A Review of World's Fastest Connected Component Labeling Algorithms: Speed and Energy Estimation

Laurent Cabaret Lionel Lacassagne Louiza Oudni
Laboratoire de Recherche en Informatique
Inria/Univ. Paris Sud, F-91405 Orsay, France
email: firstname.name@lri.fr

***Abstract*—Optimizing connected component labeling is currently a very active research field. The current most effective algorithms although close in their design are based on different memory/computation trade-offs. This paper presents a review of these algorithms and a *detailed* benchmark on several Intel and ARM embedded processors that allows to focus on their advantages and drawbacks and to highlight how processor architecture impact them.**

INTRODUCTION

Binary Connected Component Labeling (CCL) algorithms deal with graph coloring and transitive closure computation. CCL algorithms play a central part in machine vision, because it is often a mandatory step between low-level image processing (filtering) and high-level image processing (recognition, decision). As such, CCL algorithms have numerous applications and derivate algorithms like: convex hull computation, hysteresis filtering, geodesic reconstruction.

Designing a new algorithm is challenging both from considering the overwhelming literature and the performance of best existing algorithms. Goals might be a faster algorithm on some class of computer architecture or minimizing the number of over-created labels or the smallest theoretical complexity. Yet another issue is to be most predictable.

Now, from the current state of the computing technology, reaching decent performances in actuality requires for CCL algorithms to take into account two specificities/capacities of current General Purpose Processors (GPP): the processor pipeline and its cache memories. That amounts to minimize conditional statements (like tests and comparisons) to reduce the number of pipeline stalls and limit random sparse (typically vertical) memory accesses, to lower cache misses.

The embedded processing applications ask continuously to process bigger images in a smaller time and to consume as little energy as possible. That is why we focused on mobile processors in this study.

As it is an intermediate level algorithm, CCL processes the output data coming from low level algorithms (binary segmentation, ...) and provides abstract input data to other intermediate or high level algorithms. Usually, such abstract data also called *features* are the boundary of bounding rectangle (for target tracking) and the first order statistical moments (surface, centroid, orientation, ...). So, if a standalone CCL algorithm can be considered at first step, the couple "CCL + features computation" is the procedure to be actually evaluated at end.

Our contribution consists of three elements:

- an enhanced benchmark that incorporates random images with different granularities. That can be seen as a bridge between classical random images of density and data base images,
- a performance benchmark with all state-of-the-art algorithms on embedded general purpose processors from Intel and ARM,
- an analysis from the energy point of view.

In the remainder of this paper we shall describe modern algorithms and describe the benchmark's procedure and hardware. We then present the results on Intel's and ARM's architectures and finally provide a comparison from the energy point of view.

I. CONNECTED COMPONENT LABELING ALGORITHMS

Historical algorithms were designed by pioneers like Rosenfeld [14], Haralick [4], and Lumia [10] who designed pixel-based algorithms, and Ronse [13] for run-based algorithm. Modern algorithms derive from the historical ones and try improvements by replacing some components by a more efficient one. An extensive bibliography can be found in [5] and [16].

Except Contour Tracing algorithm [1] that is aesthetic but inefficient, all modern algorithms are two-passes (or less) algorithms, none is a data dependent multi-pass algorithm. They share the same three steps:

- first labeling that assigns a temporary/provisional label to each pixel and builds labels equivalence,
- label equivalences solving that is to compute the transitive closure of the graph associated to the label equivalence table,
- second labeling to replace temporary label by the final label (usually the smallest one of the component).

They differ on three points: the mask topology, the number of tests for a given mask to find out the label, and the equivalence management algorithm.

Using Rosenfeld mask (fig. 1), only two basic patterns trigger label creation (fig. 2), whatever is the connectivity (here 8-connectivity). The first one is the *stair*. It is responsible for the unnecessary provisional label created by pixel-based algorithm

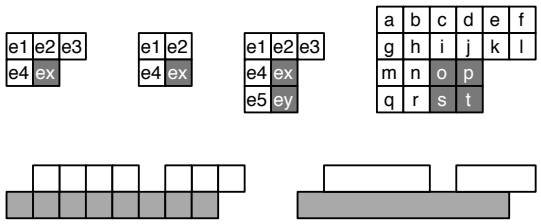

Fig. 1. Masks: first line: $Rosenfeld$, RCM, HCS_2, and $Grana$, second line: HCS and LSL

like Rosenfeld's one. The second one is the *concavity*. With the neighborhood CCL locality, it is obvious that the label creation cannot be avoided.

