On GPU Connected Components and Properties

A Systematic Evaluation of Connected Component Labeling Algorithms and their Extension for Property Extraction

Pedro Asad, Ricardo Marroquim, and Andréa L. e L. Souza
Computer Graphics Laboratory, Federal University of Rio de Janeiro

***Abstract*—Connected Component Labeling (CCL) is a fundamental image processing problem that has been studied in many platforms, including GPUs. A common approach to CCL performance analysis is studying the total processing time as a function of abstract image features, like the number of connected components or the fraction of foreground pixels, and input data usually includes synthetic images and segmented video datasets. In this work, we develop on these ideas and propose an evaluation methodology for GPU CCL algorithms based on synthetic image patterns, addressing the nonexistence of a standard and reliable benchmark in the literature. Our methodology, applied on two important algorithms from existing literature, uncovers their data dependency with great detail, and allows us to model their processing time in three real-world video datasets as functions of abstract, high-level, image concepts. We also apply our methodology for studying the memory and performance requirements of two strategies for computing connected component properties: an existing memory-hungry approach, and a new memory-preserving strategy.**

***Index Terms*—Parallel processing, region growing, labeling, image analysis, connected components, Graphics Processors**

I. Introduction

CONNECTED COMPONENT LABELING (CCL) is a fundamental operation in many image processing and computer vision pipelines and has been applied in different scenarios and architectures [5], [9]. It is usually described as the task of partitioning image pixels into maximal connected subsets and assigning unique labels to them. Connectivity depends on a fixed adjacency criterion, such as 4-connectivity or 8-connectivity and on a prior segmentation phase that separates pixels into different categories. A common case is that of binary images, in which pixels are classified as either foreground or background. A convenient output format consists in an array (with the same dimensions as the input image) containing, at each pixel position, a numeric label that identifies the connected component (CC) the pixel belongs to. From that format, various other information such as size, bounding boxes, and object center may also be calculated and used in tracking algorithms [17], [30], [38].

During the last decade, the advent of the GPU as a general-purpose, massively parallel programming platform that is able to deliver considerable speedups over traditional CPUs at end-consumer prices had a profound impact on real-time computing research. Many methods in the fields of engineering, physics and simulation received a performance boost after being adapted (or even completely reinvented) to take advantage of the GPU's computation and memory models [20], [24]. This has naturally led to the invention of some CCL algorithms designed specifically for GPUs [1], [8], [10]–[12], [15], [23], [28], [33], [36], [40]. However, despite many techniques having been proposed, research literature still lacks a comparative study or standard benchmark. Some of these works do present comparisons with others, but the emphasis is usually on measuring GPU vs. CPU speed-ups using specific image datasets.

Our contributions are:

- An evaluation methodology for GPU CCL algorithms based on synthetic image patterns that allows for predicting processing times in real-world videos.
- The extension of an existing strategy that computes connected component sizes [11] for the computation of centroids and co-variance matrices in the *Label Equivalence* (LE) [8] and *Tile Merging* (TM) [33] algorithms.
- A new strategy for computing CC properties that may be combined with *Inclusive Scans* [7] to drastically reduce GPU memory usage and GPU to CPU transfer times.
- As minor contribution, we also explore some implementation choices of the LE method that were not explicitly discussed in past literature [8], [11].

The paper is organized as follows: in Section II we review the literature on CCL algorithms with emphasis on GPU approaches; Section III discusses the extended and new property computation strategies; Section IV describes our evaluation methodology; Section V presents the experimental results; finally, Section VI concludes the paper and suggests future directions.

II. Literature review

A good discussion on the main categories of sequential CCL methods, including some representative works, may be found in [5], [37]. Algorithms developed in the last decade (not including GPU algorithms), are addressed specifically in [9]. There are also considerably many works concerning hardware implementations of CCL [13], [18], [31], [34], [41]. In this section, we will briefly review some of the existing GPU algorithms that are most relevant to our discussion. We will attain our discussions to binary images.

A. Existing GPU methods

GPUs are massively parallel processors with limited synchronization mechanisms, making it difficult to implement data structures such as trees and linked lists, normally used in CPU algorithms, in a thread-safe way. Since CCL is a global problem, GPU algorithms have to cope with these limitations by employing local operations and merging partial solutions iteratively or recursively.

Label Equivalence (LE) GPU methods [8], [10], [11], [40] are akin to sequential two-pass algorithms in that they represent and manipulate equivalence chains. The first method in this category was published by Hawick et al. [8], who studied its application to hyper-cubic mesh graphs in the context of Monte-Carlo phase state transition simulations. Their algorithm, which we dub as *Hawick's Label Equivalence* (HLE, for short), is comprised of an initialization part (usually referred as the *Prelabel* kernel) and an iterative part, further divided into the *Scan*, *Analysis* and *Update* kernels.

Given an input binary image of size $w \times h$, the Prelabel kernel initializes a label array L (also of size $w \times h$), setting any background pixels to a special value and each foreground pixel $\mathbf{p} = (x, y)$ to a *provisional label* equal to its own raster $x + yw$, effectively turning $\mathbf{p}$ into the representative pixel of a CC containing only itself. The Scan operation merges adjacent sub-components by exchanging labels between neighbors that belong to disjoint equivalence chains, and the Analysis and Update phases ensure that all pixels in the same equivalence chain point to a single representative element, the *representative pixel*. After a number of iterations, sub-components converge to fully connected components, Hence, the Scan phase detects no changes to be made, and the algorithm stops.

Kalentev et al. [11] proposed an LE variation (*Kalentev's Label Equivalence*, or KLE) that saves a considerable amount of space and time by not requiring an additional equivalence table, as HLE does, which also renders the *Update* kernel unnecessary, since label equivalences are represented directly in the output array. They also employ regular memory writes to global memory during the Scan phase, instead of using atomic operations, as in HLE, which they claim to accelerate the algorithm, despite the possibility of increasing the number of iterations. As an additional contribution, they describe the calculation of CC sizes by employing an additional array that is initialized during the Prelabel phase and updated during Analysis using atomic addition instructions. Other works in this category include [10], which is very similar to HLE, but divides the iterative part in more kernels and [40], that employs an optimization [6] that groups pixels in 2×2 blocks, making it possible to process one fourth of the original image size in the 8-connected neighborhood case.

Tile Merging (TM) algorithms, on the other hand, are based on solving the CCL problem locally, usually in shared memory, for rectangular regions, or *tiles*, then merging these local solutions into a global one. Oliveira et al. [23], who also described an LE implementation similar to HLE, described the first TM method, which was comprised of the *Local Merge*, *Global Merge* and *Path Compression* kernels. Although their work is not completely reproducible, due to absence of a detailed explanation of the Global Merge kernel and online unavailability of the source-code provided by them at the time of this writing, Stava et al. [33] implemented what seems to be a very similar technique. They also mention, but not discuss, the possibility of using the *Inclusive Scan* operation [7] in order to obtain consecutive labels for the connected components, a topic that we will address again in Section III. Yonhehara et al. [39] presented a variation of Oliveira's method that uses $w \times 1$ stripes for the Local Merge, instead of rectangles. Their experiments allegedly indicate a speedup of 1.4x over their implementation of Oliveira's method, at the resolution of 1024×1024 pixels, but their Global Merge kernel is also not clearly explained.

Many other works [1], [4], [12], [15], [25], [26], [28], [29], [32], [35], [36] proposed GPU CCL algorithms, some of which may be classified as optimizations and/or applications of LE and TM.

III. Property calculation

In this section, we discuss the computation of CC properties in GPU CCL algorithms. Various object properties used in the context of object tracking [17], [30], [38] may be calculated by following the strategy of Kalentev et al. [11], such as size (number of pixels), bounding boxes, center point, etc. Some works [3], [14], [19] estimate these object features using object representations other than connected components, but they can be computed directly from CCL. We discuss the computation of size, center and co-variance matrix, for the following reasons:

- These properties allow us to compute the principal axes of the CCs, which may be used for obtaining oriented bounding boxes or error ellipses, both more abstract object models that are sometimes employed in tracking.
- They take up a considerable storage space, significantly impacting LE and TM's performance, allowing for a deeper discussion of performance sensitivity.

We refer to the generalization of the property computation technique by Kalentev et al. to both LE and TM as *Early Computation*. We review their technique in Section III-A, before presenting the necessary modifications to compute our properties of interest in Section III-B and the particularities of Early Computation in the TM algorithm in Section III-C. Finally, we propose a completely new property computation strategy, *Late Computation*, in Section III-D.

A. CC size computation in KLE: Early Computation

In order to compute CC sizes, Kalentev et al. [11] modified the Prelabel kernel to initialize an array A_S (with the same size as L) with per-pixel provisional sizes (1 for foreground pixels and 0 for background ones) and the Analysis kernel to store the sum of sizes of merged sub-components at representative positions. For instance, if pixels $\mathbf{p}_1$ and $\mathbf{p}_2$ were the representative pixels of connected components C_1 and C_2, and C_2 was merged into C_1 during some pass of the Scan kernel, the next Analysis kernel would be responsible for executing the following operations:

$$\begin{aligned} &A_S(\mathbf{p}_1) \leftarrow A_S(\mathbf{p}_1) + A_S(\mathbf{p}_2) \text{ (atomically)} \\ &A_S(\mathbf{p}_2) \leftarrow 0 \end{aligned} \tag{1}$$

By clearing the $A_S(\mathbf{p}_2)$ position, the algorithm ensures that C_2's size is not summed up more than once, because $\mathbf{p}_2$ may change labels several times, but there is a single Scan pass during which it goes from representative to non-representative and therefore should transfer its size information to its new representative pixel ($\mathbf{p}_1$ in this case). At the end of this modified algorithm, representative pixel positions in A_S contain the size of the respective CCs and non-representative positions contain zeroes. This set of modifications adds a constant cost to the Prelabel kernel and a variable cost to the Analysis kernel, depending on image content, as will be seen in Section V.

It is essential that properties be transferred during the Analysis phase, because the representation of equivalence relations may change during this phase, but the relations themselves remain stable. In contrast, if we did it during the Scan phase, inconsistencies might result from race conditions acting on Equation 1.

