A new SIMD iterative connected component labeling algorithm

Lionel Lacassagne, Laurent Cabaret, Daniel Etiemble, Farouk Hebbache, Andrea Petreto

A new SIMD iterative connected component labeling algorithm

Lionel Lacassagne
Laboratoire d'Informatique de Paris 6 (LIP6)
Univ. Pierre et Marie Curie
lionel.lacassagne@lip6.fr

Laurent Cabaret
Daniel Etiemble
Farouk Hebache
Andrea Petreto
Laboratoire de Recherche en Informatique (LRI)
Univ. Paris-Sud
firstname.name@lri.fr

Abstract

This paper presents a new multi-pass *iterative* algorithm for Connected Component Labeling. The performance of this algorithm is compared to those of State-of-the-Art two-pass *direct* algorithms. We show that thanks to the parallelism of the SIMD multi-core processors and an activity matrix that avoids useless memory access, such algorithms have performance that comes closer and closer to direct ones. This new active-tile iterative algorithm has been benchmarked on four generations of Intel Xeon processors: 2×4-core Nehalem, 2×12-core Ivy-Bridge, 2×14-core Haswell and 57-core Knight Corner. Macro meta-programming was used to design a unique code for SSE, AVX2 and KNC SIMD instruction set.

Keywords Connected Component Labeling algorithm, SIMDization, SSE, AVX, KNC, Intel Xeon-Phi, direct vs iterative algorithms.

1. Introduction

Some algorithms exist in two versions. One version is usually named *direct* and run in a fixed number of iterations (classically one or two) whereas another one is *iterative* with much more iterations than the first one. Such algorithms exist in linear algebra and in image processing fields. As SIMD and multicore parallelisms are present in all processors and their degree increase continuously, one can wonder whether a *brute-force* iterative algorithm combining SIMD and multi-threading can match the performance of a *smart* direct algorithm using only multi-threading (direct algorithms cannot use SIMD).

This paper introduces the implementation of a new iterative connected component labeling (CCL) algorithm that uses SIMD, OpenMP and tiling to reduce the number of iterations and thus accelerate its execution. CCL algorithms have been selected as they are used in many applications in computer vision (video surveillance, motion detection, motion tracking, optical characters recognition) and are part of other algorithms in image processing (hole filling, Euler number computing, hysteresis thresholding, max-tree computation, level-sets computation). They are also representative of a class of algorithms (discrete geometry, mathematical morphology) and so, optimizing them will help to optimize/transform/redesign the algorithms of the same class.

Our contribution consists in two elements:

- a new SIMD iterative algorithm with tiling,
- some benchmarks for parallel versions on multiple architectures with various SIMD (SSE, AVX2, and KNC),
- a comparison with State-of-the-Art direct algorithms.

The paper is organized as follows: the first section quickly presents the State-of-the-Art CCL algorithms and details our new iterative algorithm. The second section presents the benchmarks run on a representative set of processors with all available Intel SIMD extensions (SSE, AVX2, and KNC). The performance of our new algorithm is analyzed and compared to direct algorithms.

2. Algorithms

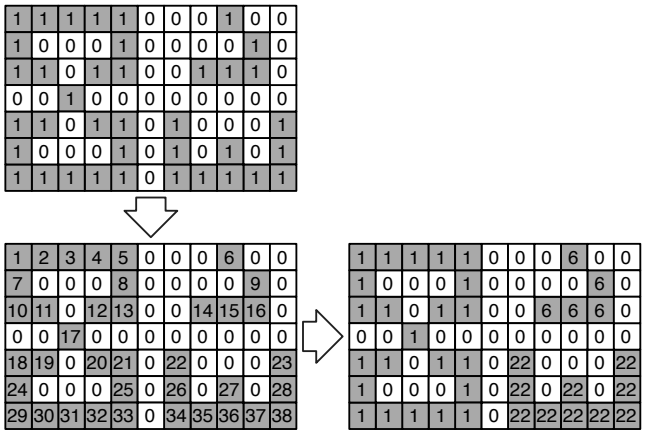


Figure 1. Labeling and $\min^+$ propagation. Top: binary image, bottom left: initial labels, bottom right: labels after propagation reaches a stabilized state

2.1 State-of-the-Art direct algorithms

Historical algorithms were designed by pioneers like Rosenfeld [24], Haralick [10] and Lumia [19] who designed pixel-based

WPMVP '16, March 13 2016, Barcelona, Spain.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2870650.2870652

algorithms, Ronse [23] for run-based algorithms. Modern algorithms derive from the algorithms of the 80's and try to make improvements by replacing some components by more efficient ones. An extensive bibliography can be found in [12] and [28]. Except Contour Tracing algorithm [6] that is aesthetic but inefficient, all modern algorithms are *direct* and require two-pass or less. None is a data-dependent multi-pass *iterative* algorithm. They share the same three steps: 1) the first labeling, that assigns a temporary/provisional label to each pixel and builds labels equivalence, 2) the label equivalences solving, that computes the transitive closure of the graph associated with the label equivalence table and 3) the final labeling (optional), to replace temporary label by the final label (usually the smallest one of the component).