As figure 4 and figure 5 show, the execution time is not directly correlated to the total amount of final labels, but to the number of stairs and concavities that generate equivalence building. So, one way to improve CCL algorithms is to widen the label mask. That leads to block-based algorithms (fig. 1) like HCS_2 [7] and Grana [3] that respectively compute 2 and 4 labels from 6-pixel and 16-pixel neighborhood. One the opposite way, RCM [8] introduces a mask with only 3 neighbors in order to reduce the amount of tests. $Grana$'s mask can detect some concavities and avoid label creation if these concavities are small enough to entirely fit in the mask. But the only way to prevent label creation from *stair* is to use a run-based algorithm like HCS [6] or Light-Speed Labeling (LSL) [9] that first detect the pixel adjacency in the neighborhood before to assign a label to the run. The LSL uses a tricky line-relative labeling to generate RLC coding to directly find adjacent runs on the previous line whereas HCS has to perform a test on every pixel to decide to continue to propagate a label or to perform an equivalence.

Fig. 2. Minimal 8-connected basic patterns generating temporary labels: stair (left) and concavity (right)

The second point to enhance algorithm efficiency is to reduce the number of tests. A *decision tree* (DT) [16] reduces the average number of neighbor to test to find out the value to assign to the current label based on mask topology. For pixel-based algorithms, it decreases the complexity of the 8-connectivity to the 4-connectivity one. For block-based algorithm, DT is mandatory. Another way to reduce complexity is to perform a *path-compression* (PC) [2]. It is a step added to the Union-Find algorithm to perform a transitive closure in climbing up to the root of the equivalence. It has been proven that PC make the Union-Find complexity to grow with the inverse of Ackermann function [15].

Finally, the third point to improve is the equivalence management algorithm. Rosenfeld's algorithm uses Union-Find algorithm and the associated table to store the equivalences. An alternative approach with three tables has been proposed by [5]

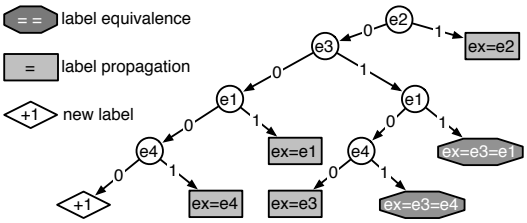

Fig. 3. 8-connected Decision Tree

now referenced as Suzuki equivalence tables. The difference is that the transitive closure is done at each equivalence rather than at the end.

The benchmarked algorithms (all 8-connected) are:

- $Rosenfeld$: original algorithm improved with DT+PC,
- $Suzuki$: Rosenfeld mask with Suzuki tables management that we improved with DT,
- RCM: pixel-based algorithm with Suzuki management,
- HCS_2: block-based algorithm with Suzuki management,
- $Grana$: block-based algorithm with 128-stage DT,
- HCS: run-based algorithm with Suzuki management,
- LSL: run-based algorithms with Union-Find management with two variants: LSL_{STD} and LSL_{RLE}.

II. Benchmarks

We present here the images and processors used for benchmarking. We also provide a qualitative analysis of temporary labels creation.

A. Random images generation and qualitative analysis

Depending on the OS and the compiler the pseudo-random number generator embedded into the libC can change, so providing the *seed* is not enough if one wants to do reproducible experiments. For that reason, the Mersenne Twister MT19937 [11] has been chosen with $seed = 0$.

Usually papers evaluate CCL performance first with random images (varying pixel density from 0% to 100%) for hard-to-label benchmark and secondly with image data base. But data base can be biased and then may favor some algorithms. In order to analyze algorithms behavior depending on some image properties: size of connected components and size of the smaller element compared to the algorithm neighborhood scale, we decided to extend random images by changing the pixel granularity. Initial random image has a granularity of 1. Then we create g-random images whose block of pixels have a size of $g \times g$, with $g \in [1 : 16]$. The symmetrical shape of these blocks ensure an equitable treatment between the different algorithms. All the random images are 1024×1024 (width $\times$ height).

The figure 4 provide the temporary labels distribution for granularity $g \in \{1, 2, 4\}$ for pixel-based, run-based and $Grana$'s algorithms (red, magenta, and blue). The number of final labels (green), concavities (cyan), and stairs (orange) is also provided.

