windows 10 desktop computer with an Intel Core i7-4770 (4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), 16 GB of RAM, and a Quadro K2200 NVIDIA GPU (640 CUDA cores and 4 GB of memory). Algorithms have been compiled for x64 architectures using Visual Studio 2017 (MSVC 19.13.26730) and CUDA 10 (NVCC V10.0.130) with optimizations enabled.

With the purpose of stressing algorithms behaviours, thus highlighting their strengths and weaknesses, we perform three different kind of tests on many real case and synthetically generated datasets provided by YACCLAB. Selected datasets cover most of the fields on which CCL is applied and are described in the following.

Medical [15] is a collection of 343 binary histological images with an average amount of 484 components to label. *Hamlet* is a scanned version of the Hamlet provided by the Gutenberg Project [32]. The dataset is composed of 104 images with an average amount of 1 447 components to label. *Fingerprints* [26] contains 960 images taken from fingerprint verification competitions (*FCV* 2000, *FCV* 2002 and *FCV* 20040) and binarized using an adaptive threshold. *XDOCS*, is a set of 1 677 high-resolution historical document images retrieved from the large number of civil registries that are available since the constitution of the Italian state [4, 6, 7]. Images have an average amount of 15 282 components to analyze. *3DPeS* [3] contains surveillance images processed with background subtraction and Otsu thresholding. *MIRflickr* is a set of images containing the Otsu binarized version of the MIRFLICKR-25000 [22] dataset. It is composed of 25 000 standard resolution images taken from Flickr, with an average amount of 492 connected components. *Tobacco800* [1, 25, 33] counts 1 290 document images collected and scanned using a wide variety of equipment over time. The images sizes vary from 1200×1600 up to 2500×3200.

As shown in Table 2, KE is confirmed to be the state-of-the-art GPU-specific algorithm, when taking into account the average run-time on all datasets. It is interesting to note that even a straightforward implementation of SAUF (C-SAUF) is able in many cases to outperform it. The two more complex tree-based



(a) Medical (b) Fingerprints

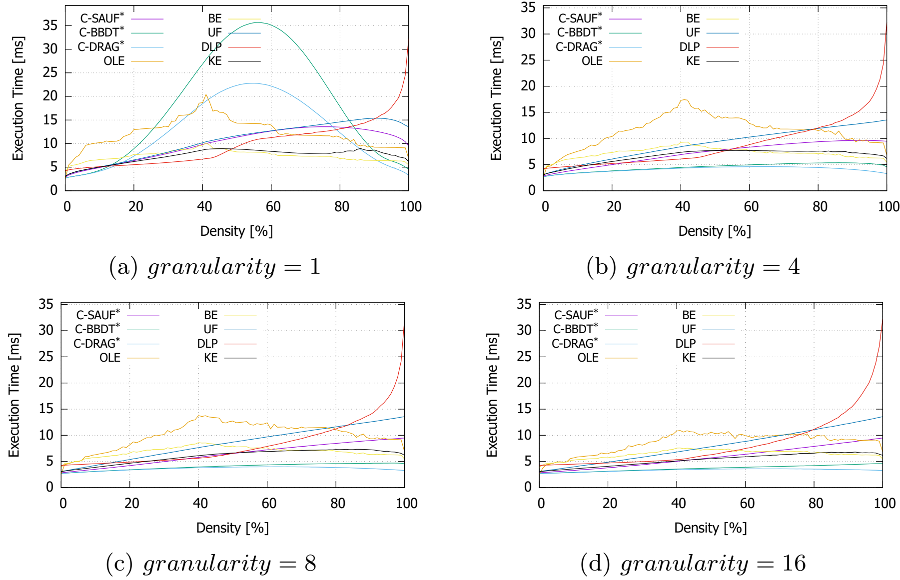
(c) XDOCS (d) 3DPeS

(e) Tobacco800 (f) MIRFlickr

Fig. 2. Average run-time results with steps in ms. Lower is better.

algorithms (C-BBDT and C-DRAG) significantly reduce the computational requirements, with C-DRAG being always the best.

To better appreciate this results, it is useful to split the time required by memory allocation and the computation. Fig. 2 shows that in some cases more than 50% of the time is dedicated to memory allocation, especially on small images. BE clearly suffers from the additional data structures. When moving to larger images (*e.g.* XDOCS), the reduced time allowed by C-BBDT and C-DRAG is evident. How is this possible? Irregular patterns of execution are supposed to slow down the thread scheduling, so a decision tree is the worst thing that could happen to GPUs. This is not a myth, but we need to understand how often different branches are taken in the execution. So, the third test (Fig. 3) is run on a set of synthetic images generated by randomly setting a certain percentage (*density*) of pixel blocks to foreground. The minimum size of foreground blocks is called *granularity*. Resolution is 2048×2048, density ranges from 0% to 100%



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

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

Fig. 3. Granularity results in ms on images at various densities. Lower is better.

with a step of 1%, and granularity $g \in [1, 16]$ have been considered. Ten images have been generated for every couple of density-granularity values, for a total of 16 160 images, and charts report the average time per couple. When granularity is 1 (Fig. 3a), it is possible to observe that both C-BBDT and C-DRAG computational time explodes around density of 50%, in accordance with our expectation. The only cases in which those algorithms outperform GPU-specific ones is when density is below 10% or over 90%. Indeed, at low/high densities the decision is taken quickly by the first levels of the tree structures, saving a lot of memory accesses and without breaking the thread execution flow. Since images on which CCL is applied are usually in that range of densities, this explains the previously observed behavior. Moreover, when granularity grows (Fig. 3b-Fig. 3d), small portions of the image have irregular patterns requiring to explore deeper tree levels, while the vast majority tends to be all white or all black, again maximizing the tree performance.

