Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs

Laurent Cabaret, Lionel Lacassagne, Daniel Etiemble

Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs

Laurent Cabaret[13], Lionel Lacassagne[2], Daniel Etiemble[3]
[1]MICS, CentraleSupélec, France [2]LIP6, UPMC, CNRS, France [3]LRI, Univ. Paris-Sud, France
email: laurent.cabaret@centralesupelec.fr, lionel.lacassagne@lip6.fr, email: daniel.etiemble@lri.fr

***Abstract*— Modern computer architectures are mainly composed of multi-core processors and GPUs. Consequently, solely providing a sequential implementation of algorithms or comparing algorithm performance without regard to architecture is no longer pertinent. Today, algorithms have to address parallelism, multithreading and memory topology (private/shared memory, cache or scratchpad, ...). Most Connected Component Labeling (CCL) algorithms are sequential, direct and optimized for processors. Few were designed specifically for GPU architectures and none were designed to be adapted to different architectures. The most efficient GPU implementations are iterative; in order to manage synchronizations between processing units, but the number of iterations depends on the image shape and density. This paper describes the DLP (Distanceless Label Propagation) algorithms, an adaptable set of algorithms usable both on GPU and multi-core architectures, and DLP-GPU, an efficient direct CCL algorithm for GPU based on DLP mechanisms.**

INTRODUCTION

Connected Component Labeling (CCL) and Connected Component Analysis (CCA) algorithms play a central part in machine vision because they often constitute a mandatory step between low-level image processing (filtering) and high-level image processing (recognition, decision). As such, CCL algorithms have a lot of applications and derivate algorithms like convex hull computation, hysteresis filtering or geodesic reconstruction. The CCL processes the output data from binary segmentation (fig. 1 top-left) and produces an image of labels easily understandable by humans (fig. 1 middle-right). The CCA also provides information on labels, typically the bounding box and the first statistical moments.

The first proposition of a direct algorithm [13] was sequential and focused on the memory management of equivalence labels. Since, many optimizations were provided for CPU implementations of direct algorithms. In [1], the authors evaluated these optimizations in a sequential and multi-core context, provided a methodology to adapt all direct algorithms to multi-core processors and established a benchmark procedure that highlights their adaptation to CPU-based architectures. The fastest CCL and CCA algorithm is LSL_{RLE} (a direct run-based one), the fastest pixel-based direct algorithm is HCS_2 DT ARemSP [2]. In order to take advantage of the SIMD instructions and the increasing number of cores, the authors of [7] proposed an adaptation of iterative algorithms for such architectures and concluded that iterative algorithms had the potential to match and outperform direct algorithms from the tile synchronization point of view. But the dependency (variable number of iterations) to the image shape and density is a considerable limitation of iterative algorithms.

The contribution presented in this paper consists of two elements:

- The Distanceless Label Propagation algorithms (DLP), the missing link between direct and iterative algorithms enhanced with a *recursive* union-find algorithm. This adaptable set of mechanisms allows for creating various implementations of CCL algorithms finely tuned to each architecture.
- The DLP-GPU: a GPU-dedicated implementation of DLP that provides an efficient CCL stage for computer vision GPU applications.

This paper is organized as follows: the first section presents the specifics of direct and iterative algorithms, the second section presents the DLP mechanisms, the third section presents the DLP-GPU specialization, and the last section presents the performance analysis of reproducible benchmarks on two GPUs architectures and a comparison with one of the fastest pixel-based CCL algorithm on CPU.

I. DIRECT AND ITERATIVE CCL ALGORITHMS SPECIFICITIES

Historical algorithms were designed by pioneers like Rosenfeld [13] and Lumia [8] who designed direct pixel-based algorithms, Ronse [11] for direct run-based algorithms and Haralick [3] for pixel-recursive iterative algorithms. Modern algorithms are derived from the algorithms of the 80's and try to make improvements by replacing some components with more efficient ones in terms of architecture. An extensive bibliography can be found in [5] and [15].

A. Direct algorithms

Direct algorithms process the input image pixel by pixel with a neighborhood mask and an equivalence table that holds a graph structure (oriented forest) to represent the label connections (fig. 1 bottom). beling (fig. 1 center-right), to replace temporary label by the final label (usually the smallest one of the component). All direct algorithms share the same three steps: 1) A first labeling (fig. 1 middle-left), that assigns a temporary/provisional label to each pixel and builds labels equivalences. 2) The label equivalences solving, that computes the transitive closure of the graph. 3) the final labeling (fig. 1 center-right), to replace each temporary label by its final label (usually the smallest one within the component). Consequently, the main improvements come from the shape

of the considered neighborhood and the optimization of the graph manipulation with Union-Find implementations.