B. Representing additional properties

The calculation of centroids and co-variance matrices requires more involved operations than shown in Equation 1. Assume that arrays A_x, A_y are used for accumulating the x, y centroid components and consider, for instance, the same scenario with components C_1, C_2 described in Section III-A. In this case, updating A_x would require

$$\begin{aligned} A_x(\mathbf{p}_1) &\leftarrow \frac{A_S(\mathbf{p}_1)A_x(\mathbf{p}_1) + A_S(\mathbf{p}_2)A_x(\mathbf{p}_2)}{A_S(\mathbf{p}_1) + A_S(\mathbf{p}_2)} \text{ (atomically)} \\ A_x(\mathbf{p}_2) &\leftarrow 0 \end{aligned} \tag{2}$$

but the first assignment cannot be completed with a single atomic operation in contemporary GPU architectures. Hence, we choose a simpler representation, in which the A_x, A_y arrays actually store the sum of x, y pixel coordinates, allowing us to update them just as in Equation 1. Similarly, the A_{x^2}, A_{y^2}, A_{xy} arrays are used for storing the sum of squared x coordinates, the sum of squared y coordinates and the sum of xy coordinate products, respectively, needed for the computation of co-variance matrices. The complete initialization operations for a provisional pixel $\mathbf{p} = (x, y)$ are

$$\begin{array}{llll} A_S(x,y) & \leftarrow 1 & A_{x^2}(x,y) & \leftarrow x^2 \\ A_x(x,y) & \leftarrow x & A_{y^2}(x,y) & \leftarrow y^2 \\ A_y(x,y) & \leftarrow y & A_{xy}(x,y) & \leftarrow xy \end{array} \tag{3}$$

It is possible, at the end of the algorithm, to obtain the centroid of an arbitrary connected component, with representative pixel $\mathbf{p}$, as

$$\frac{1}{A_S(\mathbf{p})} \begin{bmatrix} A_x(\mathbf{p}) \\ A_y(\mathbf{p}) \end{bmatrix}, \tag{4}$$

and its actual co-variance matrix as

$$\begin{bmatrix} \frac{A_{x^2}(\mathbf{p})}{A_S(\mathbf{p})} - \left(\frac{A_x(\mathbf{p})}{A_S(\mathbf{p})}\right)^2 & \frac{A_{xy}(\mathbf{p})}{A_S(\mathbf{p})} - \frac{A_x(\mathbf{p})A_y(\mathbf{p})}{A_S(\mathbf{p})^2} \\ \frac{A_{xy}(\mathbf{p})}{A_S(\mathbf{p})} - \frac{A_x(\mathbf{p})A_y(\mathbf{p})}{A_S(\mathbf{p})^2} & \frac{A_{y^2}(\mathbf{p})}{A_S(\mathbf{p})} - \left(\frac{A_y(\mathbf{p})}{A_S(\mathbf{p})}\right)^2 \end{bmatrix} \tag{5}$$

This procedure allows for a simple and efficient extension of Kalentev et al.'s area computation method for computing centroids and co-variance matrices. As a downside, it requires a considerable number of bits to be represented. Table I compares LE's memory consumption across a range of increasingly higher resolutions with and without the computation of any properties. Refer to Appendix B for explicit expressions for the number of bits needed for each property type, as functions of image resolution. In order to support the range of resolutions spanned by 32-bit unsigned labels, which include the resolutions used in our tests, we employed 32-bit unsigned integers for A_S and 64-bit unsigned integers for the remaining properties, according to these expressions. Because the memory costs grow considerably faster when properties are computed this way, we will discuss another scheme based on contiguous relabeling to reduce memory usage, described in Section III-D.

TABLE I: Memory consumption of label and property arrays (size, centroid and co-variance matrix) in LE.

Resolution.	1K^2	4K^2	16K^2	64K^2
Labels	4 MB	64 MB	1 GB	16 GB
Properties	48 MB	768 MB	12 GB	192 GB

Finally, we consider the issue of memory layout for property computation. There are two main options for storing pixel-wise properties: *structure of arrays* (SOA) where each property has its own device memory buffer; or *array of structures* (AOS) where all properties of a single pixel are interleaved. This choice may significantly impact access performance [22]. In our case, since all kernels manipulating properties must first inspect A_S before updating the other A_* arrays in parallel, the SOA structure provides superior performance by ensuring a coalesced access pattern [21].

C. Early Property Computation in TM

During the Prelabel kernel in LE, provisional labels are assigned to pixels, and if Early property calculation is to be employed, provisional properties are also initialized according to Equation 3. Since property arrays S_* are stored in global memory and global memory operations are guaranteed to be synchronized between kernel launches by the CUDA driver [21], this ensures that property update operations (Equation 1) may be applied later, during the Analysis kernel, without further concerns.

However, since TM's first step (Local Merging) performs both initialization and merging, we need to make sure that the provisional property arrays S_* are correctly initialized before any merging takes place. We have investigated three alternatives:

1) Initialize all S_* arrays with provisional properties (Equation 3) during Local Merge, and delay any property operations (Equation 1) until the first Global Merge kernel execution.
2) Use a memory setting function (such as CUDA's `cudaMemset()`) to initialize the S_* global memory arrays to zero prior to Local Merge, allowing regular property update operations to be carried on during Local Merge itself.
3) Use shared memory (which can be synchronized for threads in the same block) for storing provisional properties during Local Merge, apply Equation 1 on shared

memory instead of global memory, then store the results into global memory S_* arrays at the end of the kernel.

The first and second alternatives give very similar performance. In fact, we only observed a $1-3\%$ difference in pro of the second one. The third alternative seemed promising due to the superior performance of atomic operations in shared memory. However, we noted that the high shared memory requirements of storing all property values for all pixels considerably degrades the GPU's L1 cache performance, incurring in a 14% slowdown just from allocating the necessary space.

The remaining adjustments to TM boil down to modifying the intermediate and final Path Compression kernels so that, at the end of these kernels, properties are merged just like in LE's Analysis.

D. Late Property Computation

If we assume the number of connected components expected in a frame is low compared to the number of pixels, then allocating one property slot of each type per pixel in advance becomes very inefficient. However, if properties accumulation are delayed until CCL completion, it becomes possible to count the number of CCs, then allocate exactly one slot per representative pixel and have all foreground pixels sum their property contributions into the slots associated with their known representative labels. In summary, this is what we call *Late Property Computation*.

However, this requires mapping the sparse set of representative labels $\{l_1, \ldots, l_n\}$ produced by a CCL algorithm into a set of consecutive labels $\{0, \ldots, n-1\}$ that can be used as contiguous indices into property arrays A_* of length n. As noted by [33], consecutive labels may be obtained with an *Inclusive Scan* operation [7]. We refer to the whole process of properly initializing the required memory buffers, applying Inclusive Scan, and relabeling all pixels as *Sequential Relabeling*. Since this topic was not extensively covered in known references, for completeness, we describe it in more details in Appendix C.

Only $48n$ bytes are required for storing the properties listed in Section III-B in this case, in contrast to $48wh$ bytes in the Early Computation case ($w \times h$ being the image dimensions). An secondary benefit of compacting the property arrays is accelerating GPU-to-CPU property transfers. Also, any CPU or GPU post-processing operation that took connected component properties as input would need to inspect only n instead of wh property slots. Indeed, these additional benefits of Sequential Relabeling may also be combined with Early Computation, allowing to compact property slots after the standard algorithm finishes, yet Late Computation is still distinctive with regards to preventing massive memory allocation in advance.

IV. Evaluation methodology

The GPU CCL algorithms cited so far are data-dependent in different ways: some may require multiple iterations to complete, and checking is done on host side [8], [10], [11], [28], [40]; others encapsulate all iterations inside kernels and need a fixed number of passes [1], [15], [23], [33], [39]. When evaluating these methods, authors appeal to different datasets,

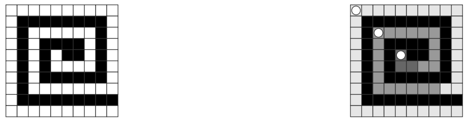

(a) Binary spiral (b) Spiral at 1st LE iter.

Fig. 1: A binary input Spiral Pattern (a) and the label array (b) after the first LE iteration. Black represents background pixels, and each shade of gray represents a different label. Representative pixels are indicated by a circle in the center.

usually for comparing against CPU methods, although they sometimes compare to other GPU methods as well. Although GPU algorithms are usually proven faster than their CPU counterparts, this is usually verified by comparing their average processing time on image datasets, which does not help in clarifying how image characteristics (such as number of objects, their sizes, shapes and locations, for instance) contribute to the total processing time.

A commonly employed test pattern for algorithms that perform host-side iterations is the Spiral Pattern, illustrated in Figure 1. Nevertheless, we noticed some limitations presented by the Spiral Pattern: (1) it produces very long equivalence chains, causing Analysis overhead in LE; (2) the number of iterations for LE is always two (Figure 1); (3) in many tracking scenarios, connected components tend to be convex, so it does not represent the typical geometry of connected components.

The first point is confirmed by our experiments in Section V-C, and since this pattern stresses Analysis passes, it makes difficult to derive a relation between processing time and other image parameters, such as percentage of foreground, number of CCs, or number of iterations. In order to overcome these limitations, we propose *Ridge Patterns*: a family of synthetic connected component shapes that can be easily replicated and parametrized in order to control, among other image characteristics, the number of LE iterations.

A. Introduction to Ridge Patterns

Spiral patterns are an unrealistic object shape model for object tracking/segmentation scenarios, although they can be used as a difficult test case for GPU CCL algorithms, as discussed in Section V. In contrast, one of the simplest conceivable rasterized shapes that could be used as an artificial object model is that of a quadrangular shape, such as a square. However, as Figure 2 illustrates, LE's preference for labels with lesser raster values causes convex shapes to be labeled in at most 2 iterations, contrary to the general case, in which an arbitrary number of iterations may be needed (the frames in the video datasets studied in Section V-E, for instance, require from 1 to 6 iterations). Controlling the number of LE iterations, while keeping the artificial object model as simple as possible, motivated us to develop *Ridge Patterns*.

Informally, Ridge Patterns consist of trapezoidal shapes topped with a *crest*, e.g. an up-and-down pattern of pixels that delays the propagation of minimal labels, forcing LE into the desired number of iterations. Crest size and shape are determined by the pattern's *level*, a parameter that corresponds to the exact number of iterations the LE algorithm takes to

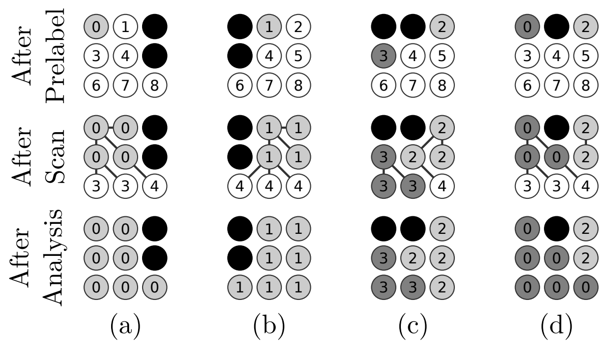


Fig. 2: Output of Prelabel, Scan, and Analysis kernels on 4 different 3×3 image patches (a-d), on first LE iteration. Black indicates background, and distinct shades of gray indicate final labels. Intermediate links formed by the Scan kernel are indicate by solid lines joining adjacent pixels. Although cases (a-b) are completely labeled after a single iteration, cases (c-d) are left with two sub-components each, and require an additional iteration.