They mainly differ on two points: the mask topology (pixel-based or run-based) and the equivalence management algorithm. There are two main equivalence management algorithms: the traditional Union-Find (UF) algorithm [7] usually associated to the original Rosenfeld algorithm and the Suzuki one that requires three tables [12]. As it had been shown in [5], Suzuki algorithm is not efficient for parallelized algorithms with random images (with a granularity of 1), we only use algorithms with UF management.

These *direct* algorithms – designed for CPUs – were parallelized and benchmarked on 2×12-core Intel IvyBridge Xeon and on a 4×15-core Intel IvyBridge Xeon for 2K, 4K and 8K images [4][5]. It appears that the fastest pixel-based algorithm is $HCS_2 - ARemSP$ [9] and the fastest one is the Light Speed Labeling (LSL_{RLE}) that is described in details in [16]. These two algorithms will be used in this article as a reference for direct algorithms. An important point to notice is that, unlike the *iterative* algorithms, the *direct* one cannot be SIMDized because some concurrency issues cannot be addressed with existing SIMD instruction set like voting and SIMD reduction within an array. Such an issue should be *efficiently* solved by AVX512 `conflict` instruction. Some algorithms were designed for specific architectures like Bailey's one [1][20] that uses a stack in order to avoid the non-deterministic `Find()` function. This algorithm was parallelized by Klaiber *et al.* in [14][15]. Some algorithms were also designed for GPUs [13][29] but are still slower (1 - 2 GPixels labeled per second) than FPGA ones (up to 3.2 GPixel/s) and those for general purpose processors (18 GPixel/s).

2.2 Iterative algorithms

Today, from an architecture point of view, the SIMD and multicore parallelisms are present in all processors. Then not using SIMD into an algorithm could lead to a loss of performance. Revisiting old iterative algorithms by adding them SIMD, adapting them to current architectures and transforming them into *cache aware algorithms* is a challenge as SIMD *emphasizes* the memory bandwidth issue.

For Haralick, who designed the first iterative CCL, the motivation was the amount of memory physically available in 1981 into a computer to run the algorithm. The idea of Rosenfeld to use an equivalence table and a second image of labels was not really relevant in 1966 as it required twice more memory than Haralick algorithm.

But the best argument to study again such kind of iterative algorithms may be to cite Haralick himself as his argument make sense today again. He wrote page 32 of his book [11]: "The iterative algorithm [10] uses no auxiliary storage to produce the labeled image from the binary image. It would be useful in environments whose storage is severely limited or on SIMD hardware." At this time, SIMD (Single Instruction Multiple Data, according to Flynn taxonomy) hardware is a parallel computer composed of independent processors running the same instruction at the same cycle.

2.2.1 The embarrassingly parallel iterative algorithm

This algorithm is not the first designed algorithm, but it is the most frequently encountered as it can be easily parallelized with OpenMP. This algorithm considers two images of labels, one for the input and another one for the output. It is a two-stage algorithm. The first stage consists in providing a temporary label to each non-zero pixel (fig. 1, bottom left). The second stage consists in computing the minimal positive value over a neighborhood (typically 3×3 like fig. 2) in the input image and writing this new value in the output image. All these computations are independent and can be done in parallel. The procedure is repeated (the output becomes the input) until there is no more change (fig. 1, bottom right). The parallelization is straightforward: cut the image into horizontal strips with a `# pragma omp parallel for` loop. Note that in this paper we only consider 8-connectivity (each label has eight neighbors) and not 4-connectivity. This choice has only an impact on the number of iterations that is higher in 4-connectivity.

a	b	c
d	x	e
f	g	h

Figure 2. extraction of the positive min value over a 3×3 neighborhood

This algorithm has a major drawback: the number of iterations is *data-dependent* and cannot be predicted. Figure 3 focuses on this propagation issue. For a 5×5 square, after the initial labeling, five iterations are required: four to reach the stability and another one to detect it. But for the same 5×5 square with a hole inside, the number of iterations reaches eight.

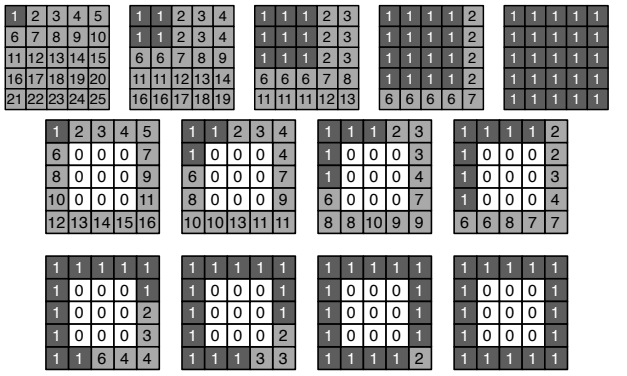


Figure 3. Impact of the shape on labels propagation. Top: 5 iterations for a full 5×5 square, bottom: 8 iterations for the same square with a hole. labels in light gray are not stabilized, labels in dark gray are stabilized

The number of iterations is directly related to the geodesic distance computation. For a convex shape, the geodesic distance of two points is the classical Euclidian distance (straight line). But for a concave one, it is the length of the shortest path inside the shape that does not cut its boundary. In other words, it is a constrained distance. The number of iterations is the longest geodesic distance between two pixels belonging to the shape plus one. Figure 4 provides four examples of geodesic distances (gd) for 5×5 shapes.