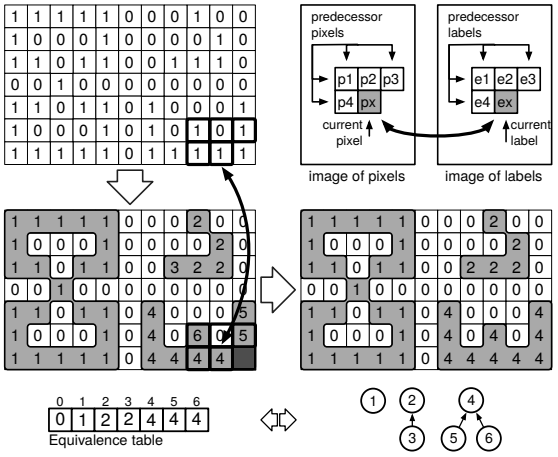


Fig. 1. An example of 8-connected component labeling with Rosenfeld's direct algorithm. Top-left: binary image, top-right: predecessor pixels direct mask and predecessor labels direct mask, middle-left: image of temporary labels, middle-right: image of final labels, bottom: equivalence table and its associated graph.

B. Iterative algorithms

Where direct CCL algorithms use an additional equivalence table to manage the connexion information, iterative algorithms propagate the labels step by step across the image. The number of iterations depends on the blob shapes within the image. The main implementations are: embarrassingly parallel (EP), pixel-recursive with forward scan (F) and pixel-recursive with forward-backward (FB) scans. EP is fundamental to understanding the behavior of iterative algorithms and their drawbacks.

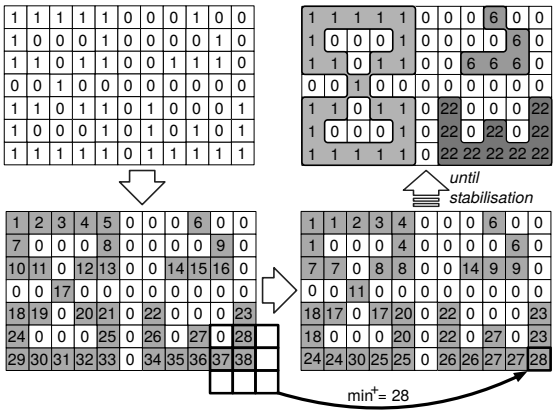


Fig. 2. Iterative EP: top-left: binary image, top-right: image of final labels, bottom-left: The initialization with unique labels, bottom-right: labeling in progress.

1) The EP iterative algorithm: It uses two images of labels, one for the input and one for the output. The initialization consists in providing a unique temporary label to each non-zero pixel (fig. 2, bottom left). Then the iterations consist in computing the minimal positive (min^+) value over a neighborhood (typically 3×3 like fig. 3 left) from the input and writing this new value to the output. All these computations are independent and can be done in parallel. The procedure is repeated (swapping input and output) until there is no more change (fig. 2, top-right).

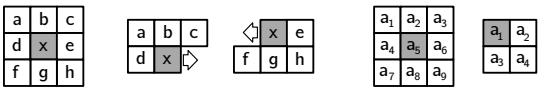


Fig. 3. Left: EP 3×3 propagation mask, center-left: forward (F) 3×2 propagation mask (identical to the direct one), center: backward (B) 3×2 propagation mask, center-right: DLP 3×3 propagation mask, right: DLP 2×2 propagation mask (used for DLP-GPU).

As the number of iterations is data-dependent, the execution time can not be predicted and is a major drawback for implementation with time constraints. For a 5×5 square (fig. 4), five iterations are required after the initial labeling: four to reach the stability and another one to detect it. But for the same square with a hole, the number of iterations is eight.

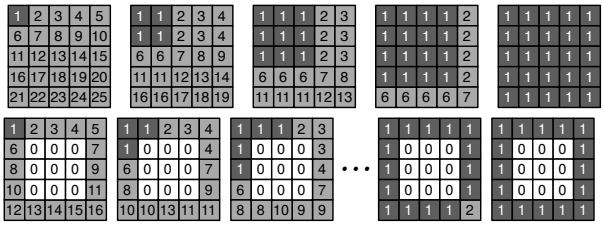


Fig. 4. The impact of the shape on labels propagation. Top: 5 iterations for a full 5×5 square. Bottom: 8 iterations for the same square with a hole. Labels in light gray are not stabilized, labels in dark gray are stabilized.

