Optimized Block-Based Algorithms to Label Connected Components on GPUs

Stefano Allegretti, Federico Bolelli, *Student Member, IEEE*, and Costantino Grana, *Member, IEEE*

Abstract—Connected Components Labeling (CCL) is a crucial step of several image processing and computer vision pipelines. Many efficient sequential strategies exist, among which one of the most effective is the use of a block-based mask to drastically cut the number of memory accesses. In the last decade, aided by the fast development of Graphics Processing Units (GPUs), a lot of data parallel CCL algorithms have been proposed along with sequential ones. Applications that entirely run in GPU can benefit from parallel implementations of CCL that allow to avoid expensive memory transfers between host and device. In this paper, two new eight-connectivity CCL algorithms are proposed, namely Block-based Union Find (BUF) and Block-based Komura Equivalence (BKE). These algorithms optimize existing GPU solutions introducing a block-based approach. Extensions for three-dimensional datasets are also discussed. In order to produce a fair comparison with previously proposed alternatives, YACCLAB, a public CCL benchmarking framework, has been extended and made suitable for evaluating also GPU algorithms. Moreover, three-dimensional datasets have been added to its collection. Experimental results on real cases and synthetically generated datasets demonstrate the superiority of the new proposals with respect to state-of-the-art, both on 2D and 3D scenarios.

Index Terms—Parallel Processing, Connected Components Labeling, GPU, CUDA.

1 Introduction

Connected Components Labeling (CCL) is a fundamental image processing algorithm that extracts connected components (CC) from an input binary image. During the labeling procedure, the binary input is transformed into a symbolic image where all pixels belonging to a CC (object) are given the same label, typically an integer number.

Originally introduced by Rosenfeld and Pfaltz in 1966 [1], CCL has been in use for more than 50 years in multiple image processing and computer vision pipelines, including Object Tracking [2], Video Surveillance [3], Image Segmentation [4], [5], [6], Medical Imaging Applications [7], [8], [9], [10], Document Restoration [11], [12], Graph Analysis [13], [14], and Environmental Applications [15].

Since CCL has an exact solution, the main difference between algorithms is the execution time. Moreover, given that labeling represents the base step of many real-time applications, it is required to be as fast as possible. Therefore, research in the field moved towards the optimization of the performance of these algorithms in term of execution speed.

In the last years, the fast advance of Graphic Processing Units (GPUs) encouraged the development of algorithms specifically designed to work in a data parallel environment. So, along with sequential solutions [16], [17], [18], [19], many novel algorithms exploiting the parallelism of both CPUs and GPUs have been proposed [20], [21], [22], [23], [24].

Unfortunately, CCL is not as easy to parallelize as many other image processing tasks. Being it essentially a graph theory problem, algorithms need to perform graph traversal at a certain degree, which is an inherently sequential operation. For this reason, CPU and GPU algorithms usually have comparable performance [25]. However, the existence of efficient data parallel algorithms is valuable for applications that entirely run on GPU, allowing to remove the need for data transfers between host and device memory.

In this article, we propose two new 8-connectivity GPU-based connected components labeling methods that improve previously proposed algorithms by taking advantage of the 2×2 block-based approach originally presented in [26] for sequential algorithms. This allows to drastically reduce the amount of memory accesses, thus improving overall performance.

In previous work [22], the 2×2 blocks were applied to the iterative GPU algorithm presented in [27] and improved in [28]. However, the chosen algorithm was rather slow compared to other approaches [20], [29]. Moreover, the benefit introduced in [28] was partially lessened by an increased allocation time, caused by the need for additional data structures to record information about blocks connectivity and blocks labels. We managed to adapt the block-based approach to the direct and more efficient algorithms proposed in [20] and [29]. Moreover, we removed the need for additional data structures, in order to minimize the amount of time spent for allocating and deallocating device memory. The results are two extremely fast solutions, able to improve state-of-the-art over both real case and synthetically generated datasets. Variations of our proposals are also described, meant to perform CCL on three-dimensional binary volumes.

Our contribution also includes the extension of YACCLAB [30], [31] (Yet Another Connected Components LAbeling Benchmark), a public benchmark to evaluate the performance of sequential CCL algorithms. This extension introduces support for GPU and 3D CCL algorithms, alongside new 3D tests and datasets. The source code of our proposals, as well as the extended benchmark, is available online [32].

The rest of this paper is organized as follows. Section 2 defines the basic concepts and notation used throughout the paper. In Section 3, the main contributions on parallel CCL are resumed, comparing their properties and performance. Section 4 analyzes relevant techniques that constitute the basis of our work, then Section 5 details the proposed algorithms. Section 6 illustrates the extension of YACCLAB. Section 7 demonstrates the effectiveness of our approach in comparison with other state-of-the-art methods, providing an exhaustive evaluation. Finally, conclusions remarks are given in Section 8.

2 Connected Components Labeling

This section defines the problem of connected components labeling, introducing the basic notations used throughout the paper. Moreover, a common paradigm exploited by both sequential and parallel algorithms is described: the *disjoint-set* data structure, also referred to as *union-find* or *merge-find*.

We will call I_2 an image defined over a two dimensional rectangular lattice $\mathcal{L}_2$, and $I_2(p)$ the value of pixel $p \in \mathcal{L}_2$, with $p = (p_x, p_y)$. Two different kinds of neighborhood can be defined (*4-neighborhood* and *8-neighborhood*):

$$\mathcal{N}_4(p) = \{q \in \mathcal{L}_2 \mid |p_x - q_x| + |p_y - q_y| \leq 1\}$$
$$\mathcal{N}_8(p) = \{q \in \mathcal{L}_2 \mid \max(|p_x - q_x|, |p_y - q_y|) \leq 1\}$$

Two pixels, p and q, are said to be *4-neighbors* if $q \in \mathcal{N}_4(p)$, that implies $p \in \mathcal{N}_4(q)$, and are said to be *8-neighbors* if $q \in \mathcal{N}_8(p)$, that implies $p \in \mathcal{N}_8(q)$. From a visual perspective, if we imagine pixels as square-sized, p and q are *4-neighbors* if they share an edge, and they are *8-neighbors* if they share an edge *or* a vertex.

Similar concepts can be defined for three-dimensional images, also known as *volumes*. Given a volume I_3, defined over a three dimensional lattice $\mathcal{L}_3$, we define two kinds of neighborhood of a voxel (volume pixel) $v \in \mathcal{L}_3$, with $v = (v_x, v_y, v_z)$:

$$\mathcal{N}_6(v) = \{v \in \mathcal{L}_3 \mid |v_x - w_x| + |v_y - w_y| + |v_z - w_z| \leq 1\}$$
$$\mathcal{N}_{26}(v) = \{v \in \mathcal{L}_3 \mid \max(|v_x - w_x|, |v_y - w_y|, |v_z - w_z|) \leq 1\}$$

This time we can visualize voxels as cubes. So, *6-neighbors* voxels share a side, while *26-neighbors* ones share a side *or* an edge *or* a vertex.

From now on, we will use the word *image* to refer to 3D volumes also, and generic symbols $\mathcal{L}$ and I in definitions suitable for both dimensionalities. 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 ones:

$$\mathcal{F} = \{p \in \mathcal{L} \mid I(p) = 1\} \quad (1)$$
$$\mathcal{B} = \{p \in \mathcal{L} \mid I(p) = 0\} \quad (2)$$

The aim of connected components labeling is to identify disjoint objects composed of foreground pixels. So, for a chosen neighborhood definition (simply called $\mathcal{N}$), and given two foreground pixels $p, q \in \mathcal{F}$, the relation of *connectivity* $\diamond$ can be defined as:

$$p \diamond q \Leftrightarrow \exists\{s_i{\in}\mathcal{F} \mid s_1{=}p, s_{n+1}{=}q, s_{i+1}{\in}\mathcal{N}(s_i), i{=}1,\dots,n\} \quad (3)$$

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 pixels exist, from p to q. Note that, with this formalism, background pixels are excluded from the concept of connectivity. Given that pixel connectivity satisfies the properties of *reflexivity*, *symmetry* and *transitivity*, $\diamond$ is an equivalence relation. Therefore, the equivalence class of a pixel p is denoted as $[p]$ and is defined as the set:

$$[p] = \{q \in \mathcal{F} \mid p \diamond q\} \quad (4)$$

In this case, equivalence classes based on $\diamond$ relationship are called *connected components*. Every two connected components $[p]$ and $[q]$ are either equal or disjoint. Therefore, the set of all connected components is a partition of $\mathcal{F}$.

Connected components labeling algorithms aim at assigning a different label, typically an integer number, to every connected component. When applied to an image I defined over a lattice $\mathcal{L}$, the output of such an algorithm is a symbolic image L where, for every $p \in \mathcal{F}$, $L(p)$ is the label of the connected component that p belongs to ($[p]$), and for every $q \in \mathcal{B}$, $L(q) = 0$.

Depending on the chosen neighborhood definition, *n-neighborhood*, a connected components labeling algorithm is said to employ *n-connectivity*. Many computer vision tasks require 8-connectivity for 2D images, and 26-connectivity for 3D volumes. In fact, according to the *Law of Closure* of Gestalt psychology, our senses perceive an object as a whole even if it is composed of loosely connected parts. [26].

2.1 The Union-Find data structure

Since two connected components are always disjoint, CCL can be seen as the partitioning of the lattice $\mathcal{L}$. A possible technique for performing such a partitioning consists of building initial sets of connected pixels, and then joining together sets that are part of the same connected component. This kind of problem can take advantage of the *union-find* data structure, that was firstly applied to CCL by Dillencourt *et al.* in [33].

The *union-find* data structure keeps track of $\mathcal{P}$, a partition of a set $\mathcal{S}$, and provides two basic operations on the elements of $\mathcal{S}$:

- `Find`(a), with $a \in \mathcal{S}$: returns the identifier of the subset that contains a.
- `Union`(a, b), with $a, b \in \mathcal{S}$: joins the subsets containing a and b.

$\mathcal{P}$ is usually represented as a graph, in particular as a forest of directed rooted trees with orientation towards the root (*anti-arborescence*). Each element of $\mathcal{S}$, a, corresponds to a node and has a unique identifier id_a. An ordering exists between identifiers. A tree inside the forest identifies a subset belonging to $\mathcal{P}$.

A directed edge leading from b to a, with $id_a < id_b$, states that b and a belong to the same subset. According to the definition of directed rooted tree, we will say that a is the *father* of b. Of course, each element can have at most one out-going edge. The id of a tree (subset) corresponds to that of its root node.