For the full square, the longest geodesic distance is the length of the diagonal (in discrete geometry).

Examples of worst cases (which set is not always countable) are a Z and a spiral, with $gd = 12$. The spiral is usually considered as the most famous worst case as for a $n \times n$ image, the number of iterations is proportional to $n^2/2$. That explains why the number of iterations cannot be evaluated and why this algorithm is not suited to real-time implementation (as its upper bound is too high) and why such algorithms are never used today.

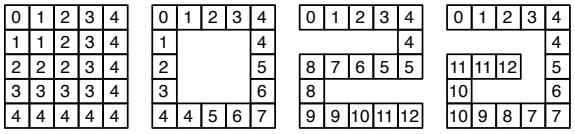


Figure 4. Geodesic distance of four 5×5-pixel shapes. From left to right: full square (gd=4), square with a hole (gd=7) a "Z" (gd=12) and a spiral (gd=12)

Some optimizations exist for this algorithm like using a larger neighborhood, but a 5×5 will only halve the number of iterations compared to a 3×3 (and will increase the stress on the bus connected to the external memory). The mathematical properties of the min^+ operator (associativity and idempotence) could also be used to factorize the operator, but it will not be enough to reach real-time processing. The most efficient transform is to make the algorithm pixel recursive.

2.2.2 The pixel recursive algorithm

The pixel recursive algorithm consists in using only one image for input and output. By writing the minimal positive value into the input image, it makes this value available for the computations related to the pixels connected to the current one. To take advantage of this information the pixels should be scanned in order, typically from left to right and from top to bottom (called *forward* scan). That creates a serialization into the algorithm (that make it unsuitable for GPUs). The positive min value cannot be propagated beyond the element of parallelization (typically a set of lines). Moreover, the propagation is no more isentropic.

Considering an image composed of only one connected component and the min positive value in x (fig. 5 left), the forward scan will propagate it in the area in gray. Such an asymmetry makes shapes that required the same number of iterations with the previous algorithm (because they have the same max geodesic distance) to require a different amount of iterations. There are four ways to scan the image : $\{$from left to right, from right to left$\} \times \{$top - down, bottom-up$\}$.

The optimal number of iterations (for unknown random images) is reached when the four scan ways are used alternatively. But for efficient cache usage, only the *forward* and the *backward* (bottom-up, from right to left) (fig. 5, right) scans will be used. Considering the scanning order, some labels of the 3×3 mask are useless as they will be used by the opposite scan. That makes the 3×3 mask (where nine labels should be read to produce a value) to be split into two masks – known as Rosenfeld masks (fig. 7) in the literature – that require only five values instead of nine to compute the positive min. One for the forward scan and one for the backward scan.

Figure 8 shows the number of iterations for the embarrassingly parallel (EP), the pixel-recursive *forward* algorithm (F) and the

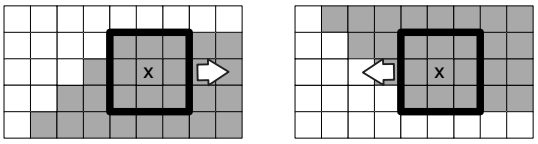


Figure 5. Instantaneous propagation of the min value in x to the whole area in gray

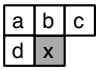


Figure 6. Extraction of the positive min value over the Rosenfeld's mask

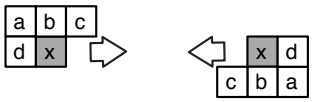


Figure 7. Forward and backward Rosenfeld masks

pixel-precursive forward-backward algorithm (FB) for an image of size $n \times n$ ($n = 128$). For $g = 1$, the max number of iterations is close to the percolation threshold ($d = 50\%$). For $d = 100\%$, the graph of the EP algorithm is close to the horizontal asymptote $iter = n$ that is the max geodesic distance (here $iter = n = 128$).

algorithm	EP		F		FB	
granularity	max	avg	max	avg	max	avg
$g = 1$	252	102	112	21.2	25	5.9
$g = 4$	232	110	95	20.5	14	4.3
$g = 16$	176	96	34	15.9	3	2.2

Table 1. Max and average number of iterations, for EP, F and FB scans, for $g \in \{1, 4, 16\}$

The Haralick original algorithm is the pixel-recursive one with Forward and Backward masks (fig. 7) and not the embarassingly parallel algorithm with the 3×3 mask.

2.2.3 A new SIMD pixel-recursive algorithm

In the following section, we only consider 128-bit integer register in order to simplify the explanation and reduce the size of the figure. The extension to 256 and 512-bit register is straightforward as all the different steps of the SIMD algorithm can be coded in each SIMD instruction set existing today.

The SIMDization of the positive min value of five labels (a, b, c, d and x) (fig. 9) is quite complex and computing the positive min of five SIMD register is not enough (fig. 10). One must propagate the positive min value within the register, taking into account the presence of zeros that stop the propagation.

This is done with a do-while loop whose exit condition written in SSE (and AVX) is the combination of a comparison and a reduction instruction that converts an SIMD register of flags into a scalar (typically `_mm_cmpeq_epi32` and `_mm_movemask_epi8`). In KNC, only one instruction is required (`_mm512_cmpeq_epi32_mask`) as the result of the comparison is directly a 32-bit integer. As a matter of fact, the SIMD code requires about fifty SIMD instructions.