The number of iterations is the longest geodesic distance [12] between two pixels belonging to the shape plus one. Figure 5 provides four examples of geodesic distances (gd) for 5×5 shapes. If for the full square, the longest geodesic distance is $gd = 4$ (the length of the diagonal in discrete geometry), for a Z or a spiral $gd = 12$. The spiral is usually considered as the worst case for iterative CCL: for a $n \times n$ image, the number of iterations is $n^2/2 + 1$ with n even and $(n + 1) \times (n - 1)/2 + 1$ with n odd.

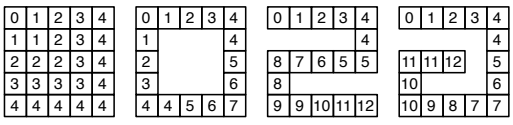


Fig. 5. The geodesic distance of four 5×5-pixel shapes (8 connected). From left to right: full square (gd=4), square with a hole (gd=7) a "Z" (gd=12) and a spiral (gd=12).

2) The pixel-recursive algorithms: The pixel-recursive algorithm (F) consists in using only one image for input and output. As the min^+ is written into the input image, it is available for the following computations. We can use either a *forward* (F) scan (fig. 3 center-left - equivalent to direct mask) or a *forward-backward* (FB) scan (fig. 3 center-left and center), invented by Haralick [4] and detailed in [7]. Note that such a pixel-recursive scan creates a serialization that is unsuitable for GPU.

II. DLP: Beyond the mask horizon

The pixel-recursive algorithms are nonetheless limited by the mask's size. Increasing the size is not a valid solution as it will lead to a reduction of the number of iterations but also to an increase of the complexity of each iteration. The DLP paradigm is about vanishing the mask influence by providing a distanceless propagation procedure. For the sake of understanding, we will first describe the different steps in DLP in a non pixel-recursive context (with a couple of images E_k, E_{k+1}) but they are suitable and more efficient for recursive ones.

A. *DLP-I: labels initialization*

DLP-I initializes all non-zero pixels $p(i,j)$ with a unique label $e(i,j) = i \times w + j + 1$ with w the image width. (fig. 6 center and fig. 7 center). Doing so, a graph structure is added into the image: the address of the implicit equivalence table T differs from the image E from one element: $T[e] \equiv E[e-1]$ that is $E[e-1] = e$ after the initialization.

In figure 6, a single-line image (left) is initialized with DLP-I (center). Once the image is labeled (right) all labels point to the minimum label (root) of their connected component.

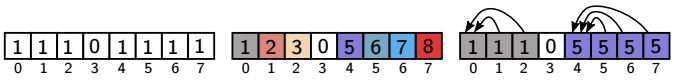


Fig. 6. From left to right: binary image, image initialized with $e_{(i)} = i+1$, after the labeling (any method) all labels point to their root.

B. *DLP-SR: propagate labels with a gather-scatter*

If classical iterative algorithms gather non-zero values and set the min^+ to the current label, DLP-SR (alg. 1) also scatters the min^+ (using SetRoot procedure - alg. 2) back to every root of the mask. The SetRoot procedure ensures that the assigned label is always minimal even in a multi-threaded context, thanks to an `atomic_min` instruction.

Algorithm 1: DLP-SR

Input: The RoI E_k can be either a tile or a full image
Result: The updated RoI is E_{k+1}
foreach e_x *of coordinates* (i,j) *in RoI* **do**
  ▷ Gather labels
  $a_1 \leftarrow E_k[i\text{-}1][j\text{-}1]$ $a_2 \leftarrow E_k[i\text{-}1][j\text{+}0]$ $a_3 \leftarrow E_k[i\text{-}1][j\text{+}1]$
  $a_4 \leftarrow E_k[i\text{+}0][j\text{-}1]$ $a_5 \leftarrow E_k[i\text{+}0][j\text{+}0]$ $a_6 \leftarrow E_k[i\text{+}0][j\text{+}1]$
  $a_7 \leftarrow E_k[i\text{+}1][j\text{-}1]$ $a_8 \leftarrow E_k[i\text{+}1][j\text{+}0]$ $a_9 \leftarrow E_k[i\text{+}1][j\text{+}1]$
  $\varepsilon \leftarrow min^+(a_1, \ldots, a_9)$
  ▷ Scatter ε
  foreach $a_p \in [1,9]$ **do**
    if $a_p \neq 0$ *and* $a_p \neq \varepsilon$ **then**
      $SetRoot(E_k, E_{k+1}, a_p, \varepsilon)$