Representing $\mathcal{P}$ as a forest of trees is especially useful because a forest can be efficiently stored in memory using an array. Each index of this array is the id of a node, and the value stored at that index is the id of its father node, or the

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

```
function FIND($L$, $a$)
    while $L[a] \neq a$ do
        $a \leftarrow L[a]$
    return $a$

procedure COMPRESS($L$, $a$)
    $L[a] \leftarrow \texttt{Find}(L, a)$

procedure INLINECOMPRESS($L$, $a$)
    $id \leftarrow a$
    while $L[a] \neq a$ do
        $a \leftarrow L[a]$
        $L[id] \leftarrow a$
    return $a$

procedure UNIONNAIVE($L$, $a$, $b$)
    $a \leftarrow \texttt{Find}(L, a)$
    $b \leftarrow \texttt{Find}(L, b)$
    if $a < b$ then
        $L[b] \leftarrow a$
    else if $b < a$ then
        $L[a] \leftarrow b$

procedure UNION($L$, $a$, $b$)
    $done \leftarrow false$
    while $done = false$ do
        $a \leftarrow \texttt{Find}(L, a)$
        $b \leftarrow \texttt{Find}(L, b)$
        if $a < b$ then
            $old \leftarrow \texttt{atomicMin}(\&L[b], a)$
            $done \leftarrow (old = b)$
            $b \leftarrow old$
        else if $b < a$ then
            $old \leftarrow \texttt{atomicMin}(\&L[a], b)$
            $done \leftarrow (old = a)$
            $a \leftarrow old$
        else
            $done \leftarrow true$
```

index itself in the case of roots [34]. When applying *union-find* to GPU-based CCL algorithms, a common strategy is to identify pixels as nodes of the graph. In this case, the node id is the pixel raster index and each tree represents a connected component. An additional tree is required for background pixels. In such scenario the set $\mathcal{S}$ is the lattice $\mathcal{L}$. Thus, the array-based representation requires the same size as the output labels image: both require as many elements as the number of pixels in $\mathcal{L}$. Given that memory allocation on GPUs is considerably time consuming, since it requires an expensive operating system call to the driver [35], many GPU CCL algorithms make the *union-find* array coincide with the output labels image L [20], [29], [36], [37], [38], [39].

A possible implementation of the *union-find* functions is reported in Alg. 1. Implementations are made to fit CUDA data-parallel environment, where many threads can run the same function simultaneously, though on different input data. A complete description is reported in the following:

- `Find`(L, a) consists of traversing the tree to which a belongs, starting from a and leading to the root node, whose index is the tree identifier.
- `Compress`(L, a) is a procedure that links a directly to the root of its tree. It is used to flatten *union-find* trees. When every tree in the *union-find* array exactly matches a connected component, the `Compress` procedure can be performed on every node/pixel to produce the final output image.
- `InlineCompress`(L, a) is a variation of `Compress`, optimized for a data parallel environment. This procedure updates the father of a at every step of the tree traversal. This way, possible concurrent threads that read a can use the updated value. This approach is called *InlineCompression (IC)* and was firstly introduced in [36].
- `UnionNaive`(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. This solution does not take into account the possibility that two threads reach and modify the same root starting from different input nodes, possibly causing race hazards.
- `Union`(L, a, b), firstly introduced by Oliveira *et al.* [29], solves the problems of `UnionNaive`, introducing CUDA atomic operations to make a thread aware of possible update losses caused by concurrent execution.

Fig. 1 provides an example representation of $\mathcal{P}$ as a forest of trees stored in memory as an array.

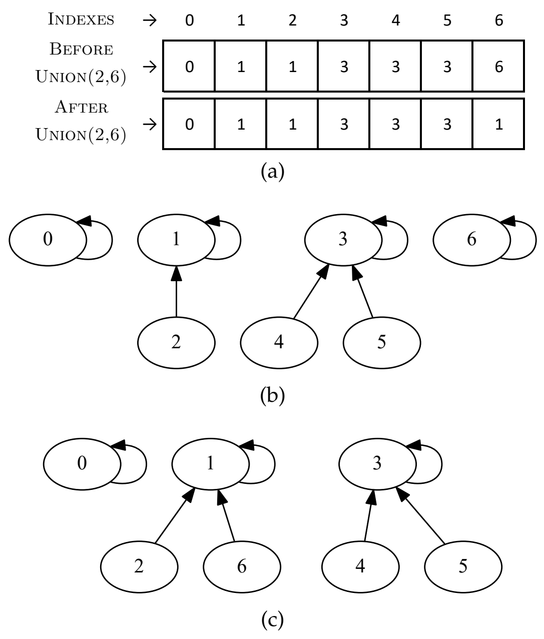

INDEXES → 0 1 2 3 4 5 6
BEFORE UNION(2,6) → 0 1 1 3 3 3 6
AFTER UNION(2,6) → 0 1 1 3 3 3 1
(a)
0
1
2
3
4
5
6
(b)
(c)

Fig. 1. Visualization of $\mathcal{P}$ represented as a forest of trees and stored in memory as an array called L. (a) is L before and after the execution of the `Union` operation between identifiers 2 and 6. (b) and (c) are the corresponding forests respectively before and after the `Union`. In this example, the `Find` function, called on 2 and 6 during the `Union`, returns 1 (the root of the tree 2 belong to) and 6 (since 6 is the root of the tree). The `Union` procedure replaces the identifier at index 6 of the L array with 1, thus storing the equivalence between the two classes.

3 Related Work

Parallel connected components algorithms can be divided into two disjoint sets, depending on the number of kernel executions. Iterative algorithms repeat one or more kernels until no more changes in the data structures that they modify are detected. Such a situation is called *convergence*, and the number of kernel calls needed to reach it depends on the configuration of the input data. Conversely, direct algorithms are characterized by a fixed number of kernel executions. Most of the published works can be described by means of the *union-find* functions, even if the authors did not explicitly refer to them.

3.1 Iterative algorithms

Label Equivalence (LE), proposed by Hawick *et al.* [27] in 2010, is an iterative algorithm that records *union-find* trees using an auxiliary data structure. In the first step, both the output image and *union-find* data are initialized with sequential values. The algorithm then consists of three kernels that are repeated in sequence until convergence. They aim at propagating the minimum label through each connected component, exploiting a procedure similar to `Compress` to flatten *union-find* trees at every step.

In 2011, Kalentev *et al.* [28] proposed an Optimization of Label Equivalence (OLE), noticing that the need for a separate data structure to store label equivalences could be removed by directly using the output image.

Zavalishin *et al.* [22] were the first to apply the block-based strategy proposed by Grana in [26] to a data-parallel CCL algorithm. This approach is based on the observation that, when dealing with 8-connectivity, foreground pixels in a 2×2 block always share the same label. The proposed strategy, which is a variation of Label Equivalence named Block Equivalence (BE), makes use of two additional data structures besides the output image: a block label map and a connectivity map, respectively to contain blocks labels and to record which blocks are connected together. At the beginning of the algorithm, the image is divided into blocks, a label is assigned to each of them, and the necessary information about blocks connectivity is calculated and stored into the connectivity map for future use. 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.

3.2 Direct algorithms

The first GPU CCL algorithm that makes use of *union-find* was proposed by Oliveira *et al.* in 2010 [29]. We will refer to it as UF. The output image is initialized with sequential values as usual. Then, *union-find* primitives are used to join together the trees of neighbor pixels. Finally, a flattening of trees, performed with the `Compress` procedure, ends the task. The algorithm is first performed on rectangular tiles, and then large connected components are merged in a subsequent step.

In 2015, Yonehara and Aizawa proposed Line Based Union Find (LBUF) [36], a variation of UF that employs single lines as tiles in the first step. This choice reduces the neighborhood of a pixel to a subset containing only the two neighbors which belong to the same row, allowing to simplify the logic of the local step. The remaining of the algorithm is left unchanged, except for the use of *InlineCompression* to speed up the flattening of trees.

Komura Equivalence (KE) [20] was created in 2015, as an improvement over Label Equivalence. Anyway, it has more in common with Union Find. Indeed, its structure is very similar to that of UF, but for a different initialization, which starts building *union-find* trees while assigning the initial values to the output image. An improved version of KE, that gets rid of many redundant `Union` operations, has been proposed by Playne *et al.* [25]. The original algorithm and the aforementioned optimization employ 4-connectivity. A 8-connectivity variation has been presented in [38].

Distanceless Label Propagation (DLP) [37] is a proposal that tries to put together positive aspects of both UF and LE. The general structure is similar to that of UF, with the difference that a `Union` is performed between each pixel and the minimum value found in a 2×2 square. The `Union` procedure itself is implemented in an original and recursive manner.

4 Preliminaries

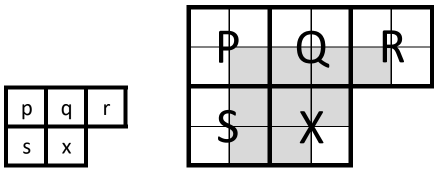
In order to better explain our proposals, we need to detail some of the algorithms introduced in the previous Section.

4.1 The Union Find algorithm

As said, UF is the first algorithm exploiting *union-find* on GPU. The original version of the algorithm employs 4-connectivity, but it can be easily extended to 8-connectivity [38]. The algorithm is based on three kernels: *Initialization*, *Merge* and *Compression*. An example of execution is depicted in Fig. 3. Each kernel runs on a number of threads equal to the image size, and each thread is assigned a pixel, which we will refer to as x. The *union-find* trees are coded in the output label image L.

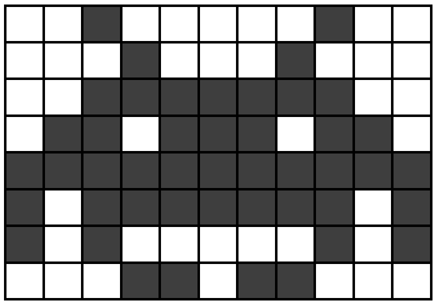
During *Initialization*, every foreground pixel p in the output image L is initialized with its id, which corresponds to its raster index increased by 1. More specifically, thread t_p performs $L[p] \leftarrow id_p$. From the *union-find* point of view, this procedure corresponds to the creation of a separate tree for every pixel. Background pixels are set to 0 instead.

During *Merge*, each thread working on a foreground pixel checks its neighbors, and for every foreground neighbor performs a `Union` with x. Since `Union` is a symmetric

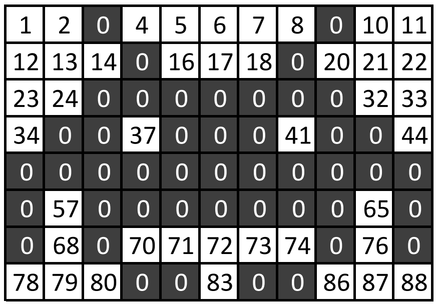


(a) Rosenfeld (b) Block-based

Fig. 2. Neighborhood masks used by the two-dimensional algorithms described in the paper. UF and KE employ mask (a), where the central pixel is x. The block-based mask (b) is used by BUF and BKE instead. Central block is X, and the connectivity between it and its neighbors depends on the value of grey pixels.