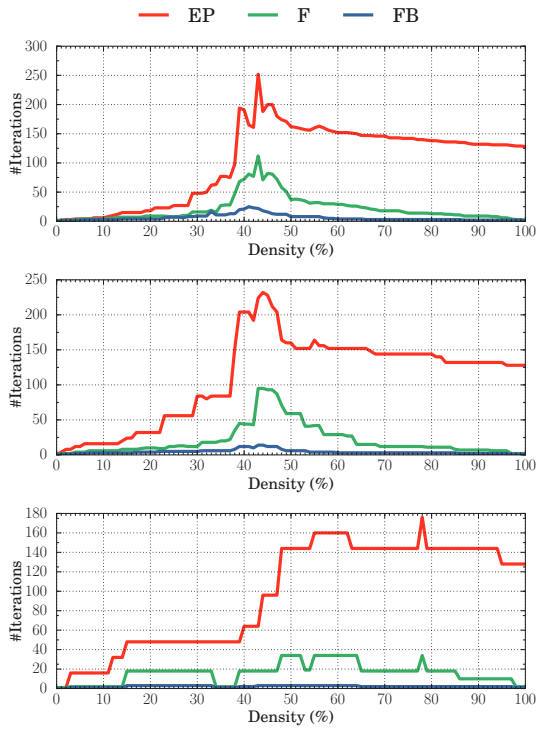


Figure 8. Number of iterations of the EP algorithm (red), the pixel-recursive F algorithm (green) and the pixel-recursive FB algorithm (blue) for 128×128 images

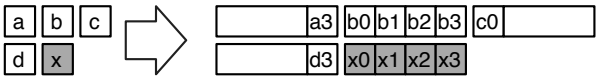


Figure 9. Forward SIMD Rosenfeld mask

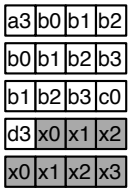


Figure 10. extraction of the positive min value of 5 SIMD registers

For a given connected component and a set of equivalence labels, the iterative algorithm can be viewed as a *greedy* algorithm that, propagates the minimum positive value over a neighborhood (typically 3×3) until stabilization.

As far as we know, this algorithm has never been published before, but we cannot claim it is new as the only difficult part is the min^+ propagation within an SIMD register. Wende published the embarrassingly parallel algorithm with KNC instruction, but do not mention any pixel-recursive algorithm [27].

In order to have only one code for all the SIMD architecture, the algorithms have been written with macro meta-programming for SSE, AVX2 and KNC (Xeon Phi Knight Corner) instruction set. These sets of macros can be easily extended for IBM Altivec or ARM Neon SIMD extension. An alternative is to use a generic SIMD library like [18] [8] [26]. Another one is to use the Intel SPMD Program Compiler (ISPC) [2].

2.2.4 The new SIMD algorithm with active tiles

Combining SIMD and OpenMP to vectorize and parallelize the algorithm can be very efficient as long as all the data fit in the cache memory hierarchy. But usually, in order to provide enough data to all cores running SIMD instructions one has to provide data that does not fit in the cache anymore. So additional optimizations must be done to enforce cache locality and to reduce the amount of cache overflow [17].

The idea – as it is an iterative algorithm – is to tile it and launch computations only if a propagation within a tile reaches one border and has to be propagated to the connected tiles. Such a strategy will reduce the amount of computations (to the only tiles that need it) but most of all, will reduce the amount of memory accesses that is the major limitation of a parallel SIMD code.

We use an *activity* matrix A that holds – for each tile – 0 if the tile is stabilized or a positive number otherwise. In order to use only one activity matrix and not two, we use two bits to encode the *stabilization* information. One bit for the tile itself and one bit for the neighboring tiles. There are two cases:

- $(00)_b$: tile is stable
- $(01)_b$: tile is unstable

Then the information of the *unstable* state is propagated (dilation), leading to four cases:

- $(00)_b$: tile and neighboring tiles are stable
- $(01)_b$: tile is unstable, neighboring tiles are stable
- $(10)_b$: tile is stable, neighboring tiles are unstable
- $(11)_b$: tile and neighboring tiles are unstable

If at least one bit is set, the tile has to be scanned. Initially, A is set to 1: all the tiles have to be scanned. Then two algorithms are applied until the whole image is stabilized ($A = 0$).

The algorithm 1 processes tiles that need to be scanned. The algorithm 2 is a sub-part of the previous one and details the processing of one tile. It corresponds to the line 4 of the first algorithm (scan tile). Then we have to update and propagate (into A) the information of which tiles should be processed again (algo. 3).

Algorithm 1: Processing all tiles

```
1 foreach tile t(i_t, j_t) do
2     if A(i_t, j_t) ≠ (00)_b then
3         A(i_t, j_t) ← (00)_b
4         scan tile t(i_t, j_t)
5         if t is not stabilized then
6             A(i_t, j_t) ← (01)_b
```

This tiling enables load balancing: instead of scanning a tile until its stabilization, there is a fixed number of scans (here equal to 2). In the following, we will only consider the FB scan for efficiency reasons.