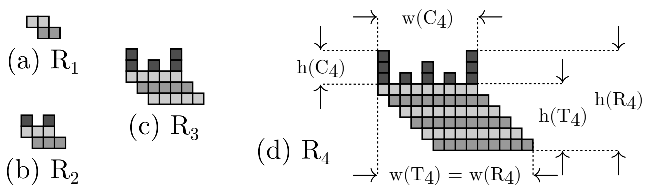


Fig. 3: Examples of minimum Ridge patterns R_l, for levels $1 \leq l \leq 4$. Crest pixels are colored with a darker gray hue, and pixel rows in the trapezoidal base are colored with alternating lighter hues.

complete on the pattern. Basic crest shapes for levels 1 and 2 can be deduced from the cases presented in Figure 2(c-d), and higher levels may be achieved by recursively replicating that basic structure. Figure 3 illustrates ridge patterns for levels 1-4, and Figure 4 provides a good intuitive understanding of the recursive nature of crest design, by showing the partial labeling through iterations 1-4.

In the next subsection, we will present a more thorough description of Ridge Patterns that can be used for reproducing them. Note that some of our choices, such as using a trapezoidal instead of a square base, are not strict requirements for the purpose of obtaining simple shapes, or controlling the number of LE iterations. The purpose of trapezoidal shapes is to ensure that sub-components get an even distribution of pixels, an important performance factor when computing properties, as will be discussed further in Section V-C. For instance, if the level 4 pattern shown in Figure 4 had a rectangular base, the number of pixels in the left-most sub-component $C_{\mathbf{p}}$, connected to the root pixel $\mathbf{p}$ throughout the first iteration, would be considerably greater than the other sub-components. This would cause the first Analysis kernel to issue too many atomic sum operations for the same target memory addresses $A_*(\mathbf{p})$, that hold the property sums of $C_{\mathbf{p}}$.

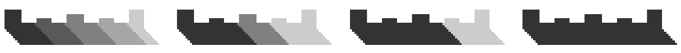


Fig. 4: Consecutive LE iterations applied to a Level 4 Ridge Pattern.

B. Formal description of Ridge Patterns

A Level-1 Ridge Pattern, indicated by R_1, is a 2×2 rasterized trapezoidal shape, as shown in Figure 3(a). A *Level-l Ridge Pattern*, indicated by R_l, for $l \geq 2$, is formed by a trapezoidal base T_l, topped with a *crest* C_l, which consists in a sequence of vertical pixel stripes, or *peaks*. The number of crest peaks is

$$|C_l| = 2^{l-2} + 1 \tag{6}$$

Peaks are enumerated from left to right, and indicated by P_i, $0 \leq i \leq 2^{l-2}$. The x-coordinate of P_i is

$$x(P_i) = 3i \tag{7}$$

and each peak is one pixel-wide, which gives a separation of two pixels between consecutive peaks. Furthermore, the height of P_i, relative to the top of T_l, corresponds to the number of powers of two of the form 2^k, $0 \leq k \leq l-2$, that divide i, plus one, that is

$$h(P_i) = 1 + \sum_{k=0}^{l-2} \left\lfloor \frac{2^k - (i \bmod 2^k)}{2^k} \right\rfloor \tag{8}$$

The crest's width $w(C_l)$, and height (equal to the height of the tallest peak) $h(C_l)$, are therefore given by

$$w(C_l) = (2^{l-2} + 1) + 2(2^{l-2}) = 2^{l-1} + 2^{l-2} + 1 \tag{9}$$
$$h(C_l) = l \tag{10}$$

Appendix A proves that a crest shape defined like this causes the desired number of iterations.

The trapezoidal base T_l, as previously explained, is a secondary aspect of the pattern's definition. The 45° inclination *wrt.* raster coordinates is meant to produce an even distribution of atomic operations during the first Analysis kernel, and its height is defined so that the pattern's total area is approximately that of a square with size equal to the base size. Hence, its width $w(T_l)$ and height $h(T_l)$ are given by

$$w(T_l) = 2w(C_l) - h(C_l) - 1 = 2^l + 2^{l-1} + 2 - l \tag{11}$$
$$h(T_l) = w(C_l) - h(C_l) = 2^{l-1} + 2^{l-2} + 1 - l \tag{12}$$

C. Using Ridge Patterns for image generation

In this section, we introduce six families of synthetic images composed of Ridge Pattern combinations that we use for evaluating GPU CCL algorithms. Each family constitutes a group of test images generated by some free parameter, and some of the families are themselves characterized by parameters. We assume images sizes have the form $2^n \times 2^n$ pixels.

The starting point for defining these families is to introduce the R_l^k notation: it indicates a R_l pattern in which C_l is horizontally scaled by a factor of 2^k, but vertical dimension is kept and T_l dimensions are recomputed from Equations 11-12. As a result, we obtain

$$w(C_l^k) = 2^{k+l-1} + 2^{k+l-2} + 2^k \tag{13}$$

$$h(C_l^k) = l \tag{14}$$

$$w(T_l^k) = 2^{k+l} + 2^{k+l-1} + 2^{k+1} - l \tag{15}$$

$$h(T_l^k) = 2^{k+l-1} + 2^{k+l-2} + 2^k - l \tag{16}$$

If we are to place multiple scaled patterns side-by-side, we also need to add a two pixel horizontal/vertical separation between them. The total horizontal space occupied by such patterns is

$$2 + w(T_l^k) \leq 2^{k+l+1} \tag{17}$$

and since $w(T_l^k) > h(T_l^k)$, the equation above places a lower bound on the maximum scale factor 2^k that may be applied to the base R_l pattern, in order to fit multiple R_l^k patterns side-by-side in a $2^n \times 2^n$ image. From this point, the image families are defined as follows:

1) $\mathrm{Leveling}$*:* Each image is comprised of a single level l pattern, scaled as much as possible for the image size, that is, R_l^{n-l-1}. The family is spanned by the parameter l, with $1 \leq l \leq n-1$. This family tests the sensitivity of the LE algorithm to the number of iterations.

2) $\mathrm{Scaling}(l)$*:* Given the R_l^{n-l-1} (the largest level-l pattern for the given resolution), each image is generated by uniformly scaling R_l^{n-l-1} by a factor λ in the $[0,1]$ interval, which is always possible, since this Ridge Pattern contains no crest. The family is spanned by all $\lambda \in [0,1]$, which includes an empty image ($\lambda = 0$) and the unmodified R_1^{n-l-1} pattern ($\lambda = 1$). It is useful for testing whether algorithms are affected by a large connected component.

3) $\mathrm{Subdivision}(l)$*:* For a given l, the first image contains the R_l^{n-k-1} pattern. The second image is obtained by replacing this pattern with four R_l^{k-n-2} patterns, which keeps the number of pixels approximately constant, but multiplies the number of connected components by 4. Further family members are obtained by recursively applying this replacement rule, a process that exponentially increases the number of connected components, and is limited to $n-l-1$ times. This is a way of determining if algorithms are affected by the number of connected components.

4) $\mathrm{RandomInstancing}(l, s)$*:* Given the $\mathrm{Subdivision}(l)$ family, each image in this family is generated by applying s subdivisions (that is, picking the s-th family member), then erasing each pattern with probability $1-p$, $p \in [0,1]$. The family is generated varying the values of p, but since random numbers are generated for each pattern instance, members are probabilistic, unlike previous families. This family constitutes another way of testing whether algorithms are sensitive to the number of connected components, but in a way that foreground varies with the number of components, unlike $\mathrm{Subdivision}$, and with great flexibility in CC placement. Note that s is restricted by $0 \leq s \leq n-l-1$.

5) $\mathrm{RandomMerging}(l, s)$*:* Members of this family are also probabilistically generated from the s-th members of $\mathrm{Subdivision}(l)$, as in the previous family, but here the parameter p, which characterizes family members, controls the probability of two adjacent patterns being merged into a single component. Patterns are merged by simply adding a horizontal or vertical line of foreground pixels to connect them. The number of foreground pixels in this family is constant, as in $\mathrm{Subdivision}(l)$, but the number of connected components and their shapes can vary greatly.

TABLE II: Thread block layouts (B_x, B_y, B_z) for LE/TM kernels. Global Merge's layout depends on merge level k, and is limited by the maximum number of threads per block, 1024.

Kernels	B_x	B_y	B_z
LE Prelabel	8	8	1
LE Scan/Analysis, TM Path Comp.	16	8	1
TM Local merge	16	16	1
TM k-th Global Merge	4	4	$4^{\min\{3,k+1\}}$

V. Experiments

This section is divided as follows: V-A details our hardware and software conditions; V-B discusses practical considerations on LE implementation, backed up by experiments; V-C evaluates our implementations of the LE and TM algorithms (briefly introduced in Section II-A) combined with the two property computation strategies presented in Section III; V-E analyzes the effectiveness of our methodology in predicting processing times in video sequences.

A. Hardware and software details

We implemented our versions of the LE and TM algorithms for binary images, only. Labels were represented with 4-byte unsigned integers, but one label was reserved for background, resulting in a maximum xy-resolution product of $2^{32}-1$ pixels. Our software was developed using the CUDA 9.1 [21] platform and is available for download under version 3.0 of the GNU General Public License at https://gitlab.com/lcg/cuccl, where more information on how to modify, compile and use our code is also provided. The Sequential Relabeling phase uses the Inclusive Scan implementation from the Thrust open source library (https://thrust.github.io/), that is distributed along with the CUDA toolkit.

We performed our experiments with resolutions of the form $2^n \times 2^n$, $10 \leq n \leq 14$, and observed a series of behaviors that are consistent across these resolutions. However, the memory requirements of the Early Computation strategy limited its use to 8192^2 ($n = 13$) pixels, with our GPU. All subsequent discussions, unless otherwise mentioned, are based on this specific image size.

The hardware and software specifications of the test environment are: Intel Core i7 7700K (8 cores, 4.2 GHz), 32GB Ram (1.6 GHz), NVIDIA GTX 1080 8GB memory, Ubuntu 16.04.02, NVIDIA Driver 390.30, CUDA SDK 9.1, Gnu C/C++ Compiler 5.4.0, Thrust library 1.9.01.