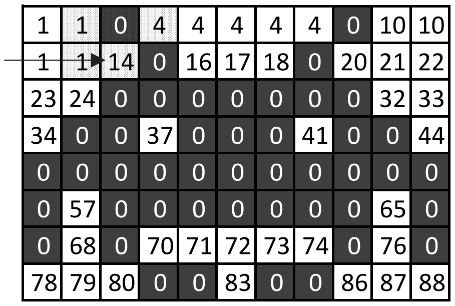


(a) Binary Input



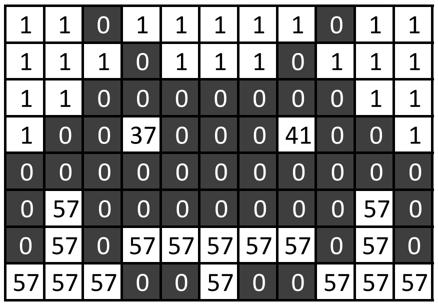
1	2	0	4	5	6	7	8	0	10	11
12	13	14	0	16	17	18	0	20	21	22
23	24	0	0	0	0	0	0	0	32	33
34	0	0	37	0	0	0	41	0	0	44
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	65	0
0	68	0	70	71	72	73	74	0	76	0
78	79	80	0	0	83	0	0	86	87	88

(b) Output Initialization



1	1	0	4	4	4	4	4	0	10	10
1	1	14	0	16	17	18	0	20	21	22
23	24	0	0	0	0	0	0	0	32	33
34	0	0	37	0	0	0	41	0	0	44
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	65	0
0	68	0	70	71	72	73	74	0	76	0
78	79	80	0	0	83	0	0	86	87	88

(c) Provisional Result



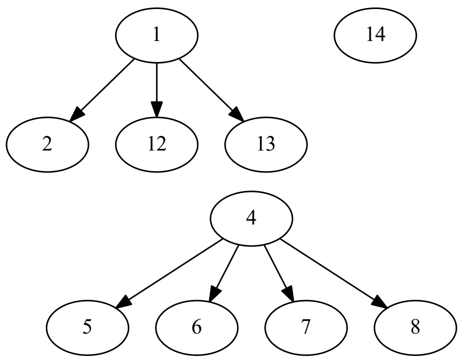
1	1	0	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	0	1	1	1
1	1	0	0	0	0	0	0	0	1	1
1	0	0	37	0	0	0	41	0	0	1
0	0	0	0	0	0	0	0	0	0	0
0	57	0	0	0	0	0	0	0	57	0
0	57	0	57	57	57	57	57	0	57	0
57	57	57	0	0	57	0	0	57	57	57

(d) Output Labels

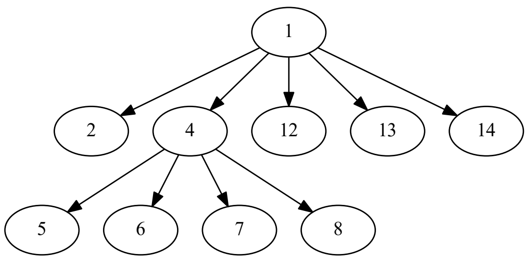
Fig. 3. Example of Union Find execution on a negative Space Invaders character. The goal is to label differently the white areas. (b) is the expected labels image after *Initialization*. (c) is a provisional result of *Merge* kernel, under the exemplifying assumption that threads run in raster scan order, and the execution reached thread 14. The configuration of the *union-find* data structure before and after the execution of thread 14 is shown in Fig. 4. (d) is the labels image after the execution of the last kernel, *Compression*.

operation, there is no need to check the entire neighborhood. Therefore, only neighbors with a smaller raster index than x are considered. These neighbors are identified by the mask depicted in Fig. 2a.

Fig. 4 shows the effects of *Merge* on the configuration of the union-find data structure, before and after the execution



(a) 13



(b) 14

Fig. 4. Change in the configuration of *union-find* trees operated by the thread working on pixel 14, and starting from the hypothetical provisional result of Fig. 3c. The thread checks its neighborhood mask in increasing raster index order. So, firstly, a `Union` is performed between the single-node tree 14 and the tree that has 1 as its root. Then, a subsequent `Union` between pixel 14 and pixel 4 causes the linking of the subtree starting at node 4 to the tree containing 14, rooted in 1.

of thread 14, and starting from the provisional result of Fig. 3c. In this example, the thread operating on pixel 14 performs a `Union` between the tree rooted in 14 and the trees rooted in 1 and 4.

After *Merge*, every connection between pixels in the image is reflected in the *union-find* structure, in the sense that every separate tree represents a connected component.

Finally, *Compression* kernel performs the compression of trees: every threads t_p runs $\texttt{Compress}(L, p)$. This operation makes sure that every pixel is assigned a label that corresponds to the raster index of the root of its tree. Thus, every pixel in the same CC ends up sharing the same label, and the labeling task is completed.

The aforementioned basic structure of the algorithm is enhanced with the addition of another common parallelization strategy, known as Tile Merging (TM). This means that the entire algorithm is first performed on rectangular blocks the image is divided into: this preliminary phase is called *Local Merge*. Then, a *Merge* kernel is performed on border pixels only, and a final *Compression* is run over the whole image.

4.2 The Komura Equivalence algorithm

The Komura Equivalence algorithm [20] is another GPU algorithm that can be seen as a variation of UF, which involves a more complex initialization in order to remove the need for one `Union` per pixel later. It consists of four steps: *Initialization*, *Compression*, *Reduction*, and *Compression* again.

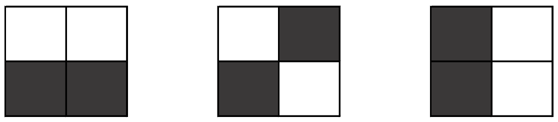


Fig. 5. Examples of foreground pixels in a $2{\times}2$ block.

The main difference between KE an UF lies in the first kernel. Differently from UF, KE *Initialization* does not create single-node trees. Instead, every thread checks the neighborhood of x in increasing raster index order, and the smallest foreground neighbor is set as the father node of x. Thus, this first phase aims at creating non single-node trees, that are flattened by the subsequent *Compression* kernel. UF and KE *Compression* kernels are exactly the same.

The *Reduction* kernel is a variation of *Merge*, that only performs a `Union` between x and foreground neighbors which were not chosen during *Initialization*. Analogously to UF, a final *Compression* is required for the flattening of the forest trees. Same as UF, the original KE is a 4-connectivity algorithm. It can as well be modified to deal with 8-connectivity, adding the supplementary neighbors in both *Initialization* and *Reduction* kernels [38].

4.3 The Block-Based approach

Grana *et al.* noticed in [26] that, in the case of a two-dimensional image (I_2) and 8-connectivity, all foreground pixels within $2{\times}2$ blocks always share the same label (Fig. 5). This observation can be extended to volumes (I_3), where 26-connectivity implies that only one label can be assigned to foreground voxels of a $2{\times}2{\times}2$ block.

Consequently, when 8-connectivity (26-connectivity for volumes) is used, CCL can be applied to blocks, and block labels can be assigned to single pixels at the end of the algorithm. Such an approach usually has some advantages in terms of execution time, depending on the chosen algorithm. Considering UF, the use of blocks would divide the number of initial trees by 4 (8 for volumes), thus requiring considerably less `Union` calls to achieve the final configuration. Moreover, the average depth of trees is reduced as well, speeding up `Find`. A similar reasoning can be applied to KE.

However, when blocks are considered in place of pixels, the definition of *connectivity* must change accordingly. We say that two blocks P, Q are *connected* if one pixel of P is connected to one pixel of Q:

$$P \diamond Q \Leftrightarrow \exists p \in P, q \in Q \mid \; p \diamond q \tag{5}$$

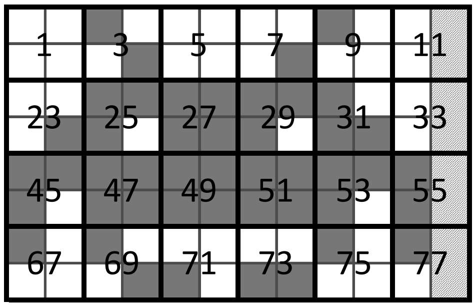
As a consequence, finding the connected neighbors of a block is more difficult than finding those of a single pixel, because the number of pixels to be checked is higher. Moreover, since the same internal pixel of a block can be responsible for connecting it to more than one neighbor blocks, a method that avoids multiple reads of the same pixel is valuable. Zavalishin *et al.*, who first applied blocks to GPU CCL, make use of pixel connectivity maps to find neighbor blocks while reading pixels only once [22].

Algorithm 2 Block-based Union Find kernels. I and L are input and output images, linearly stored in memory row-by-row. A padding can be added at the end of rows for alignment purpose, so *step* stores the effective length of rows in memory. Checks on image borders are not shown.