For the spiral (fig. 7), DLP-SR seems still limited by the mask size, but as the image is also considered as a graph, we can see (fig. 8) that this step creates a connection between the labels of the two remaining connected components: 1 and 11.

After the first iteration, the labels are not all connected. An additional iteration is needed to achieve correct labeling:

Algorithm 2: SetRoot(E_k, E_{k+1}, e, ε)

$v \leftarrow E_k[e-1]$
if $v > \varepsilon$ **then**
  $E_{k+1}[e-1] \leftarrow \varepsilon$
▷ For the pixel-recursive version $E_{k+1} = E_k = E$
▷ For multi-threaded version, the *atomic min* instruction is used to implement the SetRoot procedure: **`atomic_min`**(E, e, ε)

the maximal number of iterations still depends on data. DLP-SR can be used with different labeling masks. For GPU, the minimal 2×2 neighborhood (fig. 3 right) will be used as it reduces the amount of memory accesses per label.

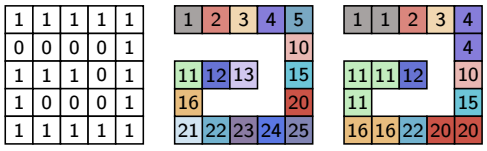


Fig. 7. From left to right: binary image, DLP-I labels initialization (with $e_{(i,j)} = i \times 5 + j + 1$) and first DLP-SR iteration with the 2×2 mask.

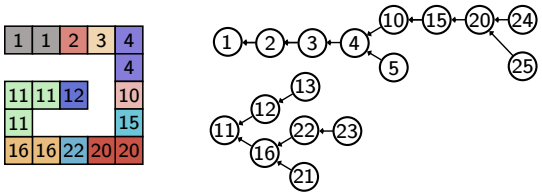


Fig. 8. Duality between image of labels and the graph structure: after the first DLP-SR iteration, only two distinct connected components remain.

C. *DLP-R: simultaneous relabeling and transitive closure*

Thanks to the duality (fig. 8) between the image of labels and the graph structure (equivalence table), both the relabeling and the transitive closure are performed by replacing a label by the root of its tree. For iterative algorithms this step is not mandatory but, depending on the architecture, it can accelerate the propagation as it regularizes the image. DLP-R flattens the graph and makes the geodesic distance limitation (fig. 9 center-left and right) disappear. With these three procedures (DLP-I, DLP-SR, DLP-R), the 5×5 spiral can be labeled in 2 iterations after initialization: (DLP-SR + DLP-R) + (DLP-SR + DLP-R) as represented in figure 9 and an additional iteration is needed to detect stabilization. Compared to the classical iterative algorithms, the $n^2/2+1$ iterations for a $n \times n$ spiral (here 13) are replaced by only 2 iterations regardless of n. This is a significant improvement but the number of iterations is still data-dependent in other cases.

This dependency comes from race conditions between labels. In the spiral case (fig. 9), the second iteration is needed due to a race condition between 2 labels: 22 (connected to 11) and 24 (connected to 1) at the site of the original label 23. It leads to lose the information "24 is connected to 23". For direct algorithms, such a kind of conflicts is solved with a Union-Find procedure, while classical iterative algorithms iterate until stabilization. In fact, very few points of conflict exist but they still trigger an additional processing of the

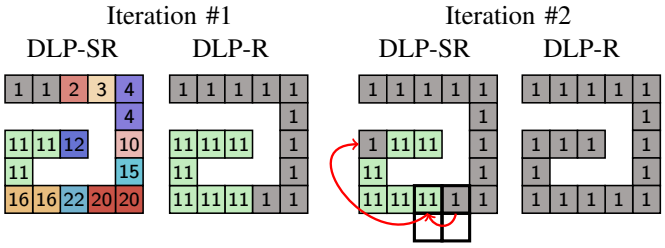


Fig. 9. Left: result after the first iteration of DLP-SR and DLP-R, right: after the second iteration. The root 1 is equivalent (assigned) to the root 11 by writing 1 to the original position of 11.

entire image. To solve these conflicts, a Union-Find procedure should replace the SetRoot procedure (alg. 2). As a Union-Find procedure can only be implemented in a pixel-recursive context (one image), all DLP procedures will be considered – in the remainder of the paper – in a pixel-recursive and multi-thread context to present their full potential.