We do not discuss our kernels' sensitivity to runtime parameters such as thread block and grid dimensions, although these parameters can significantly affect performance in CUDA programs. However, various configurations were tested and the best ones, shown in Table II, were picked for the experiments. Note that, with these settings, at 8192^2 resolution, the TM algorithm needs 5 Global Merge levels to complete.

B. Performance considerations on LE implementation

Label Equivalence implementations [8], [10], [11], [40] sometimes disagree on details such as usage of atomic operations, GPU memory allocation, and distribution of operations among kernels, to name a few. We implemented LE borrowing ideas from both HLE and KLE: as in HLE, we used texture memory instead of global memory for storing current label equivalences during the scan phase, due to its superior performance; like KLE however, we stored equivalence chains in the output label array L, instead of using an auxiliary equivalence table T, reducing memory requirements, and rendering the Update kernel at the end of every iteration unnecessary. We still opted for atomic operation-based label replacement as in HLE though, because: (a) we did not observe any substantial performance improvements, as suggested by Kalentev et al. [11] from not using atomic operations, and (b) the number of iterations might slightly vary between different runs if atomic operations were not used, making comparisons to TM more difficult. What follows is a discussion about some implementation alternatives for the Scan kernel. Based on experimental evidence, we will recommend some implementation choices that were not explicitly detailed in the past literature.

1) Changes in first Scan pass: We considered three alternative policies for label selection in the first Scan:

- **F8** (full 8) – searching all 8 neighboring pixels (just as in HLE and KLE).
- **F4** (full 4) – searching only the four pixels with smaller raster values.
- **P4** (pruned 4) – same as F4 but stopping at the first non-background found.

This is motivated by the fact that the first pass follows the initialization of all pixels with provisional labels $x+yw$, hence the smallest label neighbor is necessarily the one with smallest raster index. Figure 5 shows the application of Level 1 Ridge

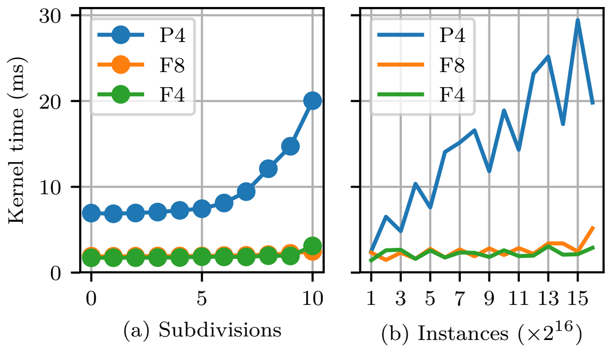


Fig. 5: Performance of the P4, F8 and F4 variations of the first Scan kernel in LE against the $\mathrm{Subdivision}(1)$ and $\mathrm{RandomInstancing}(1, 9)$ image families, at resolution 8192×8192.

Patterns to evaluate these policies, and reveals that F8 and F4 have very similar and stable performance, regardless of the number of connected components and foreground occupancy. In contrast, the P4 variant is visibly affected by these variables, performing much worse than F8 and F4 at all times. This behavior is likely to originate from thread branching in the P4 variant, which tends to become more irregular as more components (and consequently, more component borders) are present. Meanwhile, F4 and F8 cause all threads to execute the same path, issuing consistent and contiguous texture fetches for every thread. The first Scan pass is also necessarily followed by at least one Analysis pass, so it is not necessary to employ a global flag to mark changes in the label array L. A possible further optimization would be implementing the Prelabel and first Scan operations as a single kernel, saving a pair of read-write operations and the overhead of a kernel launch. However, we kept the Prelabel operation separate because its duration depends only on image resolution, as will be seen in Section V. In the remainder of this paper, all tests used the F4 selection policy, and do not use a global flag for marking changes in L.

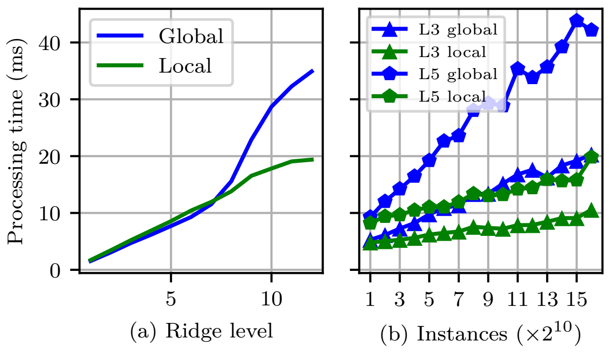


Fig. 6: Total time spent in LE Scan using global and shared memory approaches for taking note of equivalence changes: (a) was tested with the $\mathrm{Leveling}$ family, and (b) with the $\mathrm{RandomInstancing}(1, 5)$ family, for $l = 3$ and $l = 5$, indicated as $L = 3$ and $L = 5$, respectively.

2) Changes in later scan passes: Neither Hawick et al. nor Kalentev et al. explicitly specified if the global memory flag used to mark changes in L should be allocated in *page-locked memory*, but we assume it to be the case in both works, since this memory type is expected to provide faster access [21] to the flag's value in the host code that checks for additional iterations. However, the overhead of several threads writing to a single global memory address when there are label changes is considerable, so we propose using shared memory to reduce this cost as follows:

1) A shared memory flag is allocated for each thread block and cleared at the beginning of kernel execution.
2) A block-level synchronization command (for example, `__syncthreads()` in CUDA) is issued.
3) Threads carry out a regular Scan operation, setting the shared flag if they issue an atomic operation, regardless of its result.
4) Threads are again synchronized block-wise, and if the shared flag is set, the global flag is set by the first thread in the block.

Figure 6 (a) shows that, for a single large component, the global flag approach is only slower after 8 subdivisions (approximately 260000 components). However, Figure 6 (b) also reveals that the global approach is quickly surpassed by the local one as the number of connected components increases. In particular, the discrepancy between the global and local versions with level 5 patterns is much greater than with level 3 patterns. During the rest of this paper, the local flag version with the F8 neighbor selection policy should be assumed for later Scan passes.

C. Evaluating LE and TM with synthetic images

In this section, we discuss the performance of our LE and TM implementations when running against the members of the five synthetic image families described in Section IV, beyond the classical Spiral Pattern images, which constitute a sixth family. Three variations of each algorithm were studied: Ordinary LE/TM, which involve only computing the CCL, but no CC properties, and the versions that compute CC properties using Late Computation or Early Computation (described in Section III), dubbed *Late LE/TM* and *Early LE/TM*, respectively. Although the Inclusive Scan operation is only strictly necessary for the Late algorithms, we have also applied in Ordinary and Early algorithms, since its effects (obtaining sequential labels and speeding up GPU-to-CPU transfers) are advantageous to all algorithm variants.

The plots shown in Figures 7-8 show some performance plots for different image families. For each family, the top row shows the stacked processing times of all kernels that constitute Ordinary LE/TM as functions of the family's free parameter. The two additional rows have a similar layout, but show only the overhead added by Late/Early LE/TM to each kernel. Note that Late LE/TM are implemented by chaining Sequential Relabeling (which is already performed for all algorithms in our tests) and Properties Sum kernels after Ordinary LE/TM, so the only overhead in Late LE/TM is due to Property Accumulation.

Figure 7 presents kernel execution times for the $\mathrm{Leveling}$, $\mathrm{Subdivision}(3)$, $\mathrm{RandomInstancing}(3,5)$, and $\mathrm{RandomMerging}(3,5)$ Ridge Pattern image families. We picked level $l = 3$ for non-$\mathrm{Leveling}$ families because it is the most frequent number of LE iterations observed in the datasets evaluated in Section V-E. Similarly, we picked $s = 5$ subdivisions for the random families because higher values result in more than 1024 connected components per image, an amount considerably larger than the average in the datasets.

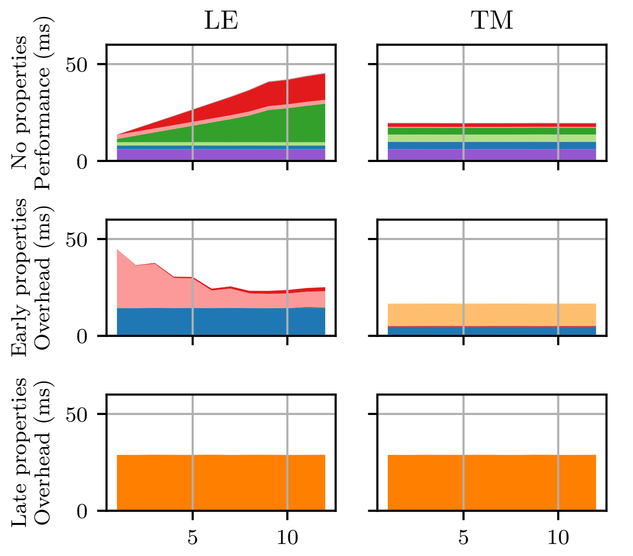
Figure 7a shows that Ordinary LE is very sensitive to the number of iterations, since the processing time increases over a 100% from 1 to 10 iterations. Nevertheless, in the range of most likely iterations in the datasets, 3 to 5, Ordinary LE's performance is comparable to Ordinary TM's, which is not affected by Ridge Pattern level, as expected. However, in Early LE more iterations tend to cause a positive impact on the first Analysis kernel's overhead, since property sums are carried by atomic operations in a greater number of addresses. As a result, total Early LE processing times do not vary much for different levels, although it is still considerably slower than Early TM when there is a single CC in the scene.

Figure 7b, reveals that Ordinary LE/TM and Early LE/TM are mostly invariant to the number of connected components when foreground occupancy is kept constant. Despite the number of CCs being exponentially increased, we only see a noticeable increase in Ordinary processing time starting at 6 subdivisions, which corresponds to 4096 CCs. Such an increase could be explained by the fact that Ridge Pattern instances become smaller than the thread size, causing considerable thread divergence in LE Scan and TM Local Merge. The case with $s = x = 0$ subdivisions is identical to $x = l = 3$ in the $\mathrm{Leveling}$ family and causes Early LE to take longer in the First Analysis kernel, but greater values of $s = x$ diminish this effect, by spreading atomic operations. For the same reason, Late LE/TM usually performs better on scenes with a greater number of components, as the Late overhead plots show.