First, if we compare run-based and pixel-based label distribution, we can see that run-based curve has always the same

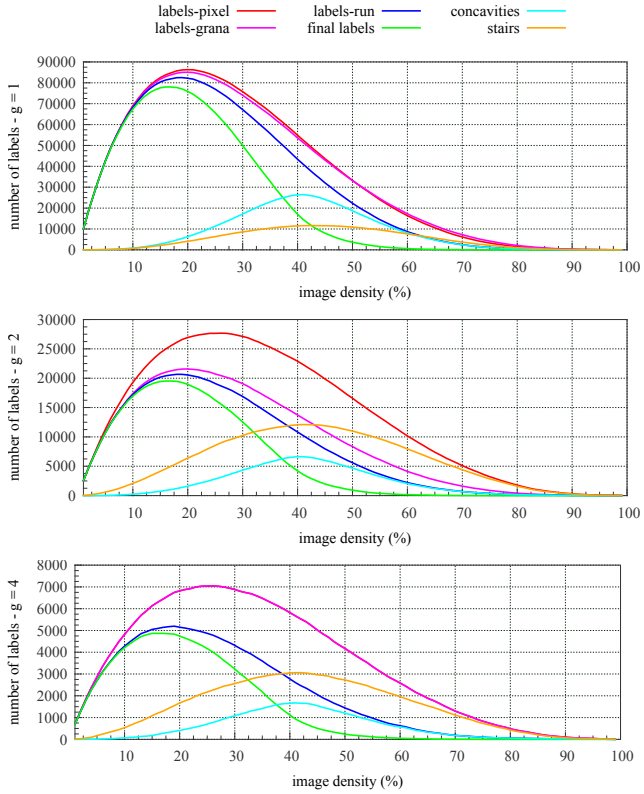

Fig. 4. Distribution of labels, concavities and stairs versus density for granularity $g \in \{1, 2, 4\}$

behavior (close to the final label curve), contrary to the pixel-based curve. The reason is that the amount of concavities is proportionally constant (from one granularity to another one) to the number of final label. For $g \geq 2$, it appears that the amount of stairs becomes bigger than concavities, and thus increases the gap between the number of labels of pixel-based and run-based. That is the reason why run-based algorithms have a better execution time when g is growing: they avoid more and more label creation.

Concerning $Grana$ algorithm, it generates quite the same number of temporary labels for $g = 1$ than pixel-based ones. For $g = 2$ it comes closer to run-based algorithms as its wide mask avoids many temporary labels. But for $g \geq 4$, its wide mask does not avoid label creation, as 4-pixel wide stair and concavities are beyond the pixel's neighborhood.

B. Image data base

The Standard Image Data-Base (SIDBA) has been used for natural image labeling. Gray-scaled images have been automatically binarized with Otsu algorithm [12]. For both random images and natural ones, we provide the *cpp* (cycle per pixel) of each algorithm, with *features* computation. The features extracted for each component are: the bounding box ($[x_{min}, x_{max}] \times [y_{min}, y_{max}]$) and the first statistical moments (S, S_x, and S_y).

C. General Purpose Processors

In order to evaluate the impact of the architecture on the execution time, we selected two mobile processors from Intel: PenrynM (U9300, 1.2GHz, 10W), HaswellM (4650U, 1.7GHz, 15W) and two embedded from ARM: CortexA9 (OMAP4460, 1.2GHz, 1.2W) and CortexA15 (Exynos5250, 1.7GHz, 1.7W). HaswellM and CortexA9 were chosen for the curves and tables, however all the SIDBA results were reported in figures 9 and 10. Executable codes were generated with Intel ICC v14.0.1 and gcc-arm 4.6.3.

III. Results and Analysis

TABLE I
AVERAGE *cpp* VERSUS GRANULARITY

	granularity				
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$
algorithms	*HaswellM*				
$Rosenfeld$	13.15	7.55	4.97	4.28	4.02
$Suzuki$	12.53	7.26	**4.68**	3.96	3.68
RCM	13.30	9.15	7.21	6.34	5.90
HCS	13.36	8.52	6.12	5.06	4.56
HCS_2	11.22	6.93	5.77	5.30	5.15
$Grana$	15.00	7.36	5.15	4.08	3.59
LSL_{STD}	**8.70**	**5.88**	4.97	4.63	4.48
LSL_{RLE}	16.42	8.54	4.73	**3.14**	**2.55**
algorithms	*CortexA9*				
$Rosenfeld$	42.20	35.82	34.14	33.35	32.46
$Suzuki$	40.67	34.86	32.86	31.85	31.21
RCM	50.99	40.92	37.63	36.27	35.88
HCS	**28.79**	**22.84**	20.74	20.25	20.19
HCS_2	40.42	30.24	29.13	29.00	29.02
$Grana$	32.85	23.18	20.61	19.60	19.46
LSL_{STD}	32.69	27.11	24.94	23.99	23.58
LSL_{RLE}	35.03	23.03	**17.20**	**14.41**	**13.75**

a) Density behavior: Figure 5 shows us that algorithm curves - except HCS_2 -, are symmetrical about their maximum value. The abscissas of the maximum values are contained in the $[45\%; 55\%]$ area depending on the algorithm. Concavities and stairs (fig. 4), lead to temporary label creation and labels merging, they also increase the probability of having more tests to perform in the decision tree (e.g., stair makes to traverse all the DT graph until the label creation node "+1" - figure. 3) and doing so, increase *cpp*.