D. *RUF: Recursive Union-Find for pixel-recursive algorithms*

Algorithm 3: RUF(E, e_k, ε): a *recursive* Union-Find

```
e_r ← E[e_k − 1] ▷ Is e_k a root?
if e_r = e_k then
    E[e_k − 1] ← ε
else
    if e_r > ε then
        RUF(E, e_r, ε)
    else
        if e_r < ε then
            RUF(E, ε, e_r)
```

In a pixel-recursive context, SetRoot procedure is replaced by a *recursive* union procedure to ensure the graph is always up-to-date: RUF (alg. 3). So, DLP-RUF can label the whole image in one iteration. In a mono-thread context, DLP-RUF combined with the forward mask (fig. 3 center) is equivalent to the classical Rosenfeld algorithm. And in a multi-thread context, the `atomic_min` instruction is mandatory to address concurrency issue (alg. 4). Thanks to DLP-RUF, the number of iterations is no more data-dependent and no additional stability check is needed: DLP procedures had created a bridge between iterative and direct algorithms.

Algorithm 4: *atomicRUF*(E, e_k, ε)

```
if e_k > ε then
    minResult = atomicMin(&E[e_k − 1], ε)
if ε > minResult then
    ▷ minResult < ε < e_k
    atomicRUF(E, ε, minResult)
else
    if e_k > minResult then
        ▷ ε < minResult < e_k
        atomicRUF(E, minResult, ε)
```

III. DLP-GPU: DLP implementation for GPU

GPUs are good candidates for a DLP implementation: they give access to efficient global and local (shared memory) *atomic min* instructions used in SetRoot and RUF procedures and provide a high number of execution units (2816 with our reference card Nvidia GTX 980_{Ti}).

A. Related works

There are few works on GPUs. In [6], the authors compare the row-col algorithm with an algorithm close to DLP-I + a classical iterative propagation called "Label Equivalence". In [10], the authors compare labeling with "Label Equivalence" and labeling with an iterative Union-Find procedure close to RUF but based on a `while` loop. Both implementations were quite slow and inefficient. [16] is the most recent iterative algorithm that is an improved version of a previous algorithm [17]. In [14], the authors present a pyramidal algorithm: iterative labeling of tiles, border tiles relabeling and iterative tile fusion in a pyramidal way. All results were obtained on a different and very restricted set of images that does not allow for concluding on performance regarding the images characteristics and thus to completely compare our results with existing benchmarks. However, we provide indications based on two partial images extracted from the Stava [14] paper (the most efficient of these algorithms) using the same GTX 480. For a low density (d) image (2%$<d<$6%, 2048×2048) DLP-GPU is a least 1.3× faster and for a higher density image (12%$<d<$36%, 2048×2048), DLP-GPU is a least 1.6× faster. To allow for an efficient comparison with future works, we based our benchmark on a reproducible procedure described in the benchmark section.

B. DLP-GPU

Creating an efficient algorithm for the GPU programming model implies minimizing the thread-divergence. Our proposition is to combine and tune the DLP mechanisms to minimize the execution time of each step.

Algorithm 5: DLP-GPU

```
▷ Step 1: tile labeling [in shared memory]
foreach tile do
    DLP-I(tile)
    DLP-SR(tile) [optional]
    DLP-R(tile) [optional]
    DLP-RUF(tile)
    DLP-R(tile)
    Labeltranslation
▷ Step 2: border merging [in global memory]
DLP-RUF(Borders)
▷ Step 3: whole image relabeling [in global memory]
DLP-R(Image)
```

DLP-GPU is a three-step algorithm (alg.5): a local tile-labeling, a global border merging and a global relabeling.

Step 1: tile labeling As the GPU works on sets of threads grouped in blocks, the image is divided into tiles (fig. 10 top-left).

1) DLP-I is locally applied to each tile (fig. 10 top-center) allowing the whole step to be executed in shared memory.