Algorithm 2: scan tile $t(i_t, j_t)$ of coordinates $[i_0, i_1] \times [j_0, j_1]$

```
1  flag ← (00)_b
2  for (i = i_0, i ≤ i_1; i++) do
3      for (j = j_0, j ≤ j_1; j++) do
4          a ← E(i−1, j−1), b ← E(i−1, j)
5          c ← E(i−1, j+1), d ← E(i, j−1)
6          x ← E(i, j)
7          x' ← min+(a, b, c, d, x)
8          E(i, j) ← x'
9          flag ← (flag or (x xor x'))
10 for (i = i_1, i ≥ i_0; i−−) do
11     for (j = j_1, j ≥ j_0; j−−) do
12         a ← E(i+1, j+1), b ← E(i+1, j)
13         c ← E(i+1, j−1), d ← E(i, j+1)
14         x ← E(i, j)
15         x' ← min+(a, b, c, d, x)
16         E(i, j) ← x'
17         flag ← (flag or (x xor x'))
18 return flag
```

Algorithm 3: information propagation

```
1  foreach cell A(i_t, j_t) do
2      if (A(i_t, j_t) and (01)_b) = (01)_b then
3          A(i_t−1, j_t−1) ← [A(i_t−1, j_t−1) or (10)_b]
4          A(i_t−1, j_t+0) ← [A(i_t−1, j_t+0) or (10)_b]
5          A(i_t−1, j_t+1) ← [A(i_t−1, j_t+1) or (10)_b]
6          A(i_t−0, j_t−1) ← [A(i_t−0, j_t−1) or (10)_b]
7          A(i_t−0, j_t+1) ← [A(i_t−0, j_t+1) or (10)_b]
8          A(i_t+1, j_t−1) ← [A(i_t+1, j_t−1) or (10)_b]
9          A(i_t+1, j_t+0) ← [A(i_t+1, j_t+0) or (10)_b]
10         A(i_t+1, j_t+1) ← [A(i_t+1, j_t+1) or (10)_b]
```

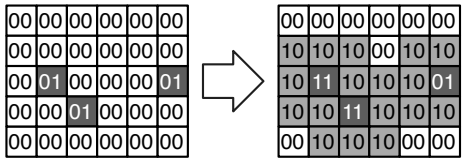


Figure 11. Example of dilation: left 1-bit A matrix before dilation, right: 2-bit A matrix after dilation

We used OpenMP2 to parallelize the tiles processing (with an `#omp parallel for` on a 1D *view* of the 2D matrix A). OpenMP3 (tasking) or OpenMP4 (tasking with activation linked to array dependency) or TBB can also be used. Note that TBB implements the *workpile* pattern [22] that is the processing model used here.

3. Benchmarks and analysis

3.1 Targeted SIMD machines

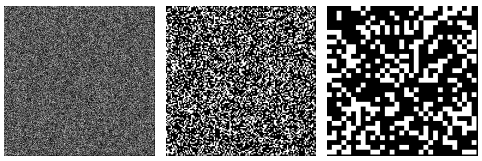
Three SIMD extensions are used: SSE on Nehalem and Ivy-Bridge processors, AVX2 on Haswell processors and KNC on Xeon-Phi Knight Corner processor. Intel C compiler (icc 15 and 16.1) is used. The latest release does not generate faster SIMD code and does not vectorize scalar code because an *anti dependence* between lines 4, 5 and 8 of algorithm 2. The table 2 provides the specifications of the benchmarked machines with their *theoretical* peak performance (in giga operations per second) and peak bandwidth (in gigabytes per second). Two other parameters are calculated: π and C/BW ratio. The value of π is the product of processor's parallelisms: the core number multiplied by the SIMD cardinal (here, the number of 32-bits integer within a register). The C/BW ratio is the peak performance divided by the peak external memory bandwidth. It is usually used to characterize an algorithm, in that case it is the number of computations divided by the number of memory accesses (the lower, the more the algorithm is memory-bound). Here C/BW is the ratio of the peak performance by the peak bandwidth of the processor. A high ratio is interesting for processors handling algorithms with many computations and few memory accesses, while a low ratio indicates that the processor is not too sensitive (or less sensitive) to memory bound algorithms.

3.2 Benchmark procedure

Usually, papers evaluate CCL performance first with random images (varying pixel density from 0% to 100%) for hard-to-label benchmarks and secondly with image database. As we want our benchmark to be as fair as possible (quite difficult with data-dependent algorithms) we decided to select Mersenne Twister MT19937 [21] to control the random number generation and to extend random images by changing the pixel granularity.

The initial random image has a granularity of 1. Then we create g-random images whose blocks of pixels have a size of $g \times g$ (Fig. 12), with $g \in [1 : 16]$. The pixel block is set to 1 if the random value is smaller or equal to the density d and set to zero otherwise.

This methodology highlights some algorithm behavior linked to the number of labels and to the image density. An important point is that we propose a reproducible benchmark procedure [25]. As the random number generator is not the `rand` function provided into the `libC` library, but MT19937 generator with *seed* equal to zero, our procedure can be exactly reproduced by any reader.



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

Figure 12. Random images with density = 35% at granularity $g \in \{1, 4, 16\}$

For "small" machines, we use 2K images and 4K for big ones, to ensure that each core has *enough* data to process. As the number of cores of each processor is different, the performance of the machines can not be easily compared. Explanations are by far not straightforward. That is why we focus on the comparison of direct versus iterative algorithms on each machine separately.