One can observe that when the number of stairs and concavities decrease (g comes higher) the density curves tend to flatten. As described in [6], HCS_2 algorithm make no usage of decision tree and so, it needs to load the neighborhood's labels for each pixel to label. Doing so, it is not able to reduce *cpp* when density grows above 50%.

b) Granularity influence: Table I and figure 5 describe the behavior of algorithms faced to images of different granularities. The main trend is that when g grows *cpp* drops. First quickly $[\times 0.49; \times 0.69]$ for $g \in \{1, 2\}$, and then slowly $[\times 0.30; \times 0.76]$ for $g \in [2{:}16]$. One can notice that LSL_{RLE} is the most accelerated when granularity grows while LSL_{STD} is the most regular. It comes from their construction as explained in [9]. LSL_{RLE} is inefficient for $g = 1$ because of its run length encoding kernel. RCM is efficient but only for $g = 1$, this is due to the smaller number of tests it performs compared to $Rosenfeld$ which is an efficient strategy on unstructured data.

LSL_{STD} is first for $g \in [1{:}4[$, $Suzuki$ is first for $g = 4$ and LSL_{RLE} is first for $g \in]4{:}16]$. Note that $g = 4$ is a

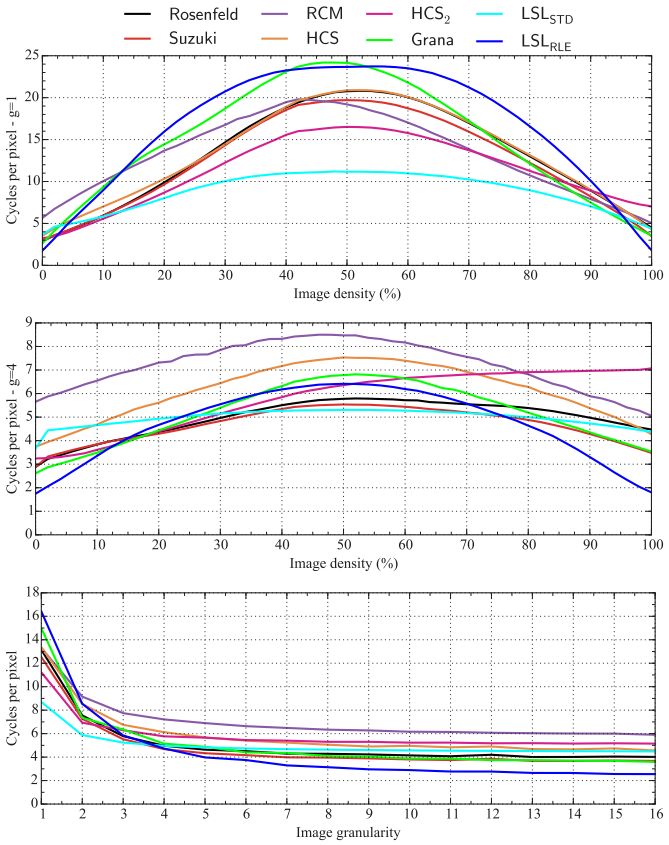

Fig. 5. HaswellM: *cpp* over density for granularity $g \in \{1, 4\}$ and average *cpp* versus granularity

turning point where LSL_{RLE}, LSL_{STD}, $Suzuki$, $Grana$, and $Rosenfeld$ are close. Their different trade-offs between memory management and number of tests are broadly equivalent in performances for this granularity value. For structured data (higher g values), the algorithm ranking is (first to last): LSL_{RLE}, $Grana$, $Suzuki$, $Rosenfeld$, LSL_{STD}, HCS, HCS_2, and RCM.