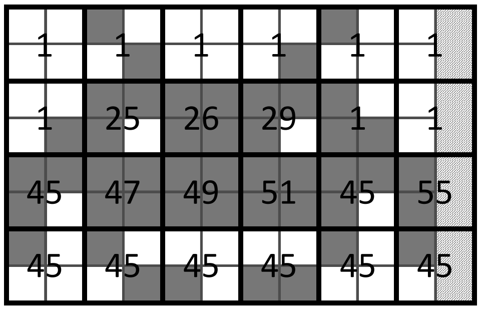
```
kernel INITIALIZATION($L$, $step_L$)
    $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
    $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
    $x_L \leftarrow r \times step_L + c$
    $L[x_L] \leftarrow x_L$

kernel MERGE($I$, $step_I$, $L$, $step_L$)
    $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
    $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
    $x_I \leftarrow r \times step_I + c$
    $x_L \leftarrow r \times step_L + c$
    $\mathcal{BS} \leftarrow 0$
    if $I[x_I] = 1$              then $\mathcal{BS}$ |=  $0x777$
    if $I[x_I + 1] = 1$          then $\mathcal{BS}$ |= $(0x777 << 1)$
    if $I[x_I + step_I] = 1$ then $\mathcal{BS}$ |= $(0x777 << 4)$
    if $\mathcal{BS} > 0$ then
        if HasBit($\mathcal{BS}, 0$) and $I[x_I - step_I - 1]$ then
            Union($L, x_L, x_L - 2 \times step_L - 2$)
        if (HasBit($\mathcal{BS}, 1$) and $I[x_I - step_I]$) or
            (HasBit($\mathcal{BS}, 2$) and $I[x_I - step_I + 1]$) then
            Union($L, x_L, x_L - 2 \times step_L$)
        if HasBit($\mathcal{BS}, 3$) and $I[x_I - step_I + 2]$ then
            Union($L, x_L, x_L - 2 \times step_L + 2$)
        if (HasBit($\mathcal{BS}, 4$) and $I[x_I - 1]$) or
            (HasBit($\mathcal{BS}, 8$) and $I[x_I + step_I - 1]$) then
            Union($L, x_L, x_L - 2$)

kernel COMPRESSION($L$, $step_L$)
    $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
    $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
    $x_L \leftarrow r \times step_L + c$
    Compress($L, x_L$)

kernel FINALLABELING($I$, $step_I$, $L$, $step_L$)
    $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
    $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
    $x_I \leftarrow r \times step_I + c$
    $x_L \leftarrow r \times step_L + c$
    $label \leftarrow L[x_L] + 1$
    $L[x_L]$                      $\leftarrow label \times I[x_I]$
    $L[x_L + 1]$                  $\leftarrow label \times I[x_I + 1]$
    $L[x_L + step_L]$         $\leftarrow label \times I[x_I + step_I]$
    $L[x_L + step_L + 1]$ $\leftarrow label \times I[x_I + step_I + 1]$
```

5 Proposed Algorithms

We propose two new 8-connectivity algorithms, which are optimized variations of UF and KE, obtained through the application of $2{\times}2$ blocks. We also extend our proposals to three-dimensional CCL.

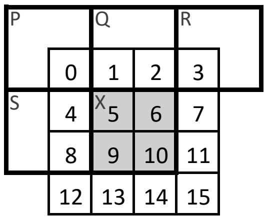


(a) Provisional Labels

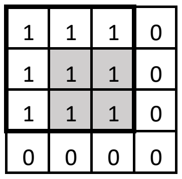


(b) Final Labels

Fig. 6. 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.



(a)



(b)

Fig. 7. (a) shows how pixels in a $4{\times}4$ square centered on the X block are numerated. These numbers correspond to the pixel position in the associated bitset $\mathcal{BS}$. 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 $3{\times}3$ bitmask (`0x777`) corresponding to the neighbors of the top-left internal pixel.

5.1 Block-Based Union Find

Block-based Union Find (BUF) inherits the base structure of Union Find (Section 4.1). The difference is that this algorithm works with block labels. In fact, every thread works on a $2{\times}2$ 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. BUF kernels are depicted in Alg. 2.

Until the end of the algorithm, block labels are stored in the output image, in the upper-left pixel of every block. In this way, we avoid the allocation of unnecessary device memory solely dedicated to block labels, which would be considerably time consuming.

In this case, *Merge* must be applied to blocks rather than to single pixels. The neighborhood mask used is depicted in Fig. 2b. Also in this case, it only contains blocks with a lower raster index than X. Since blocks connections are determined by those of their pixels, for every neighbor block of the mask we have to check whether some of its pixels are connected to some internal pixels of block X. A naive approach that just checks each adjacent block one by one would require to read multiple times the internal pixels, but better alternatives exist. The method we adopted, based on the work by Zavalishin *et al.* [22], consists in a preliminary scan of pixels inside the block: for each foreground, its external neighbors are added to a set of pixels that must be checked in the subsequent step. The aforementioned set is represented as a bitset $\mathcal{BS}$. Each pixel in a $4{\times}4$ square that encloses the X block is given an index in the bitset, as reported in Fig. 7. Initially, every bit is set to 0.

When an internal pixel p is read and recognized as foreground, external pixels q such that $q \in \mathcal{N}_8(p)$ must have their corresponding bits in $\mathcal{BS}$ set to 1. To achieve this goal easily, the whole $3{\times}3$ square centered on p is set accordingly by means of a bitmask (Fig. 7b).

Bitmask `0x777` is required to set neighbors of the top-left pixel inside block X. The other pixels bitmasks 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 is never used. To find out which neighbor blocks are connected to X, the *Merge* kernel must then check which pixels of $\mathcal{BS}$ are set and read their values. A `Union` is performed between X and connected blocks, same as what happens for single pixels in UF.

The BUF *Compression* kernel flattens the *union-find* trees by calling the `Compress` procedure defined in Section 2.1. We also propose a slight improvement of this kernel, that uses `InlineCompress` instead, thus producing the BUF_IC variation. The effects of *Merge* and *Compression* on an input image are depicted in Fig. 6.

FinalLabeling copies the label of each block into internal foreground pixels, thus producing the final output image. Background pixels are given label 0.

5.2 Block-Based Komura Equivalence

The other new algorithm we propose is obtained from KE through the application of the same block-based approach. We therefore call it Block-based Komura Equivalence (BKE). BKE is quite similar to BUF in its structure. The main difference between the two is that BKE, same as KE, starts linking together connected blocks during *Initialization*. This means that the task of finding which blocks are connected to X must be anticipated to the first kernel: the strategy to find connected blocks is the same described for BUF. During *Initialization*, X is linked to the connected neighbor block with the smallest raster index inside the mask, initializing the label of X with the index of the connected neighbor. The process of finding which blocks are connected to X involves a high number of memory accesses. Since the information about connected blocks and foreground internal pixels is needed again during *Reduction* and *FinalLabeling*, we save it

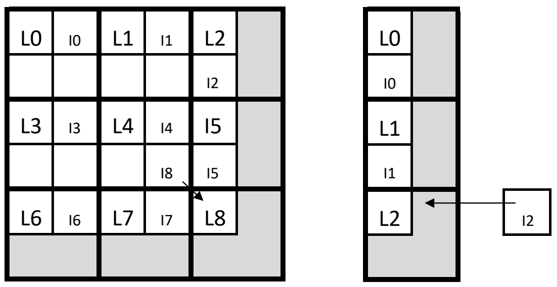


Fig. 8. Location of Label (L) and Information byte (I) of blocks in the output image, used by Block-based Komura Equivalence. In (b), the far-fetched case of a single column image with odd size is displayed. There, an additional byte is necessary to store I2.

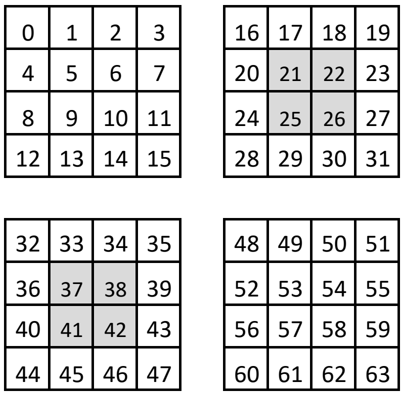


Fig. 9. The figure shows how voxels in a $4{\times}4{\times}4$ cube centered on the X block are numerated. These numbers correspond to positions in the bitset, used for 3D. Only bits corresponding to internal voxels of blocks in the neighborhood mask are used.

in a bitmapped byte, called *information byte*. Its structure is explained in Table 1.

In order to avoid the allocation of additional memory, the information byte is directly stored in the output image. The chosen location is the top-right pixel of every block, that would otherwise be unused until *FinalLabeling*, when the information byte is not necessary anymore. In the case that the image has an odd width, blocks in the last column do not have the top-right pixel, so we store connectivity information in the bottom-left pixel instead. If the image height is odd too, the last block of the last column is composed of the top-left pixel only. In that case, the connectivity information is stored in the bottom-right pixel of its neighbor P. If P does not exists, *i.e.*, when the image is a single row or column with odd size, an extra byte is allocated. So, we allocate additional memory only in degenerate cases, and common images never require extra data structure. Two possibilities of odd-sized images are exemplified in Fig. 8.

As for BUF, the *Compression* kernel flattens the *union-find* trees created in the previous phases to their roots. Two variations of this kernel exist, depending on the use of *InlineCompression*.

In the *Reduction* kernel the information byte is read for each block. Then, `Union` operations are performed between connected neighbor blocks that were not linked to X during *Initialization*. A subsequent *Compression*, followed by the *FinalLabeling*, completes the task. In *FinalLabeling*, the information byte is read again, in order to know whether each internal pixel should be assigned the block label or value 0, which corresponds to the background.

In both Block-based Union Find and Komura Equivalence, the use of blocks drastically reduces the total amount of `Union` and simplifies `Find` operations, w.r.t. their pixel-based original versions. This optimization allows to considerably reduce the number of memory accesses, which represents a bottleneck of parallel CCL algorithms.

TABLE 1
Meaning of the bitmapped byte used in Block-based Komura Equivalence to store information required by different kernels.

Bit	*Meaning*
0	Top-left pixel is foreground
1	Top-right pixel is foreground
2	Bottom-left pixel is foreground
3	Bottom-right pixel is foreground
4	Not used
5	Block Q must undergo `Union`
6	Block R must undergo `Union`
7	Block S must undergo `Union`

5.3 3D Variations

BUF can be adapted to a 3D scenario without changes in its structure. The main difference is that voxels substitute pixels and blocks become $2{\times}2{\times}2$ cubes. The bitset used to represents neighbor voxels in the *Merge* kernel is larger than the 2D analogous. In fact, every voxel in a $4{\times}4{\times}4$ cube must correspond to a bit in $\mathcal{BS}$. The 3D bitset is reported in Fig. 9. The remaining of the algorithm behaves similarly to its 2D counterpart, except for a slight optimization involving *Merge* and *FinalLabeling*. Kernel *FinalLabeling* is responsible for copying each block label in the corresponding foreground internal voxels, and assigning label 0 to background ones. In the absence of information about internal voxels, this procedure requires 8 memory readings, one for each internal voxel of a $2{\times}2{\times}2$ cube. The information about internal voxels must be retrieved anyway in *Merge*. So, we made *Merge* also responsible for writing which voxels are foreground in a bitmapped byte, and for storing it in the output image, similarly as what happens with the information byte of BKE. Then, in *FinalLabeling*, every thread just needs to read a single byte in order to know the internal configuration of X.

BKE is modified to suit 3D CCL in a similar way to BUF: $2{\times}2{\times}2$ cubes are used in place of $2{\times}2$ squares, and the bitset that represents neighbor voxels is the same described above. The information byte used by the 2D variation becomes an information integer that requires 3 bytes in this case. In fact, it must be large enough to store both internal pixels values (8 bits) and neighbor blocks that need to undergo `Union` with X (13 bits). The overall behavior and the extra information storing strategy remain exactly the same as in the 2D counterpart.