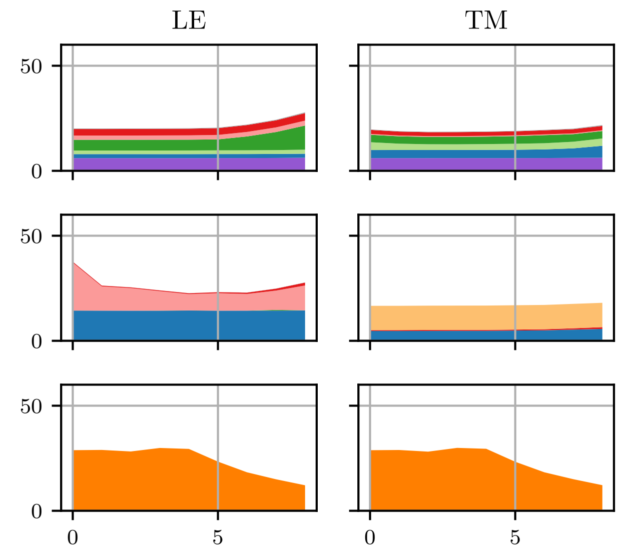
Figure 7c shows the impact of increasing the number of CCs and the total foreground occupancy, by randomly distributing CCs in the test images. Considering that each value of the instance probability p is tested 100 times, the x-axis also corresponds, in the average, to the total foreground occupancy (which is proportional to the number of CCs). That said, it is clear that LE/TM processing times are linearly influenced by the number CCs, when more CCs imply greater foreground occupancy (unlike Figure 7b). One can observe modest slopes in the Ordinary cases, more evident linear behavior in Early cases, and a considerably steep linear behavior in Late cases. When total processing times are grouped by foreground occupancy, we see maximum standard deviation of 6% for LE and 32% for TM across all experiment runs. This shows that geographic distribution of CCs is not a relevant performance factor for LE, but extremely relevant for TM.

Figure 7d is harder to interpret, because the merging probability may create complex shapes when connecting adjacent Ridge Patterns. In particular, when the merging probability is 0, no instances are connected, obtaining the same performance as $s = x = 5$ in Figure 7b, and when the probability is 1, all pattern instances are merged, resulting in the same performance as $x = l = 3$ in Figure 7a. Increasing the merging probability p above 0, at a first moment, makes larger CCs with more complex shapes more likely to exist. But as p approaches 1, the number of CCs decreases and the tendency is to form a single large component, which possibly explains why the LE processing time peaks are somewhere in between, where more complex shapes are more likely to exist. Note that this causes saturation behavior in Late overhead, which tends to take longer when fewer CCs concentrate foreground pixels. Ordinary/Early TM are also not considerably affected by this test.

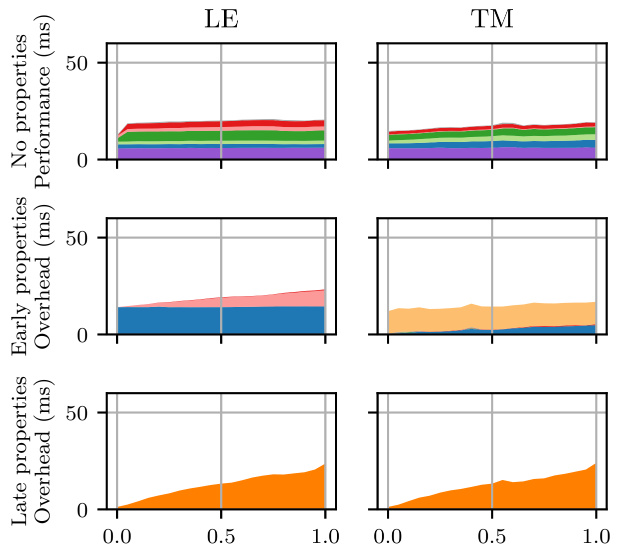
Finally, Figure 8 shows the effect of a single, increasingly large connected component, allowing us to isolate the impact of the total number of foreground pixels and to compare Spiral and Ridge Patterns. Ordinary LE/TM are not considerably affected by a larger Ridge Pattern, but the Spiral Pattern has a strong destabilizing effect on both, being the test family that causes the strongest impact on Ordinary TM, an otherwise stable algorithm in terms of average processing time. The Spiral Pattern is the only test case seen in this work in which the first LE Analysis pass represents such a significant portion of Analysis time that later passes are not even visible in the plot. Meanwhile, Ridge Scaling causes Local Merge time to grow



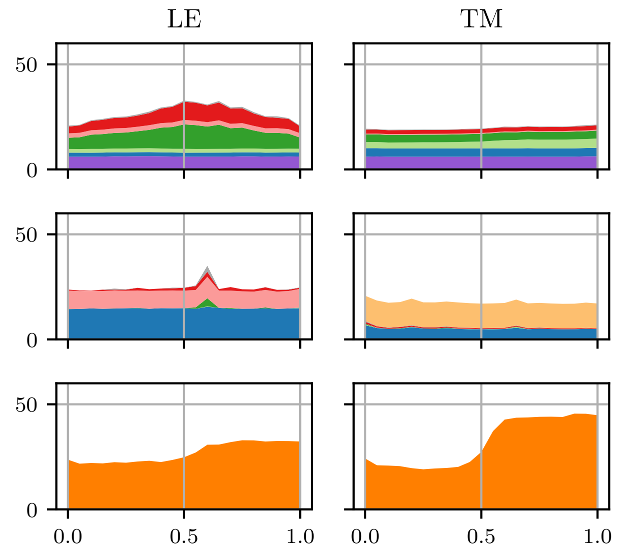
(a) Leveling family. Level of single Ridge Pattern shown along the x-axis.



(b) Subdivision(3) family. Number of subdivisions (base-4 logarithm of number of CCs) shown along the x-axis.



(c) RandomInstancing(3, 5) family. Appearance probability of each of the 1024 Scaled Ridge Patterns of form R_3^9 shown along x-axis.



(d) RandomMerging(3, 5) family. Merging probability between each pair of neighboring patterns of form R_3^9 shown along x-axis.

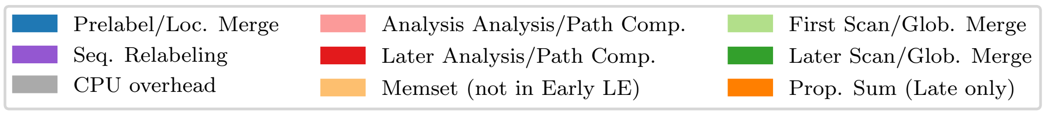
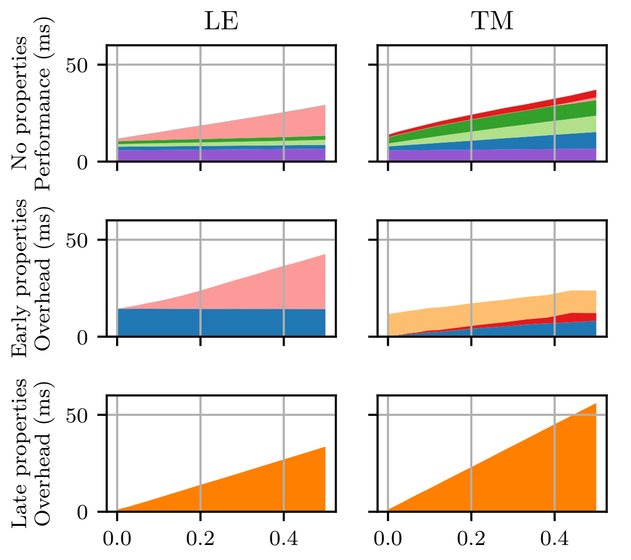
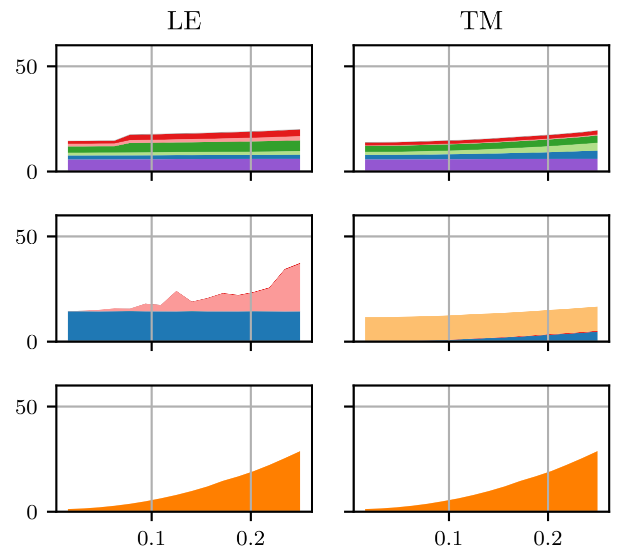


Fig. 7: Stacked kernel processing times in Ordinary LE/TM, and stacked kernel overheads due to property computation in Early LE/TM and Late LE/TM measured on four image families (a)-(d), at 8192×8192 pixels. For each family member, represented by a point in the x-axes of the respective plot groups (a)-(d), kernel processing times and overheads were averaged over 100 executions. Analogous kernels in LE/TM are colored similarly. The vertical order of the colored stripes does not necessarily reflect the order of operations.



(a) Spiral patterns (foreground occupancy percentage varies along x-axis).



(b) RidgeScaling(2) family. Foreground occupancy of a single R_2^{10} pattern varies along x-axis.

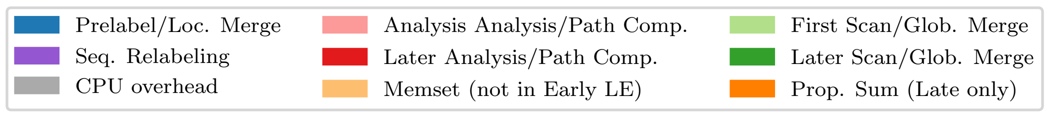


Fig. 8: Stacked kernel processing times in Ordinary LE/TM (top row), and stacked kernel overheads due to property computation in Early LE/TM (middle row) and Late LE/TM (bottom row), at 8192×8192 pixels, measured on: (a) Spiral Patterns, and (b) the $\mathrm{Scaling}_{8192}$ image family. A level 2 pattern was used in (b) to match the number of LE iterations required by Spiral Patterns. For each family member, represented by a point in the x-axes of the respective plot groups (a)-(b), kernel processing times and overheads were averaged over 100 executions. Analogous kernels in LE/TM are colored similarly. The vertical order of the colored stripes does not necessarily reflect the order of operations.

super-linearly in TM. These behaviors may be explained by the geometric differences between the patterns, as the equivalence chains in a spiral patterns are essentially linear, and lack locality of memory access. Late/Early property computation strategies seem less affected by Ridge Patterns than by Spiral Patterns, but it should be noted that the formulation of Ridge Patterns as trapezoidal shapes restricts maximum foreground occupancy to about 25%, whereas Spiral Patterns may reach up to 50%. Although the exceptional behavior of Spiral Patterns will render them less useful for processing time estimation in Section V-E, they may be considered good candidates for worst case estimates.