LSL is still the fastest algorithm on Intel architecture. If the application field provides unstructured images ($g \in [1:3]$), LSL_{STD} should be chosen, otherwise LSL_{RLE}.

c) Features computation (FC) influence: When FC is activated, LSL_{STD} and LSL_{RLE} outperform all others algorithms (table II and fig. 6). This is mostly due to on-the-fly FC, which make the last relabeling scan unnecessary [9]. FC is almost always faster than doing the second labeling, especially for LSL_{RLE} where *run length coding* speeds up FC. In facts, the addition of FC accelerates the LSL algorithm. LSL_{STD} is first for $g \in [1{:}2]$ and LSL_{RLE} is first for $g \in]2{:}16]$. For structured data (higher g values), the algorithm ranking becomes (first to last): LSL_{RLE}, LSL_{STD}, $Grana$, $Suzuki$, $Rosenfeld$, HCS, HCS_2, and RCM.

FC increases equally the *cpp* of every other algorithms depending on g (even if number of pixels is constant as density is constant). From 4.3 *cpp* for $g = 16$ up to 9.0 *cpp* for $g = 1$

TABLE II
AVERAGE *cpp* ACCORDING TO GRANULARITY WITH FC

	granularity				
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$
algorithms	*HaswellM*				
$Rosenfeld$	22.07	14.35	10.43	9.04	8.41
$Suzuki$	21.50	14.05	10.17	8.78	8.13
RCM	22.10	15.77	12.48	10.92	10.15
HCS	22.36	15.30	11.56	9.82	8.96
HCS_2	20.22	13.74	11.23	10.05	9.52
$Grana$	23.98	14.13	10.57	8.80	7.97
LSL_{STD}	**9.55**	**5.71**	4.34	3.75	3.42
LSL_{RLE}	12.73	6.25	**3.66**	**2.59**	**2.01**
algorithms	*CortexA9*				
$Rosenfeld$	65.92	57.18	54.43	53.01	51.67
$Suzuki$	65.47	56.55	53.20	51.43	50.26
RCM	76.55	62.81	58.47	56.50	55.57
HCS	53.28	44.36	41.01	39.86	39.21
HCS_2	64.06	51.66	49.74	48.91	48.29
$Grana$	57.28	44.28	40.79	39.09	38.25
LSL_{STD}	31.24	21.44	17.84	16.27	15.59
LSL_{RLE}	**25.19**	**15.36**	**11.75**	**10.10**	**9.39**

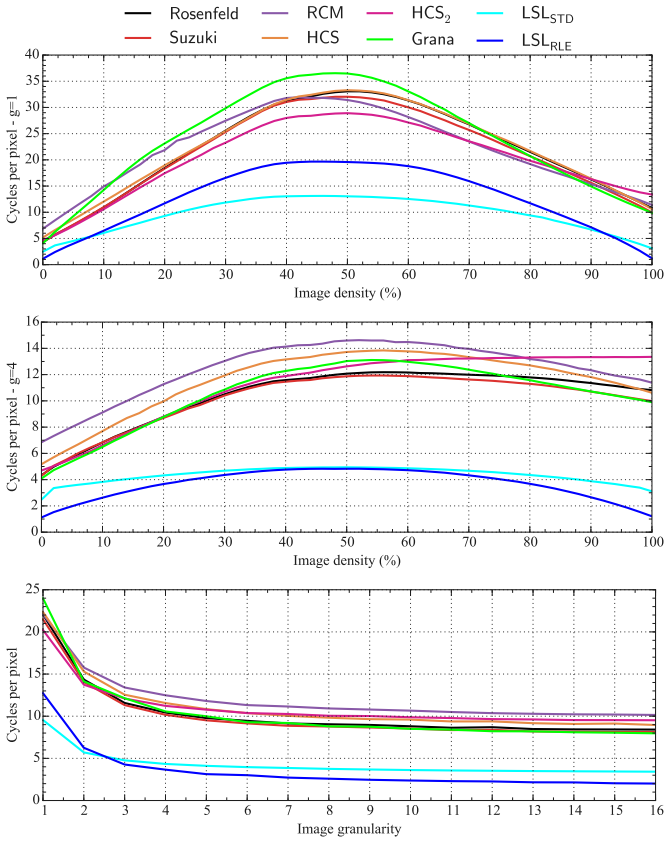

Fig. 6. HaswellM: *cpp* over density for granularity $g \in \{1, 4\}$ and average *cpp* over granularity with FC

for HaswellM (18.8 *cpp* up to 25.6 *cpp* for CortexA9). Those variations are explained by the structure of the image (fig. 4). If granularity is low there are more labels than if granularity is high. So sparser memory accesses will happen, leading to different amounts of cache hits and cache misses.