6 YACCLAB

When measuring the performance of an algorithm in terms of execution speed there are three main factors to be con-

```
class Labeling {
public:
   std::unique_ptr<YacclabTensorInput> input_;
   std::unique_ptr<YacclabTensorOutput> output_;
   PerformanceEvaluator perf_;

   Labeling() {}
   virtual ~Labeling() = 0;

   virtual void PerformLabeling() {}
   virtual void PerformLabelingWithSteps() {}
   virtual void FreeLabelingData() { output_->Release(); }
};
```

Listing 1. Simplified version of labeling base class. CCL algorithms have to inherit from a specialization of this class. Available specializations are Labeling2D, Labeling3D, GpuLabeling2D, and GpuLabeling3D.

sidered: the data on which tests are performed, the underlying hardware and operative system, and implementation details. Nevertheless, few of the paper concerning CCL and published in recent years have compared algorithms using the same data, and even less have released the source code.

In 2016, a public *C*++ benchmark that enables researchers to evaluate the performance of sequential CCL algorithms under extremely variable points of view has been released [30]. The benchmark allows to cope previously described problems providing a public dataset of binary images without any license limitations and an open source implementation of the state-of-the-art algorithms, so that anyone is able to test them on his own setting, verifying any claim found in the literature.

Unfortunately, YACCLAB was specifically designed for 2D sequential algorithms. With this paper we extend the benchmark and the associated datasets to evaluate the algorithms performance also on GPU and on 3D volumes. A comprehensive description of datasets and available tests is reported in the following of this section. The source code of the extendend benchmark is available in [32].

In order to reuse the largest possible part of YACCLAB original code, we needed to adapt the function responsible for running tests to a more generic usage. First of all, we designed a wrapper class for input and output data, called `YacclabTensor`, with sub-classes for 2D and 3D images that can reside in host or device memory. The class automatically handles data transfers, so that a copy of the data is always available in the location where it is needed. To be included in YACCLAB, CCL algorithms must be compliant with a base interface (Listing 1), implementing specific methods to perform tests. They are described in Section 6.2.

Polymorphism comes at a certain cost in term of performance. Anyway, we measured its impact and verified that no overhead is introduced in the execution of labeling algorithms. In fact, only framework operations, whose execution time is not critical, have been made generic.

6.1 Datasets

Following a common practice in the literature, YACCLAB provides a set of datasets that include both synthetic and real images [22], [25], [37]. With this paper we extend the original version of the YACCLAB datasets in order to address also 3D scenarios. All images are provided in 1 bit per pixel PNG format, with 0 being background and 1 being foreground. The extended version of the dataset can be automatically downloaded during the set up of the YACCLAB benchmark or it can be found in [40]. A complete description of the aforementioned datasets is provided below.

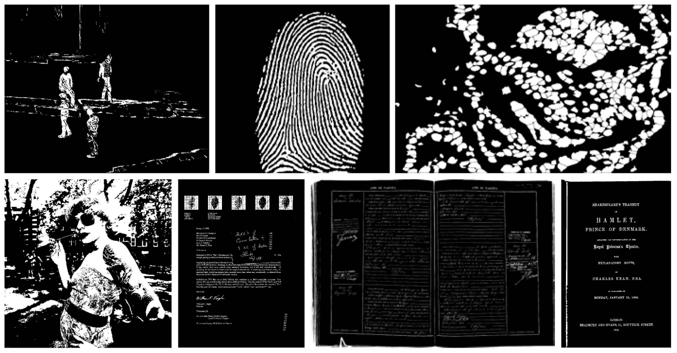


Fig. 10. Samples of the YACCLAB 2D datasets. Clockwise from the top-left: 3DPeS, Fingerprints, Medical, MIRflickr, Tobacco800, XDOCS, Hamlet.

6.1.1 2D Datasets

MIRflickr. This is the Otsu-binarized [41] version of the MIRflickr dataset [42]. It contains a set of 25 000 natural images with few connected components and an average density of 0.4459 foreground pixels.

Medical. This dataset [43] is composed of 343 binary histological images with an average amount of 1.21 million pixels to analyze and 484 components to label.

Hamlet. A scanned version of the Hamlet [44], which counts 104 images with an average amount of 1 447 components to label and an average foreground density of 0.0789.

Tobacco800. It is composed of 1 290 document images collected and scanned using a wide variety of equipment over time. Images size ranges from 1200 $\times$ 1600 to 2500 $\times$ 3200 pixels [45], [46], [47].

XDOCS. A collection of high resolution Italian civil registries images [12], [48], [49], composed of 1 677 images

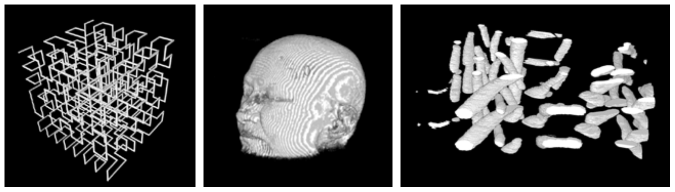


Fig. 11. Samples of the YACCLAB 3D datasets. From left to right: Hilbert space-filling curve, OASIS and Mitochondria medical imaging data.

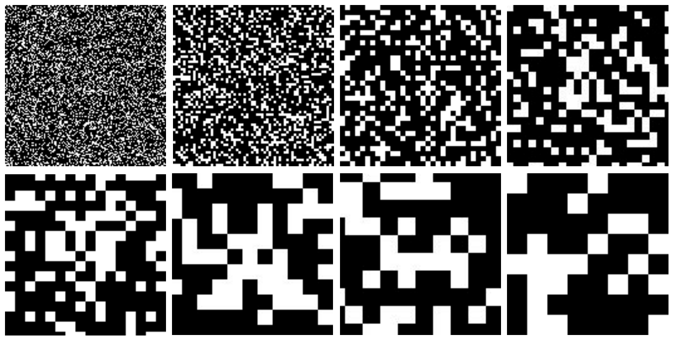


Fig. 12. Samples of the YACCLAB 2D granularity dataset: reported images have a foreground density of 30% and, from left to right, top to bottom, granularities are 1,2,4,6,8,12,14,16.

with an average size of 4853×3387 and $15\,282$ components to analyze. It has a low foreground density of 0.0918.

Fingerprints. This dataset counts 960 fingerprint images [50] binarized using an adaptive threshold [51] and then negated.

3DPeS. A set of images coming from the 3D People Surveillance Dataset [52]. Background subtraction and Otsu thresholding [41] have been applied to the original images in order to generate the foreground binary masks.

6.1.2 3D Datasets

OASIS. This is a dataset of medical MRI data taken from the Open Access Series of Imaging Studies (OASIS) project [53]. It consists of 373 volumes of $256 \times 256 \times 128$ pixels, binarized with the Otsu threshold [41].

Mitochondria. It is the Electron Microscopy Dataset [54], [55], which contains binary sections taken from the CA1 hippocampus for a total of three volumes composed by 165 slices with a resolution of 1024×768 pixels.

Hilbert. This dataset contains six volumes of $128 \times 128 \times 128$ pixels, filled with the 3D Hilbert curve obtained at different iterations (1 to 6) of the construction method. The Hilbert curve is a fractal space-filling curve that represents a challenging test case for the labeling algorithms.

6.1.3 Random Synthetic Images

Two datasets of black and white random noise images have been generated to stress how the behavior of algorithms varies with the percentage of foreground pixels (*density*) and minimum size of foreground blocks (*granularity*) [21]. Resolution is 2048×2048 for two-dimensional images, and $256 \times 256 \times 256$ for three-dimensional volumes. Images and volumes were generated with the Mersenne Twister MT19937 random number generator, implemented in the C++ standard [56]. Density ranges from 0% to 100% with a step of 1%. For every density value, each integer granularity $g \in [1, 16]$ has been considered. Ten images have been generated for every couple of density-granularity values, for a total of $16\,160$, both for 2D and 3D.

6.2 Assessment Strategies

YACCLAB allows to perform different kinds of experiments on CCL algorithms, the first of which is a direct comparison of execution times. In this case, algorithms are run on every image of a dataset, and average execution times are recorded. This evaluation uses the `PerformLabeling()` method, that must therefore implement the whole CCL algorithm, which must include the allocation of necessary data structures.

The second type of experiment available in the benchmark serves to separately evaluate the phases of memory allocation and algorithm execution. Two stage algorithms can also distinguish between the Local Labeling and the Tiles Merging steps. This test highlights how the performance of an algorithm is influenced by the different phases which it is composed of, and allows to better stress strengths and weaknesses of the various strategies. The test, also called *average with steps* in the rest of the paper, exploits `PerformLabelingWithSteps()` method, which records execution times of the different stages separately.

The last of the possible experiments is known as *granularity*. It consists of measuring the execution time of algorithms on synthetic images, in order to evaluate how performance is affected by foreground pixels density and granularity. This approach has been used in the past by several authors, and allows to draw additional conclusions, highlighting behaviors that may not emerge from the tests described above.

It is important to notice that all the experiments provided by YACCLAB for GPU algorithms are based on the assumption that the input image is already in the GPU memory before the beginning of the algorithm, and that the output is required in the device memory as well. Therefore, the allocation of the output GPU image is considered in the total elapsed time, alongside potential allocation and deallocation of additional data structures that algorithms may need. Conversely, neither the allocation of the input image nor possible data transfers between host and device are considered. This also applies to every experimental result reported in this manuscript, unless otherwise specified.

Actually, the YACCLAB benchmark provides also other assessment strategies. Anyway, only those related to the focus of the paper have been reported here, and used to evaluate our proposals.

7 Comparative Evaluation

The proposed strategies are evaluated by comparing their performance with state-of-the-art algorithms.

Experimental results reported and discussed in this Section are obtained running the YACCLAB benchmark on an Intel Core i7-4790 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.

TABLE 2
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 *.

	2D Images							3D Volumes		
	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*	*Hilbert*	*Oasis*	*Mitochondria*
BUF*	0.512	0.441	2.161	1.313	0.495	3.268	12.088	2.119	6.792	65.829
BUF_IC*	0.508	0.440	2.112	1.299	0.494	**3.163**	11.764	2.119	6.802	**65.825**
BKE*	0.509	0.429	2.190	1.221	0.452	3.409	11.989	**2.097**	**6.728**	68.251
BKE_IC*	**0.501**	**0.427**	**2.073**	**1.186**	**0.449**	3.173	**11.253**	2.123	6.815	68.441
BE [22]	1.517	1.164	4.376	2.730	1.165	5.966	20.278	5.338	10.779	93.154
UF [29]	0.594	0.529	3.000	2.040	0.659	4.304	17.316	3.504	17.852	129.236
OLE [28]	1.211	1.128	5.358	3.013	1.281	8.173	35.242			
LBUF [36]	0.573	0.509	2.776	1.699	0.541	3.889	15.039			
KE [38]	0.568	0.481	2.717	1.622	0.526	3.978	15.432			
DLP [37]	0.657	0.486	3.097	1.697	0.602	5.002	18.172			

For convenience, the acronyms used to refer to the available algorithms are summarized here: BUF (Block-based Union Find) and BKE (Block-based Komura Equivalence) are the two algorithms proposed with this paper. BE is the Block Equivalence algorithm proposed by Zavalishin *et al.* [22], UF is Union Find by Oliveira *et al.* [29], OLE is the Optimized version of Label Equivalence presented in [28] by Kalentev *et al.*, LBUF is the Line-Based Union Find algorithm by Yonehara *et al.* [36], KE is the Komura Equivalence introduced in [20] by Komura, and DLP is the Distanceless Label Propagation algorithm by Cabaret *et al.* [37]. Finally IC identifies the variation described in Section 2.1 of BUF and BKE algorithms. The 3D version is available for all the algorithms except OLE, LBUF, KE, and DLP.