2) DLP-SR is applied with a 2×2 mask (fig. 10 top-right) in order to reduce the amount of divergences by reducing the number of tests. The south (blue) and east (red) borders are not processed. The tile is labeled regardless of the neighboring tiles. Thanks to DLP-SR gather propagation, each pixel (even from the borders) contributes to the labeling (fig. 10 top-right).
3) A first DLP-R is applied to the whole tile to accelerate propagation (fig. 10 bottom-left).
4) DLP-RUF is applied to solve the remaining conflicts. When two threads address the same labels, *atomic*-RUF recursively ensures that the connected labels are at the roots of connected components. In figure 10 bottom-center: when 10 and 7 are detected as connected, *atomic*-RUF detects than 10 is already connected to 1 and then 7 is connected to 1.
5) A second DLP-R is applied to obtain the final image of labels without an additional iteration to check stability.
6) At this point, the tiles are locally labeled ($localLab$) and need to be translated into global labels ($globalLab$) with regard to the global tile & image coordinate system: $globalLab = tile_{Index} + localLab + nline \times (w_I\text{-}w_T)$ with:
   - $localLab = tile[i \times w_T + j + 1]$,
   - $nline = \lfloor localLab / w_T \rfloor$,
   - $tile_{Index} = B_Y \times w_I + B_X$,
   - (B_X, B_Y) the tile's coordinates,
   - w_T and w_I the tile and image width.

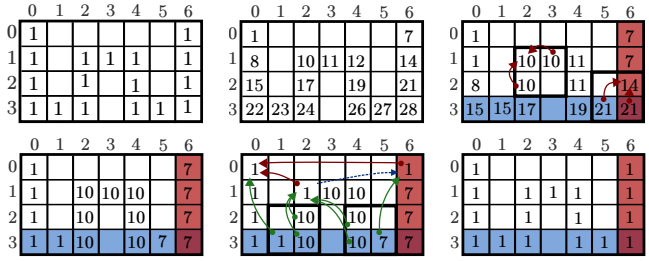


Fig. 10. Top-left: original tile, top-center: DLP-I, top-right: DLP-SR, bottom-left: first DLP-R, bottom-center: DLP-RUF, bottom-right:second DLP-R.

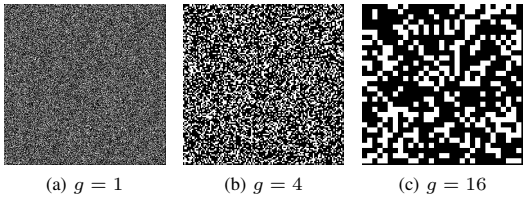
Step 2: border merging.

7) DLP-RUF is applied once to each pixel of the south (step 2a) and east (step 2b) borders. These processing are done separately for efficiency. Indeed, east borders generates non-coalescent accesses.

Step 3: relabeling

8) DLP-R is applied to each pixel of the image.

The calls to DLP-SR(2) and DLP-R(3) are optional. This is a preprocessing to *regularize* the labels within the tile. It appears (see benchmark section) that DLP-RUF + DLP-R are very time consuming. In order to reduce entropy (expressed here in sparse memory accesses), building a partial tree with DLP-SR and doing transitive closure to flatten the trees with DLP-R saves a lot of time.



(a) $g = 1$ (b) $g = 4$ (c) $g = 16$

Fig. 11. random images with density = 35% at granularity $g \in 1, 4, 16$.

IV. DLP-GPU BENCHMARK AND ANALYSIS

To allow for a fair and reproducible comparison with future works (quite difficult with data-dependent algorithms), we used the reproducible benchmark extensively detailed in [1]. The benchmark is based on random images – generated with Mersenne Twister MT19937 [9] – with variable granularity (from hard-to-label unstructured images to structured ones). The density d varying from 0% to 100% and the granularity g (size of the base pixel block) $\in [1 : 16]$ (fig. 11). Two generations of high-end Nvidia GPUs were benchmarked (tab. 1). The results are expressed with two metrics: the execution time t (in ms) and the throughput T_p (in giga-pixels/s) to allow an easy comparison between GPU generations and several architectures like multi-core processors and FPGA. The image size varies from 256×256 up to 8192×8192.