d) Architecture influence - HaswellM/CortexA9: The most noticeable differences between HaswellM and CortexA9 are the increase of *cpp* and the evolution of the algorithms rank. Table III highlights the *cpp* ratio between HaswellM and

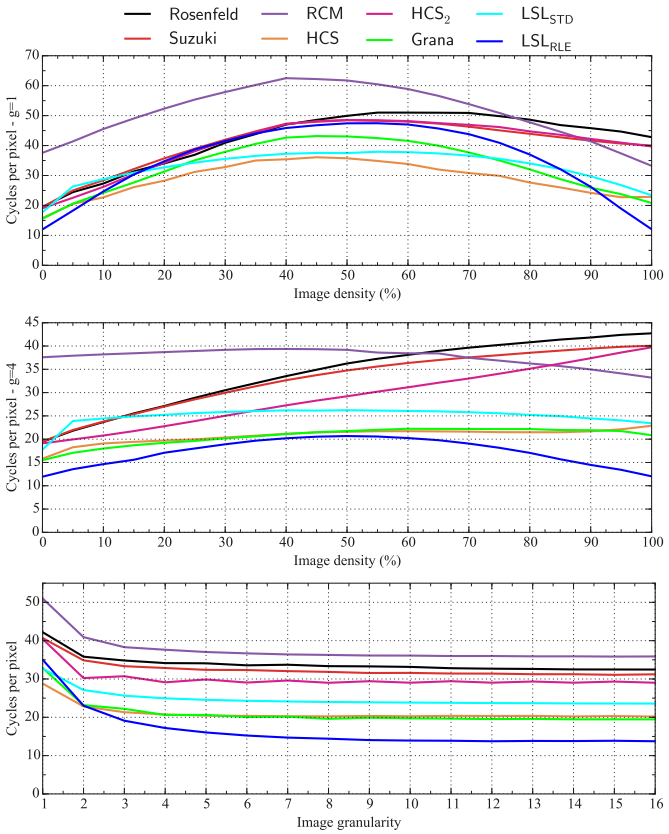

Fig. 7. CortexA9: *cpp* over density for granularity $g \in \{1, 4\}$ and average *cpp* versus granularity

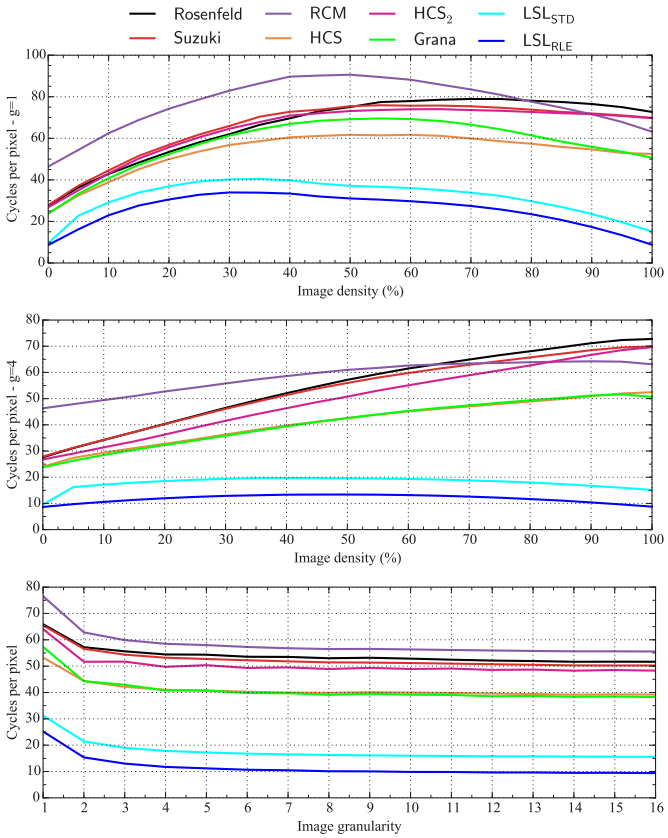

Fig. 8. CortexA9: *cpp* vs density for granularity $g \in \{1, 4\}$ and average *cpp* over granularity with FC

CortexA9 for each algorithm. The lowest value means that the algorithm is comparatively less slowed on CortexA9 than the others, One can remark that, LSL_{RLE}, HCS, $Grana$, and LSL_{STD} make a better use of the CortexA9 than HCS_2, RCM, $Rosenfeld$, and $Suzuki$. There are two explanations: conditional instructions and memory latency who question the trade-offs made by algorithms. RCM does less tests for each pixel, less loads for foreground pixel, but more loads for background ones. These choices might be valuable on HaswellM but are less efficient on CortexA9. $Grana$ and HCS_2 execute more tests than others and less memory accesses due to their block-based construction. The gap between LSL_{RLE} and LSL_{STD} is bigger on CortexA9 than HaswellM, because LSL_{RLE} performs less memory accesses (only for the start and the end of runs). As HCS is a run-based algorithm it performs less memory access than pixel based.