D. Final notes on Late/Early overhead

By studying Figures 7-8 we see that, in most cases, the processing time difference between Ordinary TM and Early TM is obtained by multiplying each kernel by some scale factor and adding a constant cost relative to memory setting. Early LE, in the other hand, usually involves a noticeable change (for worse) in Analysis behavior, making its average Early to Ordinary performance ratio worse than TM's. This effect is regulated by the size of CCs: worse performance is seen when a considerable number of pixels is distributed among fewer components (Figures 7b and 8a). For Late algorithms, the critical performance factors are the number of CCs and their size (Figures 7b-7d, and 8a). Nonetheless, in the average, Late LE/TM are usually faster than their Early counterparts, though associated with greater variance. Table III shows the mean processing time and related standard deviation in each pattern family for Ordinary, Early, and Late LE/TM, for resolution 8192×8192. The Spiral Pattern corresponds to the largest deviation, in all tables.

The Sequential Relabeling operation is fundamental to making Late Computation's average performance better than Early's, because it allows to execute a fast Memset operation for property arrays A_* initialization on fewer indices. Consequently, Memset time is hardly visible in all Late plots, while it represents a considerable fixed cost (12 ms in average) in Early TM, which always has to initialize 8192^2 elements in each of the A_* property arrays. Nonetheless, this operation is also important for Ordinary and Early algorithms, since full property arrays at this resolution (8192^2 elements per property), with the chosen numeric representation, occupy 2.5 GB of memory, which takes around 25 ms in a GPU-to-CPU transfer.

Late Computation's final merit is making it possible to apply these algorithms on higher resolutions. At a resolution

TABLE III: Mean processing time and relative standard deviation of Ordinary/Late/Early LE/TM on the synthetic image families at 8192×8192 pixels. The total number of images in each family is shown on the rightmost column.

Family	Ord. LE	Ord. TM	Early LE	Early TM	Late LE	Late TM	Images
Leveling	30.9 ms $\pm$ 34%	19.5 ms $\pm$ 1%	60.0 ms $\pm$ 9%	36.1 ms $\pm$ 1%	59.7 ms $\pm$ 18%	48.3 ms $\pm$ 1%	1200
RandomInstancing	19.3 ms $\pm$ 9%	17.0 ms $\pm$ 9%	38.0 ms $\pm$ 10%	31.6 ms $\pm$ 9%	32.1 ms $\pm$ 23%	29.6 ms $\pm$ 25%	2100
RandomMerging	26.3 ms $\pm$ 16%	19.5 ms $\pm$ 4%	50.5 ms $\pm$ 11%	37.2 ms $\pm$ 5%	53.3 ms $\pm$ 13%	51.4 ms $\pm$ 24%	2100
Scaling	17.5 ms $\pm$ 11%	15.8 ms $\pm$ 11%	38.8 ms $\pm$ 21%	29.3 ms $\pm$ 12%	28.8 ms $\pm$ 36%	27.0 ms $\pm$ 39%	1600
Spiral	17.9 ms $\pm$ 30%	22.8 ms $\pm$ 32%	41.6 ms $\pm$ 35%	39.4 ms $\pm$ 28%	30.7 ms $\pm$ 51%	44.0 ms $\pm$ 56%	900
Subdivision	21.5 ms $\pm$ 12%	19.2 ms $\pm$ 5%	47.5 ms $\pm$ 11%	36.2 ms $\pm$ 4%	45.3 ms $\pm$ 10%	43.0 ms $\pm$ 14%	1600
Total	19.7 ms $\pm$ 31%	17.5 ms $\pm$ 20%	41.3 ms $\pm$ 23%	32.6 ms $\pm$ 18%	35.7 ms $\pm$ 39%	34.4 ms $\pm$ 39%	9500

TABLE IV: Mean processing time and relative standard deviation of Ordinary/Late LE/TM on the synthetic image families at 16 K $\times$ 16 K pixels. The total number of images in all families is shown on the rightmost column.

	LE	TM	Images
Ordinary	76 ms $\pm$ 31%	68 ms $\pm$ 18%	16120
Late	145 ms $\pm$ 44%	140 ms $\pm$ 41%	16120

of 16 K $\times$ 16 K pixels, for instance, memory requirements would be 1 GB for LE/TM label arrays, and 10 GB for Early Computation property arrays, making it impossible to run Early LE/TM on most end-consumer graphics cards, such as the one used in our tests. Even if fewer or smaller bit-depth properties were being computed, the space requirements might still be a limiting factor if other parts of a processing pipeline were to be carried out in GPU memory. We applied the same battery of artificial tests previously discussed on Ordinary LE/TM and Late LE/TM, at 16 $\mathrm{K} \times 16$ k resolution, obtaining the results shown in Table IV.

The sensitivity of the Properties Sum kernel due to increasing foreground occupancy remains an issue to be observed in Late computation performance (it represents approximately 50% of Late processing time at 16 K^2 pixels), but considering the videos evaluated in Section V-E, it is possible to assume that the [0%, 25%] occupancy range explorable with Ridge Patterns is representative of real world scenarios. In this sense, Figures 7 and 8 and Tables III-IV already present us with the worst cases of Late computation.

E. Evaluation on video datasets

In order to test our implementations in real world data, we evaluated their performance against three image/video datasets [2], [16], [27]. These datasets cover a range of different image processing/understanding problems, like object segmentation [2], [16], [27], object classification [2], and tracking [16], [27]. All of them feature ground-truth foreground segmentation of a wide range of objects (persons, vehicles, animals, etc.), and some have been used in public algorithm performance contests, like the *2018 DAVIS Challenge on Video Object Segmentation* [27], and the *The PASCAL Visual Object Classes Challenge* [2].

All of these datasets have multiple incremental releases, but we used only the largest and most recent versions released up to this date. In the Pascal-VOC [2] object detection and classification dataset, each scene was frequently associated with multiple object foreground annotations; in these cases, we generated a single binary image comprised of the union of all scene objects' foreground masks. As in the original paper [33], our TM implementation is restricted to resolutions that are powers of two, so we scaled all frames up to the target resolutions using nearest neighbor sampling (to prevent losing border shapes that impact LE iterations) for all tests. DAVIS 2017 [27] videos' resolution is 1600×900 pixels, while all other datasets have varying, and considerably smaller (usually less than 1024×1024) resolutions. Table V summarizes the main characteristics of each dataset, considering the way we used them Ridge patterns are limited to a foreground occupancy of about 25% of the image pixels, so unless explicitly mentioned, the following results are based on dataset images with a foreground occupancy below 30%, which correspond to exactly 8165 images, or approximately 80% of their contents. For brevity, synthetic image families are indicated by single letters in tables, in the following manner: *L* for Leveling, *R* for RandomInstancing, *M* for RandomMerging, *K* for Scaling, *P* for Spiral, and *S* for Subdivision.

TABLE V: Main characteristics of the latest versions of the image/video datasets used.

Dataset	DAVIS	Pascal VOC	Segtrack
Reference	[27]	[2]	[16]
Year	2017	2012	v2 (2013)
Images	6208	2913	1067

Figure 9 plots the processing time distribution curves for the datasets and for the set of all synthetic images previously evaluated. Table VI presents the corresponding statistics, in numeric form. Although the processing time distributions in the datasets are not perfectly matched by the synthetic images, they produce excellent approximations of the mean and standard deviation.

Finally, we present a simple, yet effective model of performance prediction. We verified that mean processing times and processing time standard deviation could be modeled as functions $f(x)$ of the resolution, where f has the form

$$f(x) = a + bx^c \tag{18}$$

with $a, b, c \in \mathbb{R}$. We estimated these parameters for the mean processing time and for the processing time standard deviation through least-squares fitting on the whole set of synthetic tests performed across all resolutions (a total of 60780 images), obtaining the parameters shown in Tables VII-VIII. The resulting performance model curves are plotted in Figure 10. The error

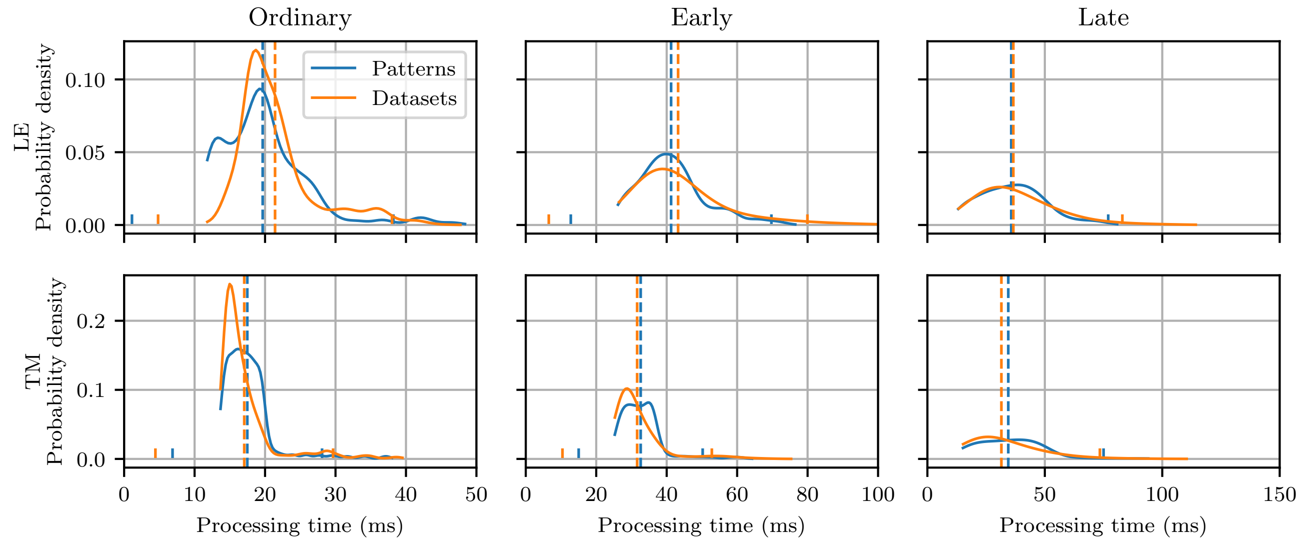


Fig. 9: Processing time distribution in datasets and synthetic test patterns for each algorithm, at 8192×8192 pixels. The distribution curves $f(t)$, were estimated by computing a 100-bin histogram of the processing time t, then convolving the histogram with a Gaussian kernel (with a diameter of 10 bins), and finally normalizing f so that $\int_0^\infty f(t)\ dt = 1$. For each case, the mean processing time μ is indicated by a vertical dashed line, and the points $\mu - 3\sigma$ and $\mu + 3\sigma$ are indicated by shorter dashed lines, with the same color of f.