The execution time is measured in cycles per point (*cpp*), as processors have variable frequency. Average *cpp* is the average, for a given granularity g of the *cpp* for all evaluated density $d \in [0\%, 100\%]$ with a step of 1%. The *cpp* are computed for all values $g \in [1 : 16]$, but we provide results only for $g \in \{1, 4, 16\}$ in order to provide synthesis results as *cpp* varies continuously with g. For the same reason, we provide only one value – the *arithmetic* mean – *cpp* for a given g and d. People interested in the evolution of the *cpp* with the density can read [3][4][5].

processors	acronym used	freq (GHz)	cache (MB)	nb cores	SIMD	π parallelism	peak perf. Gops	peak BW GB/s	C / BW ratio
Nehalem X5550	NHM_8	2.67	2×8	2×4	SSE 4.2	32	85.1	64.0	1.3
IvyBridge E5-2697v2	IVB_{24}	2.66	2×30	2×12	AVX 1	96	255.4	119.4	2.1
Haswell-EP E5-2697v3	HSW_{28}	2.6	2×14	2×14	AVX 2	224	582.4	136.0	4.3
Xeon Phi 3120A	KNC_{57}	1.1	1×28.5	1×57	KNC	912	1003.2	240.0	4.2

Table 2. Main characteristics of the evaluated machines.

To analyze the impact of the tile size, we have tested all power of two from 2^4 up to 2^{11}. For SIMD implementations, the min tile width is the cardinal of an SIMD register, that is 4, 8, 16 for respectively SSE, AVX2, and KNC.

3.3 Results

Tables 3, 4, 5 and 6 present the results for the four machines, for direct and iterative algorithms with/without SIMD.

	$g=1$	$g=4$	$g=16$	mean
cpp of direct algorithms				
LSL_{RLE}	13.167	5.233	3.559	7.32
$HCS2$	13.800	7.644	6.260	9.23
LSL_{RLE}+OMP	2.157	1.139	0.969	1.42
$HCS2$+OMP	3.080	2.342	2.242	2.55
cpp of iterative FB min$^+$ algorithm				
scalar	1028	294.0	105.7	475.90
SIMD	243.3	137.0	67.56	149.29
scalar+OMP	169.2	78.92	55.89	101.34
SIMD+OMP	44.46	33.15	26.89	34.83
activity+SIMD+OMP	27.06	19.29	15.96	20.77
cpp of iterative FB max algorithm				
SIMD+OMP	59.84	42.60	38.52	46.99
activity+SIMD+OMP	21.84	13.74	11.91	15.83

Table 3. Performance of Nehalem NHM_8 for 2K images

	$g=1$	$g=4$	$g=16$	mean
cpp of direct algorithms				
LSL_{RLE}	13.81	5.43	3.19	7.48
$HCS2$	14.09	7.57	6.17	9.28
LSL_{RLE}+OMP	1.67	0.995	0.854	1.17
$HCS2$+OMP	3.43	2.312	2.096	2.61
cpp of iterative FB min$^+$ algorithm				
scalar	1317	446.6	176.8	646.80
SIMD	348.4	175.3	87.11	203.60
scalar+OMP	95.01	40.09	26.25	53.78
SIMD+OMP	28.28	21.14	17.12	22.18
tile+SIMD+OMP	15.86	11.40	9.39	12.22
cpp of iterative FB max algorithm				
SIMD+OMP	73.70	57.34	51.21	60.75
activity+SIMD+OMP	10.16	6.36	5.46	7.33

Table 4. Performance of Ivy-Bridge IVB_{24} for 4K images

3.4 Direct algorithms analysis

The results are split into two sets. As long as the machines have few cores and a low C/BW ratio, the ratio between the algorithms LSL and $HCS2$ is alike, $\times 2.3$ with OpenMP for NHM_8 and $\times 2.6$ with OpenMP for IVB_{24}. But when the machines have lots of cores and a high C/BW ratio, the behaviors change: the LSL keeps on accelerating whereas $HCS2$ does not. The ratios become $\times 6.9$ with OpenMP for HSW_{28} and $\times 35.6$ for KNC_{57}. As the KNC's cores are not designed neither for scalar nor single-threaded computations, there is almost one order of magnitude between HSW_{28} and KNC_{57}.

The explanation comes from the intrinsic algorithm design: LSL uses run-length encoding (RLE) to compress data and reduce the amount of memory accesses (except for $g = 1$ where it has the same performance than $HCS2$), while $HCS2$ is a pixel-based algorithm that requires much more memory accesses than LSL. This result has been already observed on bi-socket and quad-socket Xeon Ivy-Bridge [4], but neither on Haswell nor Knight Corner. This is a clear evidence that none of all pixel-based CCL algorithm scales on multi and manycore processors.