TABLE III
cpp RATIO CORTEXA9/HASWELLM FOR RANDOM IMAGES WITH FC

	granularity				
algorithms	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$
$Rosenfeld$	2.99	3.99	5.22	5.86	6.14
$Suzuki$	3.05	4.02	5.23	5.86	6.18
RCM	3.46	3.98	4.68	5.17	5.48
HCS	2.38	2.90	3.55	4.06	**4.38**
HCS_2	3.17	3.76	4.43	4.87	5.07
$Grana$	2.39	3.13	3.86	4.44	4.80
LSL_{STD}	3.27	3.76	4.11	4.34	4.56
LSL_{RLE}	**1.98**	**2.46**	**3.21**	**3.90**	4.67

e) Real case images: The benchmark on natural images from SIDBA data base confirms the random images conclusion. We give the results for algorithms with FC (table IV) with min, average and max values for processing time and *cpp*, to allow direct comparison with others articles' results.

TABLE IV
EXECUTION TIME AND *cpp* FOR SIDBA WITH FC

	time(ms)			*cpp*		
	min	*avg*	*max*	*min*	*avg*	*max*
algorithms	*Haswell*					
$Rosenfeld$	2.34	2.73	3.16	8.28	9.65	11.19
$Suzuki$	2.27	2.69	3.50	8.03	9.54	12.40
RCM	2.64	2.99	3.29	9.35	10.58	11.66
HCS	2.38	2.78	3.17	8.44	9.85	11.23
HCS_2	2.63	3.06	3.61	9.32	10.84	12.78
$Grana$	2.14	2.59	3.04	7.58	9.19	10.77
LSL_{STD}	0.87	1.02	1.18	3.08	3.61	4.17
LSL_{RLE}	**0.42**	**0.69**	**1.00**	**1.48**	**2.43**	**3.55**
algorithms	*A9*					
$Rosenfeld$	19.20	23.41	28.84	48.00	58.53	72.10
$Suzuki$	18.80	22.69	27.32	47.01	56.73	68.30
RCM	23.40	24.30	25.79	58.51	60.75	64.49
HCS	15.10	17.55	20.40	37.74	43.87	50.99
HCS_2	17.42	21.39	26.19	43.55	53.47	65.47
$Grana$	14.92	17.01	19.81	37.31	42.53	49.52
LSL_{STD}	7.38	8.23	8.81	18.46	20.58	22.03
LSL_{RLE}	**5.62**	**6.00**	**6.62**	**14.04**	**14.99**	**16.56**

Figures 9 and 10 present *cpp* for all algorithms on all architectures for the SIDBA data base. On HaswellM, the

ranking without FC is LSL_{RLE}, $Grana$, $Suzuki$, LSL_{STD}, $Rosenfeld$, HCS, RCM, and HCS_2 and with FC the ranking is LSL_{RLE}, LSL_{STD}, $Grana$, $Suzuki$, $Rosenfeld$, HCS, RCM, and HCS_2.

One can remark that - in average - new architectures are faster than previous ones and Intel's architectures are faster than ARM's one. But not identically for all algorithms: HCS and $Grana$ take both a great advantage of the CortexA15 architecture. Without FC they are the fastest on CortexA15, but on a real application case (with FC), LSL (RLE and STD) are the world's fastest algorithm.

LSL_{STD} has a very stable execution time for all images (table IV): For SIDBA, the execution time variation between images is $0.31ms$ on HaswellM ($0.21ms$ without FC), whereas for $Rosenfeld$ the variation is $0.82ms$ on HaswellM ($0.48ms$ without FC) :