All the aforementioned 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 first experiment carried out is the comparison between algorithms in terms of average execution time over real datasets (Table 2). Our proposals, BUF and BKE, and all their variations outperform state-of-the-art algorithms on this test case. The best performing algorithm over 2D dataset is the IC variation of BKE, while, for what concerns 3D datasets, the base version provides better results.

As regards 2D, the speed-up between BKE_IC and KE, state-of-the-art competitor, varies from 1.1× on easy to label datasets (*3DPeS*) to 1.4× on datasets with a high number of complex connected components (*XDOCS*). In 3D test cases, the speed-up between BKE and BE ranges from 1.4× to 2.5×. Anyway, the performance of BUF and BKE are very close, and the same can be stated for their IC variations.

The *InlineCompression* optimization tries to reduce the number of memory accesses that an algorithm has to perform during the *Compression* phase, *i.e.*, the number of parent nodes a thread has to traverse to reach the root of the *union-find* equivalence tree. To achieve this goal an algorithm has to perform additional write operations that are of course expensive. Therefore, the benefit introduced by IC is valuable only when a convenient trade-off between saved readings and additional writings is achieved. This is linked to the complexity of the equivalence trees created and updated during *Initialization* and *Merge/Reduction* phases, which strictly depends on the nature of the input image like shape and dimension of the objects it contains, and on the order in which threads are executed. For this reasons, the definition of a break-even is very hard and cannot be done a priori.

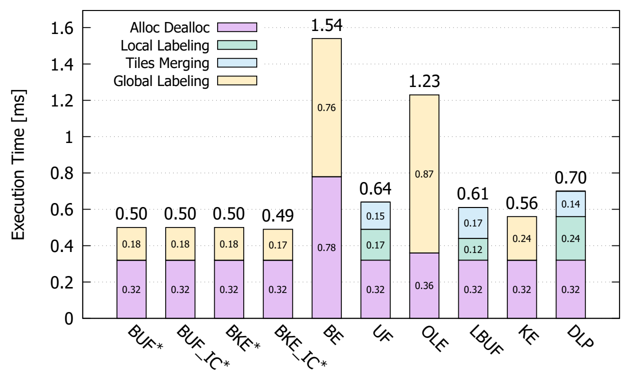
Anyway, it is possible to observe that the use of IC always improves the performance of both BUF and BKE algorithms on 2D real cases datasets taken into account. In order to demonstrate this behavior, Table 3 is provided. In Table 3a the average number of memory accesses used by BUF and BUF_IC in the *Compression* kernel is provided for the *Fingerprints*, *Medical*, *Tobacco800* and *XDOCS* datasets. This table demonstrates the strict correlation between the difference of memory reads and the speed-up of BUF_IC algorithm w.r.t. BUF. A similar consideration can be drawn for BKE (Table 3b). As regards 3D datasets, instead, the use of IC may slightly increase the total execution time. In these test cases the *union-find* trees tend to be short, and hardly any memory reads are saved by this optimization. This has a greater effect on the BKE algorithm since it performs the *Compression* kernel twice.

To better investigate the algorithms behavior, Fig. 13 and Fig. 14 are also reported for 2D and 3D datasets, respectively. In these figures, bar charts report separately the time needed for allocating data structures from the time required by the global labeling procedure. Moreover, if an algorithm employs two clearly distinct phases to compute local labeling and perform tiles merging, the time required by each of them is displayed separately. The Hamlet dataset has been omitted from Fig. 13 for space constraints. Results are very close to those of the other document datasets, *i.e.*, Tobacco800 and XDOCS.

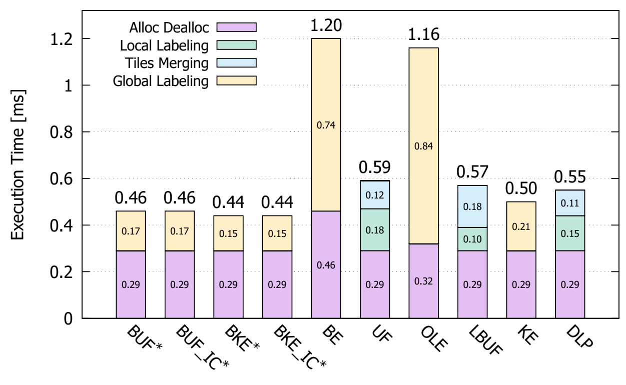
TABLE 3
Effects of *InlineCompression* on the *Compression* kernel of BUF (a) and BKE (b), in terms of average amount of memory reads.

		BUF	BUF_IC	*Diff*
(a)	*Fingerprints*	37 395	30 147	7248
	Medical	313 871	252 421	61 450
	Tobacco800	273 592	215 796	57 796
	XDOCS	2 021 381	1 496 108	525 273

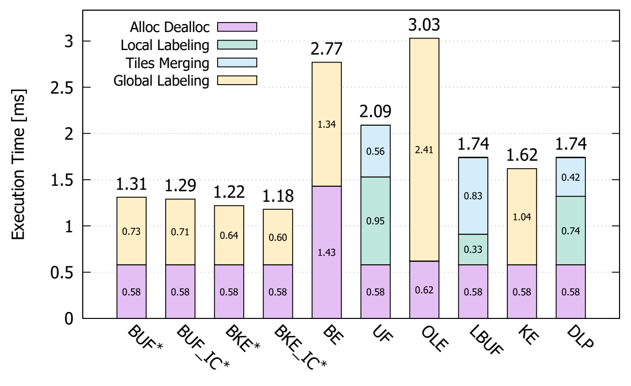
		BKE	BKE_IC	*Diff*
(b)	*Fingerprints*	23 868	21 130	2738
	Medical	149 334	145 627	3707
	Tobacco800	133 998	130 053	3945
	XDOCS	1 042 158	975 209	66 948



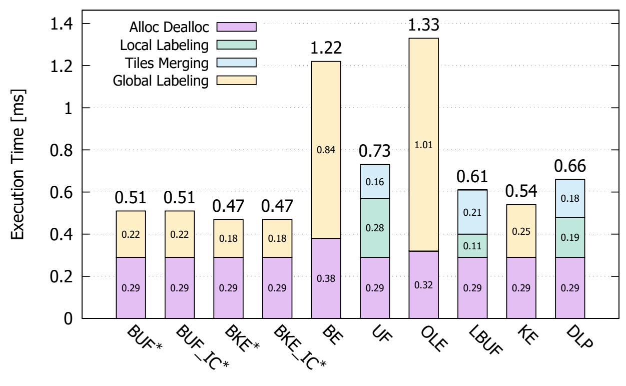
(a) 3DPeS



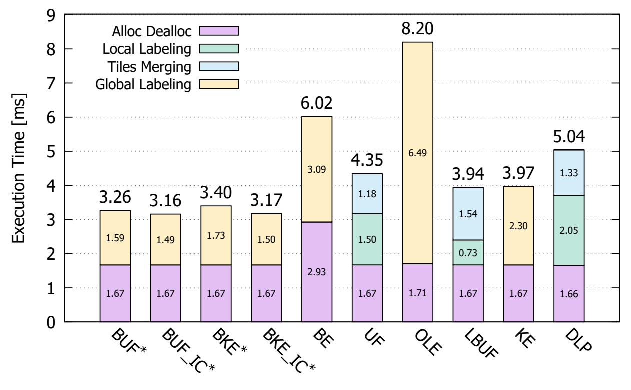
(b) Fingerprints



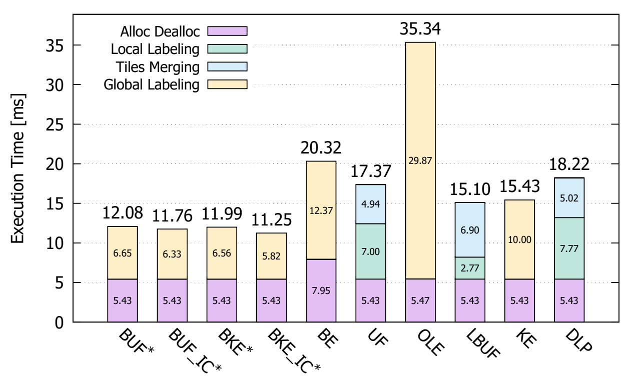
(c) Medical



(d) MIRflickr



(e) Tobacco800



(f) XDOCS

Fig. 13. Average run-time with steps test on 2D datasets. Numbers are given in ms and our proposals are identified with $*$. Lower is better. Best viewed online.

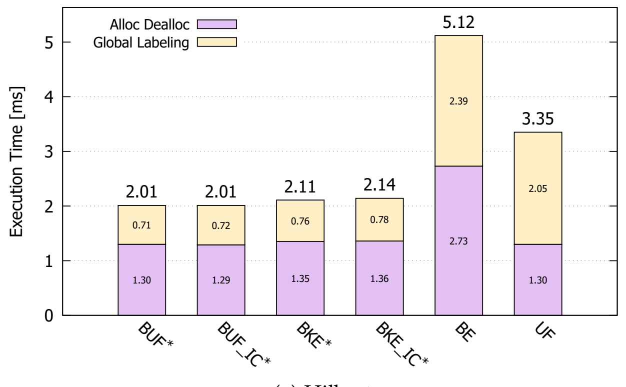
Focusing on Fig. 13, the allocation time is the same for each strategy, but for BE and OLE. Indeed, all the algorithms must only allocate memory for the output image. OLE, that is an iterative algorithm, requires an additional byte to check whether the convergence has been reached or not. This costs $0.03 - 0.04$ ms independently of the input image size. On the other hand, 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.