	$g=1$	$g=4$	$g=16$	mean
cpp of direct algorithms				
LSL_{RLE}	12.840	4.492	2.645	6.66
$HCS2$	13.195	6.695	5.620	8.50
LSL_{RLE}+OMP	1.317	0.357	0.249	0.64
$HCS2$+OMP	2.524	2.249	1.711	2.16
cpp of iterative FB min$^+$ algorithm				
scalar	1180.2	360.5	140.17	560.29
SIMD	190.4	360.5	50.67	200.52
scalar+OMP	78.73	44.24	45.79	56.25
SIMD+OMP	63.24	32.32	29.13	41.56
activity+SIMD+OMP	7.78	4.50	3.94	5.41
cpp of iterative FB max algorithm				
SIMD+OMP	28.73	22.21	18.30	23.08
activity+SIMD+OMP	5.42	3.95	3.21	4.19

Table 5. Performance of Haswell HSW_{28} for 4K images

	$g=1$	$g=4$	$g=16$	mean
cpp of direct algorithms				
LSL_{RLE}+OMP	2.816	1.721	1.460	2.00
$HCS2$+OMP	52.640	52.070	51.971	52.23
cpp of iterative FB min$^+$ algorithm				
scalar+OMP	170.4	141.5	126.0	145.97
SIMD+OMP	26.87	26.43	25.76	26.35
activity+SIMD+OMP	6.04	4.08	2.86	4.33
cpp of iterative FB max algorithm				
SIMD+OMP	30.88	29.40	27.47	29.25
activity+SIMD+OMP	5.23	3.30	2.46	3.66

Table 6. Performance of Knight Corner KNC_{57} for 4K images

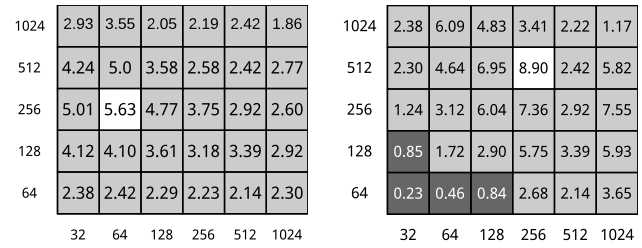


Figure 13. Hotmaps of ratio between algorithms with tiling / version without tiling for $g = 4$ (the higher, the better) for HSW_{28} and KNC_{57}. Gray legend: dark gray = slower, light gray = faster, white = fastest

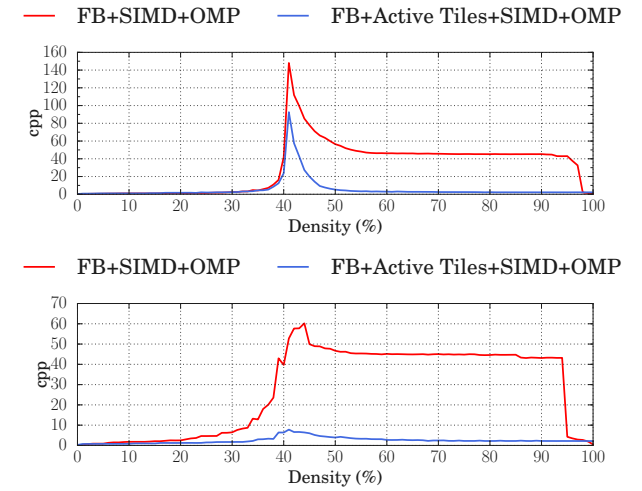


Figure 14. *cpp* for SIMD+OpenMP version with and without activity for KNC, for $g = 1$ (top) and $g = 16$

3.5 Iterative algorithms analysis

Same kind of observations can be done for the FB iterative algorithm without tiling. For machines with a low C/BW ratio, both OpenMP and SIMD provide a speedup. But, as the combination of OpenMP and SIMD generates a lot of stress on the external memory busses, the speedups are not perfect and the tiling only provides a small additional speedup: $\times 1.7$ for NHM_8 and $\times 1.8$ for IVB_{24}. For high C/BW ratio machines, the impact of tiling is significant: $\times 7.4$ for HSW_{28} and $\times 9.0$ for KNC_{57}.

As an illustration of the impact of active tiles, the figure 13 shows the ratio between the algorithms with active tiles (for 30 tile sizes) and the algorithm without ones. With active tiles, HSW_{28} is up to $\times 5.63$ faster than the algorithm without tiling. This ratio rises $\times 8.90$ for KNC_{57}.

Unlike the classic FB algorithm (without active tiles) – where data are split into horizontal strips by OpenMP – the active tiles algorithm has a better cache use and re-use, as two iterations (one forward and one backward) are done on a set of labels that fit in the core cache.

Max propagation have been implemented and benchmarked, as it makes the codes smaller (less comparisons than for $\min^+$ computation). It appears that for SIMD+OMP version without active tiles, the max propagation is slower than the min propagation whereas for versions with active tiles, result is the opposite. Right now we have no explanation of this behavior.

For each architecture and each image size, it appears that the best tile is quite always the same. So for a real application, an autotuning step can be added to set up the optimal parameters.

3.6 Direct versus iterative algorithms comparison

If we now compare direct algorithms with the iterative one with active tiles, the same two sets still exist. For NHM_8 and IVB_{24} the direct algorithm $HCS2$ is faster: respectively $\times 7.1$ and $\times 3.6$, but for a machine with a lower C/BW ratio and more cores, the ratio is smaller: only $\times 2.3$ on HSW_{28}. For the KNC_{57} the ratio is not significative as $HCS2$ is a scalar algorithm and FB + active tiles is SIMD. But we can notice that the average *cpp* of KNC_{57} is smaller than HSW_{28}: 2.46 versus 3.94. In comparison, the *cpp* without tiling were respectively 27.5 and 18.3.