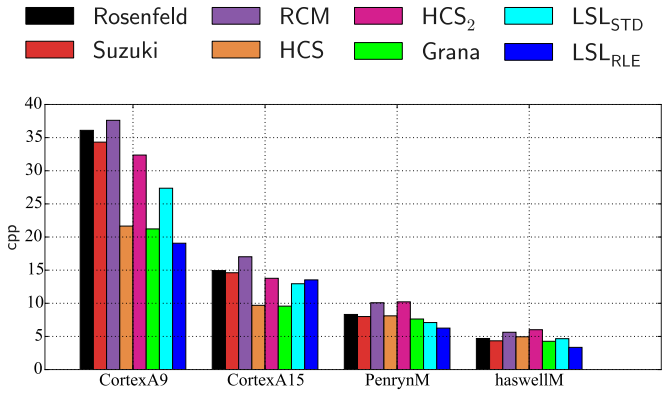

Fig. 9. Histogram of average *cpp* for SIDBA

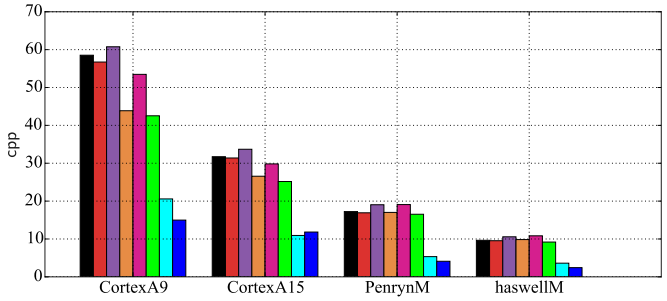

Fig. 10. Histogram of average *cpp* for SIDBA with FC

f) Architecture influence - A15/A9: The relative order of algorithm is maintained except for LSL_{RLE} that is less accelerated than the others. This is due to its already optimized memory management that takes less advantages from $A15$ optimizations.

g) Energy consumption: Table V presents I_E an energy index that is proportional to the average energy consumption (I_E = t $\times$ TDP of the whole dual-core processor). As TDP reflects the power consumption of the two cores, I_E is higher than the real energy consumption. But as the benchmarked processor have two cores, I_E enforces the order relation between processors. On HaswellM, PenrynM, and CortexA9, LSL_{RLE} is the best. On $A15$ it is LSL_{STD}. The CortexA15 is, right now, the most energy-efficient architecture.

TABLE V
ENERGY ESTIMATION WITH FC (mJ)

	Architectures			
algorithms	CortexA9	CortexA15	PenrynM	HaswellM
$Rosenfeld$	*28.1*	*13.4*	*68.9*	*40.9*
$Suzuki$	*27.2*	*13.3*	*67.6*	*40.4*
RCM	*29.2*	*14.3*	*76.2*	*44.8*
HCS	*21.1*	*11.2*	*68.1*	*41.7*
HCS_2	*25.7*	*12.6*	*76.3*	*45.9*
$Grana$	*20.4*	*10.7*	*66.1*	*38.9*
LSL_{STD}	*9.9*	**4.6**	*21.4*	*15.3*
LSL_{RLE}	**7.2**	*5.0*	**16.5**	**10.3**

IV. CONCLUSION AND FUTURE WORKS

In this paper, we have proposed a new detailed benchmark procedure for connected component labeling with granularity steps that is complemented with the use of a standard database.

The benchmark procedure, confirms that for real applications (that is with features computations) LSL_{RLE} algorithm outperforms all state-of-the-art algorithms, on both Intel and ARM processors. For time-predictability and standard deviation, LSL_{STD} is the best choice.

Future works will consider parallelization of CCL on GPP.

REFERENCES

[1] F. Chang and C. Chen. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93:206–220, 2004.

[2] T. Cormen, C. Leiseirson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[3] C. Grana, D.Borghesani, and R. Cucchiara. Fast block based connected components labeling. In *ICIP*, pages 4061–4064. IEEE, 2009.

[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, 2007.

[6] L. He, Y. Chao, and K. Suzuki. An efficient first-scan method for label-equivalence-based labeling algorithms. *Pattern Recognition Letters*, 31(1):28–35, 2010.

[7] L. He, Y. Chao, and K. Suzuki. A new two-scan algorithm for labeling connected components in binary images. In W. Congress, editor, *Proceedings of the World Congress on Engineering*, volume 2, pages p1141–1146, 2012.

[8] U. Hernandez-Belmonte, V. Ayala-Ramirez, and R. Sanchez-Yanez. Enhancing ccl algorithms by using a reduced connectivity mask. In Springer, editor, *Mexican Conference on Pattern Recognition*, pages 195–203, 2013.

[9] L. Lacassagne and B. Zavidovique. Light speed labeling: efficient connected component labeling on risc architectures. *Journal of Real-Time Image Processing*, 6(2):117–135, 2011.

[10] 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.

[11] 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.

[12] N. Otsu. A threshold selection method from gray-level histograms. *Transactions on System, Man and Cybernetics*, 9:62–66, 1979.

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

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

[15] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, pages 215–225, 1975.

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