TABLE VI: Mean processing time μ and processing time standard deviation σ in the datasets and synthetic pattern images are shown in the central columns, for each algorithm. The rightmost columns show how much these quantities, measured on the synthetic patterns, deviate from the datasets, relative to the datasets' standard deviations.

Algorithms	Datasets		Patterns		Patterns error $\left(\times \frac{1}{\sigma_D}\right)$	
	μ_D	σ_D	μ_S	σ_S	$\mu_S - \mu_D$	$\sigma_S - \sigma_D$
Ordinary LE	21.4	5.5	19.7	6.2	-0.32	0.12
Ordinary TM	17.0	4.2	17.5	3.5	0.10	-0.16
Early LE	43.2	12.2	41.3	9.5	-0.16	-0.22
Early TM	31.6	7.1	32.6	5.9	0.14	-0.17
Late LE	36.6	15.5	35.7	13.8	-0.06	-0.11
Late TM	31.5	14.0	34.4	13.5	0.21	-0.03

TABLE VII: Parameters of mean processing time estimation model, as defined by Equation 18. The error for each resolution x was computed as $\frac{f(x)-\mu_D}{\sigma_D}$, where μ_D and σ_D are the datasets' actual mean and standard deviation.

Algorithm		a_μ	b_μ	c_μ	Max. error
Ordinary	LE	75	$1.22e^{-4}$	1.05	0.12
	TM	254	$7.63e^{-5}$	1.07	0.50
Early	LE	170	$2.51e^{-4}$	1.05	0.33
	TM	289	$2.06e^{-4}$	1.05	0.51
Late	LE	93	$3.70e^{-4}$	1.02	0.01
	TM	256	$3.29e^{-4}$	1.02	0.13

TABLE VIII: Parameters of processing time standard deviation estimation model, as defined by Equation 18. The error for each resolution x was computed as $\frac{f(x)-\sigma_D}{\sigma_D}$, where μ_D and σ_D are the datasets' actual mean and standard deviation.

Algorithm		a_σ	b_σ	c_σ	Max. error
Ordinary	LE	34	$1.18e^{-5}$	1.11	0.05
	TM	108	$6.45e^{-6}$	1.11	0.17
Early	LE	24	$5.27e^{-5}$	1.05	0.02
	TM	100	$2.88e^{-5}$	1.06	0.12
Late	LE	64	$8.13e^{-5}$	1.05	0.05
	TM	122	$1.15e^{-4}$	1.03	0.06

magnitudes in these tables confirms that Equation 18 provides a reliable performance model, and suggests that synthetic images may be a viable replacement for actual datasets with respect to performance estimation. Figure 10 also shows that our: (a) Ordinary/Early TM implementation have a substantial long-term advantage over Ordinary/Early LE; (b) Late LE and Late TM share a similar behavior, which was already expected, since Figures 7-8 showed that the Properties Sum kernel dominates execution time as the number of foreground pixels grows. Notice that, although all curves are plotted up to 16K^2 pixels, the memory requirements (over 2.75 GB) for Early Properties Computation (central column plots) impose a practical limit on the maximum image resolution.

VI. Conclusion

In this work, we discussed a set of extensions and evaluation methods for GPU connected component algorithms and tested them on two existing algorithms: the Label Equivalence [8] and the Tile Merging [33] algorithms. Our main contributions are:

- A pair of strategies, described in Section III, for extending these algorithms to compute properties of connected components. In the strategy dubbed *Early Computation*, the connected component size calculation method by Kalentev et al. [11] was extend in order to compute centroids

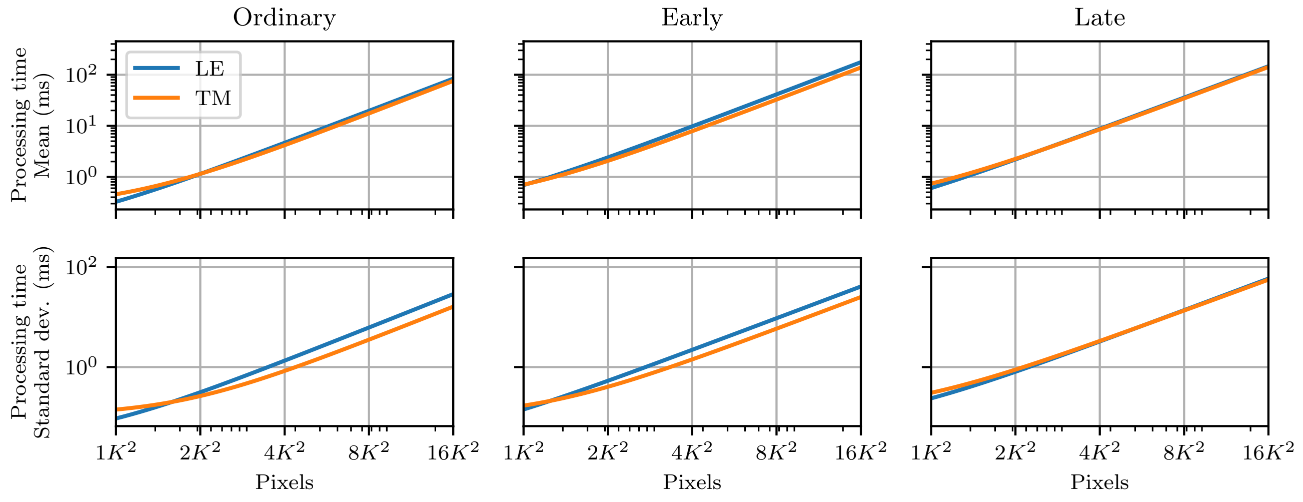


Fig. 10: Processing time mean and standard deviation as functions of resolution. These curves correspond to the model of Equation 18, with the estimated parameters shown in Tables VII-VIII. Both x and y axes are in logarithmic scale.

and coverings matrices using dense property arrays and atomic operations at GPU thread-level. On the other hand, the completely new strategy named *Late Computation* uses *Inclusive Scans* in order to work on smaller property arrays, and as result, improve mean performance, reduces the cost of GPU-to-CPU transfers and makes it possible to apply GPU CCL at greater resolutions impossible to Early Computation, at the expense of increased performance variance.

- A performance evaluation methodology for GPU CCL algorithms described in Section IV that uses synthetic image families built out of Connected Component patches, known as *Ridge Patterns*, to explore data dependency in the tested algorithms (LE and TM). Our methodology extends a test case from the literature (Spiral Pattern), resulting in mean performance close to three natural image datasets, with over 9000 total frames, that were also tested. We also described the fitting of a detailed performance prediction model based on image features, and showed that certain synthetic image combinations improved performance prediction in the evaluated datasets, allowing to predict performance without evaluating a huge number of frames.
- As a minor contribution, we highlighted some implementation choices of the LE algorithm, not explicitly justified in past literature, that allowed it to perform better when compared with TM.

Comparisons between implementations of different algorithms are frequent in the literature. However, if not-so-obvious implementation details are overlooked, such as the ones discussed in Section V-B, two implementations of the same algorithm may considerably differ in their final performance. When comparing our implementations of the LE and TM algorithms, we saw that they kept a similar level of performance, but a more important conclusion than which implementation is faster in average, is how these implementations respond to variations in the input data. With respect to that, our methodology revealed that both algorithms are sensitive to the number, size and shape of connected components, specially when computing properties, but TM is considerably more stable than LE. TM is also more suitable to Early property computation, as the impact of atomic operations is better distributed during its Local Merge phase than during LE's first Analysis phase, resulting in a smaller overhead. Nonetheless, the benefits of Late Computation over Early (faster average performance in most cases, reduced memory usage with compact representation, and reduced transfer times), are very significant.

A. Future directions

In this paper, we addressed two of the most famous and well-documented GPU CCL methods in the literature [8], [33], but applying our methodology to other methods [1], [12], [15], [28], [29], [36], [39], [40] could give a better understanding of their relative strong and weak features.

In the investigation of object properties, we chose an exact integer representation that requires a considerable number of bits, even at lower resolutions. A 32-bit floating-point representation, albeit susceptible to rounding errors due to the unpredictable order of atomic operations, might be able to give satisfactory results in some scenarios, which is left to investigate.

Finally, one limitation of the artificial image families we developed was the non-uniform distribution of image features. It seems plausible that a set of artificial images with uniform feature distributions could give better prediction results and allow even less tests to be used for reliable performance estimation.

Acknowledgments

We thank the Brazilian science and research government support agencies CNPq and CAPES for the first and third authors' grants.

References

[1] P Chen, HL Zhao, C Tao, and HS Sang. Block-run-based connected component labelling algorithm for GPGPU using shared memory. *Electronics letters*, 47(24):1309–1311, 2011.

[2] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, Jun 2010.

[3] David Fernandez, Ignacio Parra, Miguel Angel Sotelo, and Pedro A Revenga. Bounding box accuracy in pedestrian detection for intelligent transportation systems. In *IEEE Industrial Electronics, IECON 2006-32nd Annual Conference on*, pages 3486–3491. IEEE, 2006.

[4] Mohsen Ghanea, Payman Moallem, and Mehdi Momeni. Automatic building extraction in dense urban areas through geoeye multispectral imagery. *International Journal of Remote Sensing*, 35(13):5094–5119, 2014.

[5] Costantino Grana, Daniele Borghesani, and Rita Cucchiara. Connected component labeling techniques on modern architectures. In *Image Analysis and Processing–ICIAP 2009*, pages 816–824. Springer, 2009.

[6] Costantino Grana, Daniele Borghesani, and Rita Cucchiara. Optimized block-based connected components labeling with decision trees. *IEEE Transactions on Image Processing*, 19(6):1596–1609, 2010.

[7] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[8] K. A. Hawick, A. Leist, and D. P. Playne. Parallel Graph Component Labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655–678, December 2010.

[9] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[10] In-Yong Jung and Chang-Sung Jeong. Parallel connected-component labeling algorithm for GPGPU applications. In *2010 International Symposium on Communications and Information Technologies (ISCIT)*, pages 1149–1153, october 2010.

[11] Oleksandr Kalentev, Abha Rai, Stefan Kemnitz, and Ralf Schneider. Connected component labeling on a 2D grid using CUDA. *Journal of Parallel and Distributed Computing*, 71(4):615–620, April 2011.

[12] Young-Min Kang, Sung-Soo Kim, and Gyung-Tae Nam. GPU-based object identification in large-scale images for real-time radar signal analysis. *IJNCAA*, page 140, 2016.