The charts show that the allocation and deallocation of device memory require a significant amount of the total execution time: that is why allocating or freeing global memory in performance-sensitive code should be done only when strictly necessary [35]. Anyway, a CCL algorithm cannot avoid the allocation of the output image. Thus, optimization can be applied only to the core part of the labeling procedure. In the light of these considerations, if we remove this fixed allocation cost, the speed-up of BKE_IC over KE moves from $1.1\times - 1.4\times$ to $1.4\times - 1.7\times$ on 2D datasets.

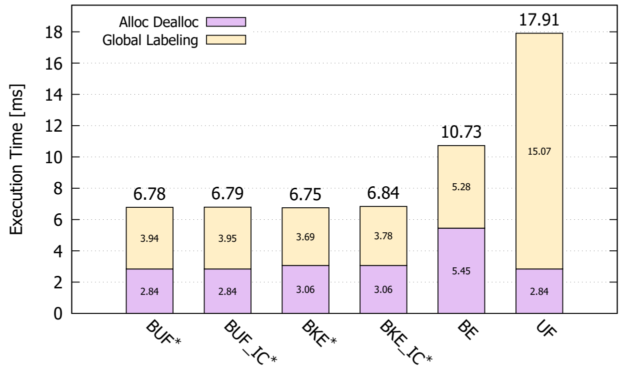
The execution time of the OLE global labeling is always the worst, given that it requires many iterations over the input image to update the output one until convergence.

The block scan approach introduced by BE allows to highly reduce the operations required by OLE, thus reducing the global labeling time at the expense of allocation step.

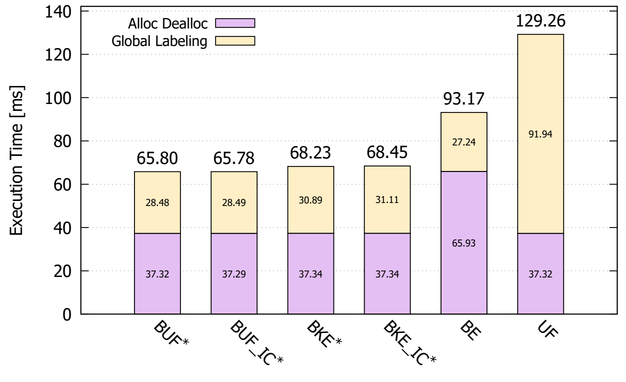
Generally, UF has better perfomance than BE on 2D datasets. This is especially true on 3DPes, MIRflickr and Fingerprints, where many connected components are large and irregularly shaped. In such circumstances, BE requires more iterations to recognize an object as a single component. In the cases of components with simpler shapes, the perfor-



(a) Hilbert



(b) Oasis



(c) Mitochondria

Fig. 14. Average run-time with steps test on 3D datasets. Numbers are given in ms and our proposals are identified with $*$. Lower is better. Best viewed online.

mances of BE Global Labeling are comparable or better than those of UF. Anyway, the allocation time is too high for BE to be competitive in these scenarios.

The line-based variation of UF (LBUF) allows to simplify the logic of Local Labeling, reducing the neighborhood of a pixel to be checked during the *Local Merge* procedure. This algorithm moves the complexity to Tiles Merging, but always improves performance w.r.t. UF on 2D datasets. DLP tries to put together positive aspects of both UF and LE, but only in few cases is able to outperform UF. KE, thanks to an optimized *Initialization* kernel w.r.t. UF (Section 4.2), always improves its performance.

With our approaches we are able to combine the strengths of different strategies and benefit from the use of a block-based algorithm without increasing the amount of required memory, thus obtaining the lowest execution times.

TABLE 4
Average number of iterations required by the OLE algorithm on images of increasing density at 1-granularity level.

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

Similar considerations can be drawn for 3D experiments (Fig. 14). On Mitochondria, the Global Scan of BE is faster than those of our proposals. This can be again explained considering the nature of the dataset, which is mostly composed of convex objects. Nevertheless, the massive memory usage makes the total execution time of BE higher than those of the proposed algorithms. That said, there are cases in which it could be fair to compare algorithms without considering memory allocation: in an embedded system in which images are always captured with the same size, for example, it could be realistic to allocate memory only once. In such scenarios, BE may be the best choice for data similar to Mitochondria.

Following a common approach in literature [26], [34], [57], [58], additional tests have been performed on images with increasing foreground density and granularity (Fig. 15 and Fig. 17), in order to highlight strengths and weaknesses of the algorithms. To make the charts more readable the IC version of both BKE and BUF has been omitted.

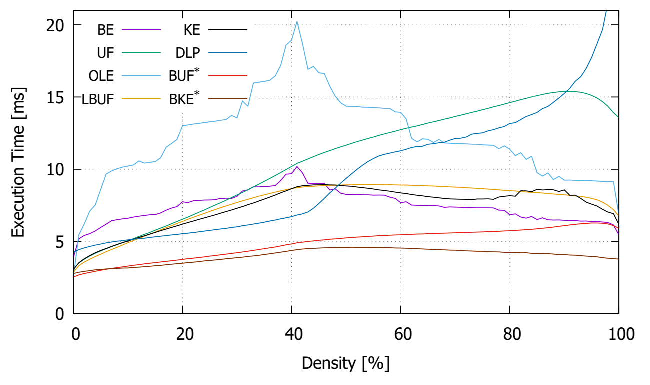
Focusing on 2D datasets, it can be said that, independently of the pixel granularity, OLE has an increasing trend in the execution time up to 40% of foreground density, and then a decreasing one after this value. This behavior is strictly linked to the iterative nature of the algorithm. Indeed, the number of iterations required by the labeling procedure to converge reaches the highest value when foreground density is about 40% (Table 4). 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 is linked to image density. The more `Union` there are, the more memory accesses and atomic operations are performed.

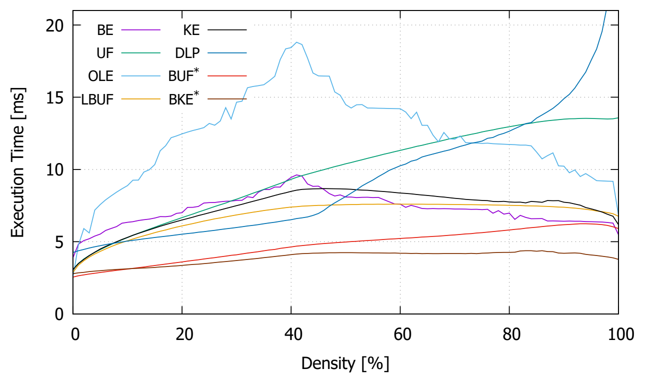
As shown in Fig. 15, KE and LBUF have a behavior equal to UF with densities up to 20%. Then, with larger connected components, KE *Initialization* is able to create shallower equivalence trees, and LBUF is able to create long lines of equivalent labels, again reducing the tree traversals required later. While using different strategies, KE and LBUF have similar trends with respect to pixels density, both largely improving on UF at high densities.

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 atomic operations and memory accesses, 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.

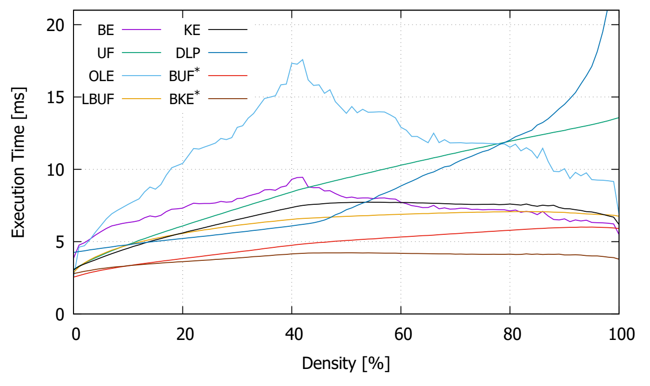
The execution times of BUF and BKE are very similar for low density images. Then, BKE has a decreasing trend after 40%. In fact, after that value large connected components



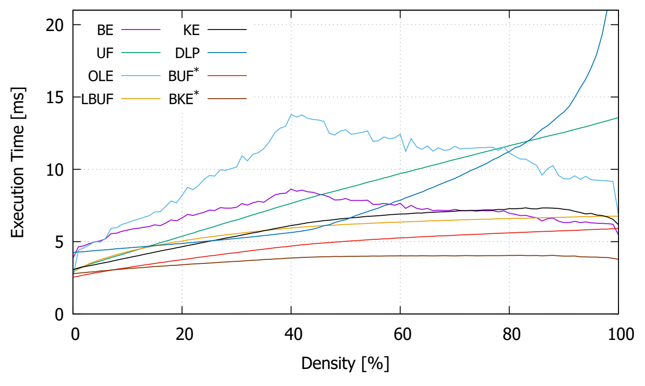
(a) g = 1



(b) g = 2



(c) g = 4



(d) g = 8

Fig. 15. Two-dimensional tests on randomly generated images. Numbers are given in ms and our proposals are identified with $*$. Lower is better. Best viewed online.

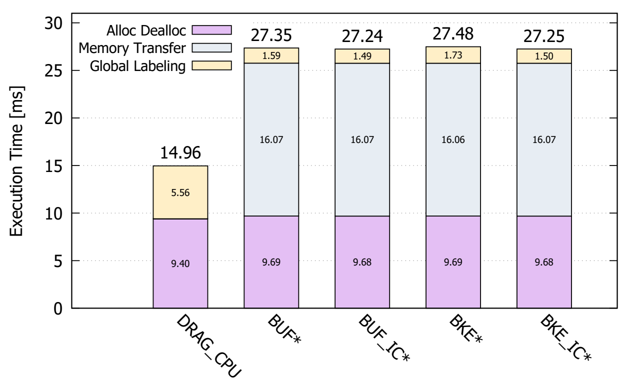


Fig. 16. Comparison of the proposed algorithms with a state-of-the-art CPU-based CCL algorithm (DRAG) on Tobacco800 dataset. Numbers are given in ms and our proposals are identified with $*$. Lower is better. Best viewed online.

start to appear, and the effect of the improved *Initialization* can be especially appreciated. The relationships between the algorithms remain the same as the granularity grows, but curves tend to be flatter.

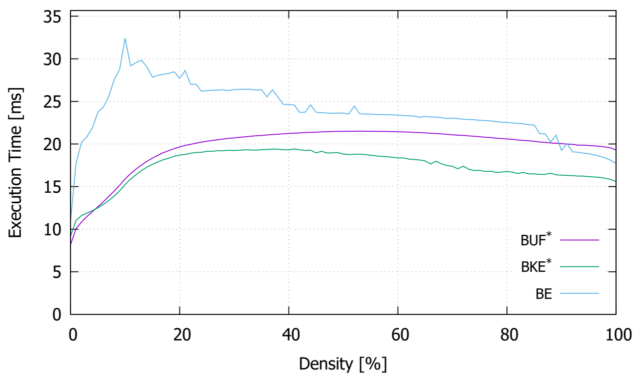
For the sake of completeness, the results of tests over randomly generated three-dimensional volumes are reported as well, in Fig. 17. The same considerations drawn for the two-dimensional case can be applied.