Table 1. Reference cards

Card	architecture	# cores	frequency	bandwidth
GTX 780_{Ti}	Kepler	2880	1020 MHz	336.0 GB/s
GTX 980_{Ti}	Maxwell	2816	1064 MHz	336.5 GB/s

A. Benchmark results and analysis

Global performance: Table 2 presents the mean execution time (ms) and the mean throughput (Gp/s) over density for $g \in \{1, 4, 16\}$. For both GPUs, real-time performance is achieved. If the execution time depends on the granularity. With $g = 16$, an image is processed respectively$\times 1.5$ and $\times 1.6$ than for $g = 1$ on the GTX 780_{Ti} and GTX 980_{Ti}.

Table 2. Mean execution performance over density for 2048×2048 images with granularity $g \in \{1, 4, 16\}$ on a GTX 980_{Ti}.

		Granularity		
		$g = 1$	$g = 4$	$g = 16$
GTX 780_{Ti}	t (ms)	1.56	1.27	1.05
GTX 780_{Ti}	T_p (Gp/s)	2.68	3.30	4.01
GTX 980_{Ti}	t (ms)	0.96	0.81	0.71
GTX 980_{Ti}	T_p (Gp/s)	4.37	5.19	5.87

GPU evolution: While the GPU characteristics are similar (number of cores and frequency), DLP-GPU is approximatively 50% faster on the 980_{Ti} than on the 780_{Ti}. The reason is that DLP-GPU takes advantages of the efficient native shared memory `atomic_min` instruction on Maxwell architecture for the first step and the increase of the L2 cache (2MB on Maxwell versus 256KB on Kepler).

Block size adaptations to each step: The modular conception of DLP-GPU allows for finely tuning the block size for each step. Optimal performance was obtained with the following configuration: step 1: (84×8), step 2a (112×1), step 2b: (96×1), step 3 (64×2).

Relative efficiency of each step: Figure 12 shows the execution time for step 1 (green), step 2 (red) and step 3

(yellow). We can notice that, when g increases, the curves flatten and come closer to a straight line (for g=16).

If we focus on the *cpp* (cycle per pixel and per core) versus the execution time (tab. 3), the border processing (step 2) is very cycle consuming. It takes quite the same time that the relabeling (step 3) while it only processes borders, which have seven times less data. It comes from the noncoalescent accesses to the east borders pixels and to the equivalence table within the image.

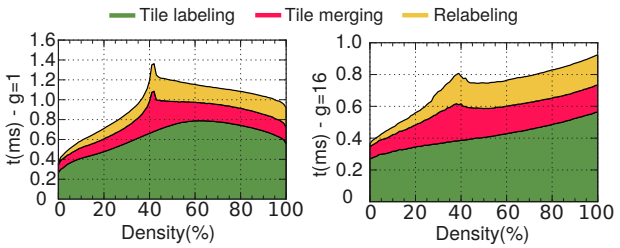


Fig. 12. DLP-GPU: execution time for 2048×2048 images, for $g = 1$ (left) and $g = 16$ (right) on a GTX 980_{Ti}.

Table 3. *cpp* and execution time of DLP step on 980_{Ti} - g=16

step	*cpp*	data (Mpixel)	t (ms)
step 1	421	4.00	0.48
step 2	897	0.57	0.14
step 3	158	4.00	0.18

Efficiency of the optional processing step: In step 1 (alg. 5), the optional preprocessing (DLP-SR and DLP-R) provides an average speedup of $\times 1.5$ for low-density images and increase up to $\times 2.5$ for high-density images. The reason is that the preprocessing performs a path compression of the equivalence table that can be viewed as a regularization.

Sustained peak performance: Finally, the throughput performance (fig. 13) increases with g and the peak is quickly reached. The image size for which half of the peak performance ($N_{1/2}$ metric) is reached is around 640×640. That is – by far – less than for linear algebra and matrix multiplication: DLP-GPU is well-adapted to GPU and the peak performance of the GPU is quickly available.

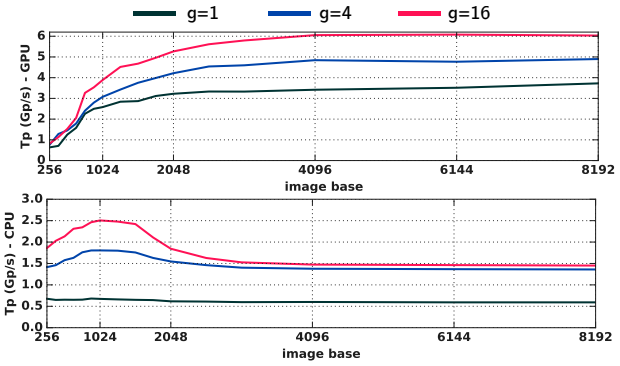


Fig. 13. Throughput versus $g \in \{1, 4, 16\}$ for DLP-GPU on a 980_{Ti} (top) and HCS_2+ARemSP on a 4-core i7 6700K (bottom).