[13] Michael J Klaiber, Donald G Bailey, Yousef O Baroud, and Sven Simon. A resource-efficient hardware architecture for connected component analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(7):1334–1349, 2016.

[14] Pawel Kmiotek and Yassine Ruichek. Representing and tracking of dynamics objects using oriented bounding box and extended kalman filter. In *Intelligent Transportation Systems, 2008. ITSC 2008. 11th International IEEE Conference on*, pages 322–328. IEEE, 2008.

[15] Praveen Kumar, Ayush Singhal, Sanyam Mehta, and Ankush Mittal. Real-time moving object detection algorithm on high-resolution videos using GPUs. *Journal of Real-Time Image Processing*, 11(1):93–109, 2016.

[16] Fuxin Li, Taeyoung Kim, Ahmad Humayun, David Tsai, and James M. Rehg. Video segmentation by tracking many figure-ground segments. In *ICCV*, 2013.

[17] Xi Li, Weiming Hu, Chunhua Shen, Zhongfei Zhang, Anthony Dick, and Anton Van Den Hengel. A survey of appearance models in visual object tracking. *ACM transactions on Intelligent Systems and Technology (TIST)*, 4(4):58, 2013.

[18] Yuhai Li. Fast multi-level connected component labeling for large-scale images. In *Optoelectronics and Microelectronics (ICOM), 2015 International Conference on*, pages 334–337. IEEE, 2015.

[19] Guo Lie, Wang Rong-ben, Jin Li-sheng, Li Lin-hui, and Yang Lu. Algorithm study for pedestrian detection based on monocular vision. In *Vehicular Electronics and Safety, 2006. ICVES 2006. IEEE International Conference on*, pages 83–87. IEEE, 2006.

[20] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15(02):285–329, 2014.

[21] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *CUDA API Reference Manual*, version 8.0 edition, 6 2017.

[22] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *CUDA C Best Practices Guide*, version 8.0 edition, 6 2017.

[23] VMA Oliveira and RA Lotufo. A study on connected components labeling algorithms using GPUs. In *SIBGRAPI*, volume 3, page 4, 2010.

[24] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[25] Fanny Nina Paravecino and David Kaeli. Accelerated connected component labeling using CUDA framework. In *International Conference on Computer Vision and Graphics*, pages 502–509. Springer, 2014.

[26] Chao Peng, Sandip Sahani, and John Rushing. A GPU-accelerated approach for feature tracking in time-varying imagery datasets. *IEEE Transactions on Visualization and Computer Graphics*, 2016.

[27] F. Perazzi, J. Pont-Tuset, B. McWilliams, L. Van Gool, M. Gross, and A. Sorkine-Hornung. A benchmark dataset and evaluation methodology for video object segmentation. In *Computer Vision and Pattern Recognition*, 2016.

[28] Allan Rasmusson, Thomas Sangild Sørensen, and Gernot Ziegler. Connected components labeling on the GPU with generalization to voronoi diagrams and signed distance fields. In *International Symposium on Visual Computing*, pages 206–215. Springer, 2013.

[29] Lubomír Říha and Manohar Mareboyana. GPU accelerated one-pass algorithm for computing minimal rectangles of connected components. In *Applications of Computer Vision (WACV), 2011 IEEE Workshop on*, pages 479–484. IEEE, 2011.

[30] Rupesh Kumar Rout. *A survey on object detection and tracking algorithms*. PhD thesis, 2013.

[31] Kurt Schwenk and Felix Huber. Connected component labeling algorithm for very complex and high-resolution images on an fpga platform. In *SPIE Remote Sensing*, pages 964603–964603. International Society for Optics and Photonics, 2015.

[32] Youngsung Soh, Hadi Ashraf, Yongsuk Hae, and Intaek Kim. A hybrid approach to parallel connected component labeling using CUDA. *International Journal of Signal Processing Systems*, 1:130–135, 12 2013.

[33] O Stava and B Benes. Connected component labeling in CUDA. In Wen-Mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, GPU Computing Series, pages 569–581. Morgan Kaufman, 2010.

[34] Robert Walczyk, Alistair Armitage, and T David Binnie. Comparative study on connected component labeling algorithms for embedded video processing systems. *IPCV*, 10:176, 2010.

[35] Martin Weigel. Connected-component identification and cluster update on graphics processing units. *Physical Review E*, 84(3):036709, 2011.

[36] Henning Wenke, Sascha Kolodzey, and Oliver Vornberger. A work-optimal parallel connected-component labeling algorithm for 2d-image-data using pre-contouring. In *International Workshop on Image Processing*, pages 154–161, 2014.

[37] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. Optimizing two-pass connected-component labeling algorithms. *Pattern Analysis and Applications*, 12(2):117–135, 2009.

[38] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *Acm computing surveys (CSUR)*, 38(4):13, 2006.

[39] Kensaku Yonehara and Kunio Aizawa. A line-based connected component labeling algorithm using GPUs. In *Computing and Networking (CANDAR), 2015 Third International Symposium on*, pages 341–345. IEEE, 2015.

[40] Sergey Zavalishin, Ilia Safonov, Yury Bekhtin, and Ilia Kurilin. Block equivalence algorithm for labeling 2d and 3d images on GPU. *Electronic Imaging*, 2016(2):1–7, 2016.

[41] Chen Zhao, Guodong Duan, and Nanning Zheng. A hardware-efficient method for extracting statistic information of connected component. *Journal of Signal Processing Systems*, pages 1–11, 2016.

Appendix

A. Proof that Ridge crest controls LE iterations

By analogy to Figure 2 (d), C_2 requires 2 LE iterations. Let our induction hypothesis be that $C_l = \{P_0, \dots, P_u\}$ requires l iterations, for some l. Assume that $P_i + 1$ denotes increasing the height of P_i by one pixel. Let us build a crest $C'_l = \{P'_0, \dots, P'_w\}$ from C_l by replicating peaks $\{P_1, \dots, P_u\}$ to the right of P_u, and increasing the height of the first and last resulting peaks by one pixel, *e.g.*

$$C'_l = \{P_0 + 1, P_1, \dots, P_u, P_1, \dots, P_{u-1}, P_u + 1\}$$

The partial crest $\{P'_0, \dots, P'_u\}$ is identical to C_l, except for $P'_0 = P_0 + 1$, so the induction hypothesis implies l iterations are required for propagating the final label (which corresponds to smaller raster index) from P'_0 to P'_u. At the same time, the partial crest $\{P'_u, \ldots, P'_w\}$ is also identical to C_l, with the exception of $P'_w = P_u+1$. Again, by the induction hypothesis, l iterations are required for transmitting a minimal label across the partial crest $\{P'_u, \ldots, P'_w\}$. However, P'_w has smaller raster index than P'_u, so there exists a number v, with $u \leq v < w$, such that the crests $\{P'_u, \ldots, P'_v\}$ and $\{P'_{v+1}, \ldots, P'_w\}$ will have been unified into separate components (rooted at P'_0 and P'_w, respectively) at the end of l iterations. With an additional iteration, crest $\{P'_{v+1}, \ldots, P'_w\}$ unifies to the label of P'_0. Since $w = 2^{l-1}$, it follows that crest C'_l is indeed C_{l+1}. ■

B. Property computation

Table IX presents the bit-depths required for each of property type as a function of image resolution, $w \times h$.

TABLE IX: At resolution $w \times h$, the number of bits per-pixel required by a property array A_* must be calculated considering the largest possible connected component, a $w \times h$ rectangle. The ceiling of the base-2 logarithm of the expressions below give these values for the property arrays we compute, where the expressions for A_y and A_{y^2} are analogous to those for A_x and A_{x^2}, respectively, and have thus been omitted.

A_S	A_x	A_{x^2}	A_{xy}
wh	$h\frac{w(w-1)}{2}$	$h\left(\frac{w^3}{3} + \frac{w^2}{2} + \frac{w}{6}\right)$	$\frac{wh(w-1)(h-1)}{4}$

C. Sequential relabeling

The Inclusive Scan, or *Prefix Sum*, operation [7] is a common parallel programming primitive. It consists in replacing each element in an array with the sum of itself with its predecessors. Formally, given an array R with K elements, the Inclusive Scan S of R is defined by

$$S(r) = \sum_{i=1}^{r} R(i) \tag{19}$$

for all $1 \leq r \leq K$.

As noted by Stava et al. [33], it is possible to use this operation for obtaining consecutive labels after completion of the Tile Merging algorithm. At the end of this algorithm, a pixel $\mathbf{p} = (x, y)$ is representative if, and only if, $L(x, y) = x + yw$, given image dimensions $w \times h$. In order to obtain a sequential relabeling L_S of L, we initialize a wh-element array R with

$$R(x + yw) = \begin{cases} 1, & \text{if } L(x, y) = x + yw \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

for all $1 \leq x \leq w, 1 \leq y \leq h$, and compute its prefix sum S. Then the elements in L_S may be initialized as

$$L_S(x, y) = S(L(x, y)) \tag{21}$$

Furthermore, $S(wh)$ holds the number of distinct labels in L, i.e. connected components. This is also valid for the Label Equivalence algorithm, since representative pixels are characterized the same way as in TM.

Pedro Asad is a DSc. student of The Systems Engineering and Computer Science Program of the Federal University of Rio de Janeiro (UFRJ), Brazil, from which he also received his MSc. degree (2016) on the topic of tracking human gestures using low-cost depth sensors. He lectured introductory and intermediate programming in undergraduate engineering courses as a temporary teacher at UFRJ, in the period of 2015-2017. His current research interests include GPU programming, computer vision and data visualization.

Ricardo Marroquim is an Associate Professor at the Systems Engineering and Computer Science Program of the Federal University of Rio de Janeiro. He received his Doctorate degree from this same institution in 2008. Before joining UFRJ as a professor in 2009, he was an ERCIM post-doc fellow at the Visual Computing Lab in Pisa. After returning from Italy he became one of the leading researchers in Brazil in the area of Digital Heritage. His other main research topics are scientific visualization, real-time rendering, animation, and neuroinformatics.

Andréa L. e L. Souza is a DSc. student of the Computer and Systems Engineering program of the Federal University of Rio de Janeiro, Brazil. She graduated in Mathematics (2008) at Fluminense Federal University and received her MSc. in Applied Mathematics (2011) from the Pontifical Catholic University of Rio de Janeiro. She acted as a mathematics teacher in the public education system during the year of 2011, and as a virtual mathematics tutor at Cecierj Foundation (2011-2013) and at Fluminense Federal University (2012). Her research includes topics in Mathematics Applied to Computing and Computer Graphics. She is currently working on natural interfaces for 3D performance-driven animations controlled by hand gestures using real-time depth sensors.