6 Conclusions

In this paper we have addressed the problem of connected components labeling on GPU. We have focused our analysis on three 8-connectivity sequential algorithms relying on decision trees and we have adapted them to GPU programming paradigm, without altering their substance. Hence, we have compared these proposals against GPU state-of-the-art algorithms on real case datasets

and the experimental results show their surprisingly good performance. An explanation of their effectiveness has been provided thanks to a set of additional tests on synthetic images. The C-DRAG algorithm always outperforms the other CUDA-designed proposals, highlighting the feasibility of decision trees on GPU. The source code of described algorithms is available in [31], allowing anyone to reproduce and verify our claims.

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 (2018)
3. 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)
4. Bolelli, F.: Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text. In: Italian Research Conference on Digital Libraries. pp. 45–55. Springer (2017)
5. Bolelli, F., Baraldi, L., Cancilla, M., Grana, C.: Connected Components Labeling on DRAGs. In: International Conference on Pattern Recognition (2018)
6. 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 (2017)
7. Bolelli, F., Borghi, G., Grana, C.: XDOCS: An Application to Index Historical Documents. In: Italian Research Conference on Digital Libraries (2018)
8. Bolelli, F., Cancilla, M., Baraldi, L., Grana, C.: Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes. In: 24rd International Conference on Pattern Recognition Workshops (2018)
9. 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)
10. Bolelli, F., Cancilla, M., Grana, C.: Two More Strategies to Speed Up Connected Components Labeling Algorithms. In: International Conference on Image Analysis and Processing (2017)
11. 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 (Jun 2012)
12. Cabaret, L., Lacassagne, L., Etiemble, D.: Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs. In: International Conference on Image Processing Theory, Tools and Applications. IPTA (2017)
13. 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 (2003)
14. Dillencourt, M.B., Samet, H., Tamminen, M.: A General Approach to Connected-Component Labeling for Arbitrary Image Representations. Journal of the ACM **39**(2), 253–280 (1992)

15. 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) (2014)
16. Grana, C., Baraldi, L., Bolelli, F.: Optimized Connected Components Labeling with Pixel Prediction. In: Advanced Concepts for Intelligent Vision Systems (2016)
17. Grana, C., Bolelli, F., Baraldi, L., Vezzani, R.: YACCLAB - Yet Another Connected Components Labeling Benchmark. In: 23rd International Conference on Pattern Recognition. ICPR (2016)
18. Grana, C., Borghesani, D., Cucchiara, R.: Fast Block Based Connected Components Labeling. In: 2009 16th IEEE International Conference on Image Processing (ICIP). pp. 4061–4064. IEEE (2009)
19. 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)
20. Grana, C., Montangero, M., Borghesani, D.: Optimal decision trees for local image processing algorithms. Pattern Recognition Letters **33**(16), 2302–2310 (2012)
21. He, L., Zhao, X., Chao, Y., Suzuki, K.: Configuration-Transition-Based Connected-Component Labeling. IEEE Transactions on Image Processing **23**(2) (2014)
22. 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)
23. 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)
24. 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)
25. Lewis, D., Agam, G., Argamon, S., Frieder, O., Grossman, D., Heard, J.: Building a test collection for complex document information processing. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 665–666. ACM (2006)
26. Maltoni, D., Maio, D., Jain, A., Prabhakar, S.: Handbook of fingerprint recognition. Springer Science & Business Media (2009)
27. Nickolls, J., Dally, W.J.: The GPU Computing Era. IEEE Micro **30**(2) (2010)
28. Oliveira, V.M., Lotufo, R.A.: A study on connected components labeling algorithms using GPUs. In: SIBGRAPI. vol. 3, p. 4 (2010)
29. 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). IEEE (2018)
30. Pollastri, F., Bolelli, F., Paredes, R., Grana, C.: Augmenting Data with GANs to Segment Melanoma Skin Lesions. In: Multimedia Tools and Applications Journal. MTAP, Springer (2019)
31. Source Code, `https://github.com/prittt/YACCLAB`, accessed on 2019-03-30
32. The Hamlet Dataset, `http://www.gutenberg.org`, accessed on 2019-03-30
33. The Legacy Tobacco Document Library (LTDL). University of California (2007)
34. Wu, K., Otoo, E., Suzuki, K.: Two Strategies to Speed up Connected Component Labeling Algorithms. Tech. Rep. LBNL-59102, Lawrence Berkeley National Laboratory (2005)
35. 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)