The execution time of the algorithm proposed in [16] is 3.7 ms for a 1024×768 image and 85 ms for a 4K image resulting in a throughput of 272 and 12 MPix/s on a GTX 680. DLP reaches a throughput of 1700 MPix/s in the same conditions.

CPU comparison: we compare it with one of the fastest pixel-based CCL on CPU: HCS_2+ARemSP (all algorithms are very close on CPU, as they are memory bound). Fig 13 shows the performance of the 4-core i7 Skylake running at 4 GHz. The GPU Thoughtput is betweeen 3.5 and 6 Gpix when the CPU is between 0.7 and 2.5 (before the cache overflow). There is a factor 2.4 for g=16 and 5.0 for g=1. For single chip, pixel-based DLP algorithm for GPU outperform pixel-based CPU ones.

V. Conclusion

In this paper, we have presented DLP, a new set of algorithms for multicores and GPUs. DLP-GPU deeply takes into account the architectural properties to be efficient. Thanks to a recursive union-find with atomic instructions, DLP is no more iterative but direct like the algorithms for multi-core processors. The equivalence table is embedded within the image in order to reduce the number of memory accesses and also to simplify and combine transitive closure and relabeling operations. Moreover, a pre-processing step is used to speedup the border processing. In the future, we plan to port DLP and its competitors on an embedded platform like Nvidia Jetson, to compare the performance of the embedded CPU with GPU.

References

[1] 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*, pages 1–18, 2016.

[2] S. Gupta, D. Palsetia, M. A. Patwary, A. Agrawal, and A. Choudhary. A new parallel algorithm for two-pass connected component labeling. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1355–1362. IEEE, 2014.

[3] R. Haralick. Some neighborhood operations. In *Real-Time Parallel Computing Image Analysis*, pages 11–35. Plenum Press, 1981.

[4] R. Haralick and L. Shapiro. *Computer and Robot Vision*. Addison-Wesley ISBN 0-201-56943-4, 1992.

[5] L. He, Y. Chao, and K. Suzuki. A run-based two-scan labeling algorithm. In *ICIAR*, pages 131–142. LNCS 4633, 2007.

[6] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider. Connected component labeling on a 2d grid using cuda. *Journal of Parallel and Distributed Computing*, 71(4):615–620, 2011.

[7] L. Lacassagne, L. Cabaret, D. Etiemble, F. Hebache, and A. Petreto. A new SIMD iterative connected component labeling algorithm. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, pages 1:1–1:8, 2016.

[8] R. Lumia, L. Shapiro, and O. Zungia. A new connected components algorithms for virtual memory computers. *Computer Vision, Graphics and Image Processing*, 22-2:287–300, 1983.

[9] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *Transactions on Modeling and Computer simulation*, 8(1):3–30, 1998.

[10] V. Oliveira and R. Lotufo. A study on connected components labeling algorithms using GPUs. In *Workshop of Undergraduate Works, XXIII Sibgrapi, Conference on Graphics, Patterns and Images*, 9 2010.

[11] C. Ronse and P. Dejvijver. Connected components in binary images: the detection problems. In *Research Studies Press*, 1984.

[12] A. Rosenfeld. Geodesics in digital pictures. *Information and Control*, 36(1):74–84, 1978.

[13] A. Rosenfeld and J. Platz. Sequential operator in digital pictures processing. *Journal of ACM*, 13,4:471–494, 1966.

[14] O. Stava and B. Benes. *Connected Component Labeling in CUDA*. GPU Computing Gems Emerald Edition, 2011.

[15] K. Wu, E. Otoo, and A. Shoshani. Optimizing connected component labeling algorithms. *Pattern Analysis and Applications*, 2008.

[16] G. Ziegler. Connected components revisited on Kepler. In Nvidia, editor, *GPU Technology Conference*, pages 1–56, 2014.

[17] G. Ziegler and A. Rasmusson. Efficient volume segmentation on the gpu. In Nvidia, editor, *GPU Technology Conference*, pages 1–44, 2010.