Finally, Fig. 16 compares the proposed strategies to a state-of-the-art CPU-based CCL algorithm (DRAG [18]) on Tobacco800 dataset. In this scenario, differently from the other results reported in the manuscript, both the input and the output images are in the host memory. Therefore, the elapsed time of the GPU algorithms includes the allocation/deallocation of GPU data structures, the allocation of the output image in CPU, and the data transfers between host and device memory. On the other hand, the CPU algorithm includes only the allocation of the output image and the required data structures. When considering only the Global Labeling, GPU algorithms have a speed-up between 3.2× and 3.7×. Anyway, given the extremely high transfer time between host and device, a CLL GPU algorithm is preferable to a CPU one only as part of a GPU pipeline.

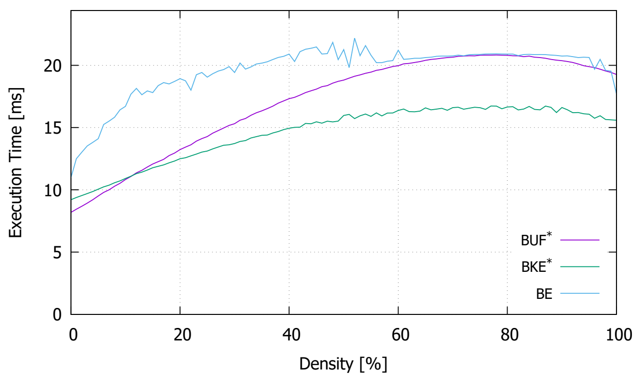
8 Conclusion

In this paper, the problem of GPU-based Connected Components Labeling in binary images and volumes has been addressed. Two new algorithms have been proposed, Block-based Union Find (BUF) and Block-based Komura Equivalence (BKE), which have been obtained by combining existing strategies with a block-based approach, to considerably reduce the number of memory accesses and consequently improve time performance.

Experiments on a wide selection of both real case and synthetically generated datasets confirm that our proposals represent the state-of-the-art for GPU-based connected components labeling. The datasets cover most of the fields where CCL is commonly used, and allow to evaluate the correlation of performance to specific characteristics of the



(a) g = 1



(b) g = 8

Fig. 17. Three-dimensional tests on randomly generated images. Numbers are given in ms and our proposals are identified with *. Lower is better. Best viewed online.

input. Among the two proposals, BKE demonstrated superior performance in every test case, except for images with very low density. In fact, on random images, BUF has better performance than BKE for density below $5-10$, depending on the granularity.

Moreover, a public benchmarking framework for sequential CCL algorithms, YACCLAB, has been extended with added functionalities, such as the possibility of evaluating GPU algorithms alongside CPU ones. Additionally, its collection of datasets has been enriched with new real case and synthetic datasets of three-dimensional volumes.

References

[1] A. Rosenfeld and J. L. Pfaltz, "Sequential Operations in Digital Picture Processing," *Journal of the ACM*, vol. 13, no. 4, pp. 471–494, Oct. 1966.

[2] A. Dubois and F. Charpillet, "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*, 2012, pp. 7–13.

[3] C. Zhan, X. Duan, S. Xu, Z. Song, and M. Luo, "An Improved Moving Object Detection Algorithm Based on Frame Difference and Edge Detection," in *Image and Graphics, 2007. ICIG 2007. Fourth International Conference on*. IEEE, 2007, pp. 519–523.

[4] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-Time Image Segmentation on a GPU," in *Facing the multicore-challenge*. Springer, 2010, pp. 131–142.

[5] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Improving Skin Lesion Segmentation with Generative Adversarial Networks," in *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE, 2018, pp. 442–443.

[6] A. Körbes, G. B. Vitor, R. de Alencar Lotufo, and J. V. Ferreira, "Advances on Watershed Processing on GPU Architecture," in *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2011, pp. 260–271.

[7] A. Eklund, P. Dufort, M. Villani, and S. LaConte, "BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs," *Frontiers in neuroinformatics*, vol. 8, p. 24, 2014.

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

[9] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Augmenting data with GANs to segment melanoma skin lesions," *Multimedia Tools and Applications*, 2019.

[10] L. Canalini, F. Pollastri, F. Bolelli, M. Cancilla, S. Allegretti, and C. Grana, "Skin Lesion Segmentation Ensemble with Diverse Training Strategies," in *18th International Conference on Computer Analysis of Images and Patterns*. Springer, 2019.

[11] T. Lelore and F. Bouchara, "FAIR: A Fast Algorithm for Document Image Restoration," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 2039–2048, 2013.

[12] 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.

[13] T. Berka, "The Generalized Feed-forward Loop Motif: Definition, Detection and Statistical Significance," *Procedia Computer Science*, vol. 11, pp. 75–87, 2012.

[14] M. J. Dinneen, M. Khosravani, and A. Probert, "Using OpenCL for Implementing Simple Parallel Graph Algorithms," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011, p. 1.

[15] S. Byna, M. F. Wehner, K. J. Wu *et al.*, "Detecting atmospheric rivers in large climate datasets," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 7–14.

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

[17] L. He, X. Zhao, Y. Chao, and K. Suzuki, "Configuration-Transition-Based Connected-Component Labeling," *IEEE Transactions on Image Processing*, vol. 23, no. 2, pp. 943–951, 2014.

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

[19] D. Zhang, H. Ma, and L. Pan, "A Gamma-signal-regulated Connected Components Labeling Algorithm," *Pattern Recognition*, 2019.

[20] 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.

[21] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel Light Speed Labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors," *Journal of Real-Time Image Processing*, pp. 1–24, 2016.

[22] S. Zavalishin, I. Safonov, Y. Bekhtin, and I. Kurilin, "Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU," *Electronic Imaging*, vol. 2016, no. 2, pp. 1–7, 2016.

[23] 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.

[24] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, "How does Connected Components Labeling with Decision Trees perform on GPUs?" in *18th International Conference on Computer Analysis of Images and Patterns*. Springer, 2019.

[25] D. P. Playne and K. Hawick, "A New Algorithm for Parallel Connected-Component Labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1217–1230, Jun. 2018.

[26] C. Grana, D. Borghesani, and R. Cucchiara, "Optimized Block-based Connected Components Labeling with Decision Trees," *IEEE Transactions on Image Processing*, vol. 19, no. 6, pp. 1596–1609, 2010.

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

[28] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2D grid using CUDA," *Journal of Parallel and Distributed Computing*, vol. 71, no. 4, pp. 615–620, 2011.

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

[30] 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.

[31] 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.

[32] The YACCLAB Benchmark. Accessed on 2019-03-21. [Online]. Available: https://github.com/prittt/YACCLAB

[33] 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.

[34] 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.

[35] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[36] K. Yonehara and K. Aizawa, "A line-based connected component labeling algorithm using GPUs," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 341–345.

[37] L. Cabaret, L. Lacassagne, and D. Etiemble, "Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs," in *Seventh International Conference on Image Processing Theory, Tools and Applications*. IPTA, 11 2017.

[38] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, "Optimizing GPU-Based Connected Components Labeling Algorithms," in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*. IEEE, 2018, pp. 175–180.

[39] ——, "A Block-Based Union-Find Algorithm to Label Connected Components on GPUs," in *Proceedings of the 20th International Conference on Image Analysis and Processing (ICIAP)*. Springer, 2019.

[40] The YACCLAB 3D Dataset. Accessed on 2019-03-21. [Online]. Available: http://aimagelab.ing.unimore.it/files/YACCLAB_dataset3D.zip

[41] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[42] 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.

[43] 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.

[44] The Hamlet Dataset. Accessed on 2019-03-21. [Online]. Available: http://www.gutenberg.org

[45] 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.

[46] D. Lewis, G. Agam, S. Argamon, O. Frieder, D. Grossman, and J. Heard, "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*. ACM, 2006, pp. 665–666.

[47] "The Legacy Tobacco Document Library (LTDL)," University of California, San Francisco, 2007.

[48] 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.

[49] ——, "XDOCS: An Application to Index Historical Documents," in *Italian Research Conference on Digital Libraries*. Springer, 2018, pp. 151–162.

[50] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*. Springer Science & Business Media, 2009.

[51] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000.

[52] 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.

[53] D. S. Marcus, A. F. Fotenos, J. G. Csernansky, J. C. Morris, and R. L. Buckner, "Open Access Series of Imaging Studies (OASIS): Longitudinal MRI Data in Nondemented and Demented Older Adults," *Journal of cognitive neuroscience*, vol. 22, no. 12, pp. 2677–2684, 2010.

[54] The Electron Microscopy Dataset. Accessed on 2019-03-21. [Online]. Available: https://cvlab.epfl.ch/data/data-em/

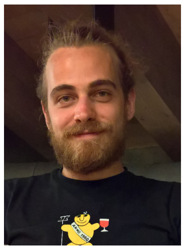
[55] A. Lucchi, Y. Li, and P. Fua, "Learning for Structured Prediction Using Approximate Subgradient Descent with Working Sets," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1987–1994.

[56] 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.

[57] 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.

[58] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors," in *International Conference on Image Processing*. IEEE, 2015, pp. 3486–3489.

Stefano Allegretti received the B.Sc. and M.Sc. degrees in Computer Engineering from Università degli Studi di Modena e Reggio Emilia, Italy. He is currently a postgraduate researcher at the AImagelab Laboratory at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli Studi di Modena e Reggio Emilia, Italy. His research interests include deep learning, pattern recognition, and image processing.



Federico Bolelli received the B.Sc. and M.Sc. degrees in Computer Engineering from Università degli Studi di Modena e Reggio Emilia, Italy. He is currently pursuing the Ph.D. degree at the AImagelab Laboratory at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli studi di Modena e Reggio Emilia, Italy. His research interests include image processing, algorithms optimization, deep learning, medical imaging, and historical document analysis.

Costantino Grana graduated at Università degli Studi di Modena e Reggio Emilia, Italy in 2000 and achieved the Ph.D. in Computer Science and Engineering in 2004. He is currently Associate Professor at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli studi di Modena e Reggio Emilia, Italy. His research interests are mainly in computer vision and multimedia and include analysis and search of digital images of historical manuscripts and other cultural heritage resources, multimedia image and video retrieval, medical imaging, color based applications, motion analysis for tracking and surveillance. He published 5 book chapters, 34 papers on international peer-reviewed journals and more than 100 papers on international conferences.