In conclusion, concerning connected component labeling, the pixel-based algorithms do not scale on State-of-the-Art architectures. The reasons are that direct algorithms require two much bandwidth (that generates too much stress on external memory) and a pyramidal transitive closure to merge labels leading to a weak parallelism. One solution is data compression, that LSL does, but on a more parallel machine – like the upcoming quadri-socket Xeon Broadwell-EX – it could be limited by the pyramidal merge for small image (if each core has few data to proceed). Another one is tiling combined with an activity matrix. It save external memory accesses and useless computations (for temporary stabilized tiles).

3.7 Anticipate the algorithm scalability on future processors

Let us try to anticipate the future performance with the Amdahl law (1), where sp is the speedup, p the parallelism (π in table 2) and τ the fraction of sequential code that cannot be accelerated.

$$sp = \frac{1}{\tau + \frac{1-\tau}{p}} \quad \Rightarrow \quad \tau = \frac{\pi - sp}{sp(\pi - 1)} \tag{1}$$

processors	NHM_8	IVB_{24}	HSW_{28}
LSL_{RLE}	7.9 %	12 %	6.3 %
HCS2	17.3%	25.0 %	22.6 %
FB	4.3 %	2.4 %	7 %
FB + activity	1.3 %	0.9 %	0.5 %

Table 7. Average fraction τ of sequential code according to Amdahl's law (lower is better)

The general interpretation of Amdahl's law is that a whole code cannot be accelerated. In our case, the codes are globally parallelized and SIMDized, even if there are very few instructions outside parallel loops and the fact that OpenMP requires an overhead to create threads. Here, the lack of scalability is due to the combination of OpenMP+SIMD puts too much stress on the external memory. All SIMD cores cannot run in parallel at full speed and must wait for the data. The value of τ reflects the algorithm scalability and the stress on memory.

We can see (tab. 7) that the proposed algorithm (pixel-recursive *Forward-Backward* with *tiling* and SIMD + OpenMP has always the smallest τ value and so the highest scalability. The τ value is not relevant for KNC_{57} because, as previously written, both single-threaded and scalar codes performance are not significant on this architecture. We can envision that it could match and outperform

$HCS2$ the best pixel-based algorithm because their *cpp* are already close and because it should continue to scale, unlike the pixel-based ones.

4. Conclusion

In this paper, we have presented a new parallel and tiled SIMD algorithm for connected component labeling that uses an activity matrix to signal what are the tiles that need to be processed. Some macro meta-programming was used to get the same SIMD code to run on SSE, AVX2, and KNC SIMD processors. The tiling makes the algorithm efficient for architecture combining wide SIMD and lots of cores. Right now, the answer to the question stated in the introduction "does a brute-force iterative algorithm can match a direct algorithm ?" is still no. But the results of the benchmarks done on four generations of processors – Nehalem, Ivy-Bridge, Haswell and Knight Corner – show that the gap, initially very large is getting smaller and smaller. On a 2×14-core 256-bit SIMD Haswell the iterative algorithm is only 1.5 slower than the fastest pixel-based algorithm. On a 57-core 512-bit SIMD Knight Corner, the processing requires fewer cycles than on Haswell. That clearly shows that if the answer was *no* for many years, it could be different in few years when general purpose processors and specialized ones will increasingly have more cores.

In future works, we will focus on AVX-512 specific instructions – available for Xeon Skylake and Knight Landing processors – to design more efficient CCL algorithms. KNC provides masked-instructions and sparse memory accesses through scatter-gather instructions but AVX-512 will provide additional instructions to tackle some concurrency issues like parallel vote with SIMD register. We will also evaluate GPU implementation and manycore processors available into research laboratories.

Acknowledgements

The authors would like to thank, Francois Hannebicq from Intel France, for the access to State-of-the-Art machines and Zakhar A. Matveev from Intel Russia, for its valuable help on Xeon-Phi.

References

[1] D. Bailey and C. Johnston. Single pass connected component analysis. In *Image and Vision New Zeland (IVNZ)*, pages 282–287, 2007.

[2] J. Brodman, D. Babokin, I. Filippov, and P. Tu. Writing scalable SIMD programs with ISPC. In *Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 25–32. ACM, 2014.

[3] L. Cabaret and L. Lacassagne. What is the world's fastest connected component labeling algorithm ? In *IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 97–102, 2014.

[4] L. Cabaret, L. Lacassagne, and D. Etiemble. Parallel Light Speed Labeling: an efficient connected component labeling algorithm for multi-core processors. In *IEEE International Conference on Image Processing (ICIP)*, pages 1–4, 2015.

[5] L. Cabaret, L. Lacassagne, and D. Etiemble. Parallel Light Speed Labeling for connected component analysis on multi-core processors. *Journal of Real Time Image Processing*, *To be published*:1–18, 2016.

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

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

[8] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté. Boost.SIMD: generic programming for portable SIMDization. In *Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 1–8. ACM, 2014.

[9] 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.

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

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

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

[13] W. W. Hwu, editor. *GPU Computing Gems*, chapter 35: Connected Component Labeling in CUDA. Morgan Kaufman, 2001.

[14] M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, and S. Simon. A memory-efficient parallel single pass architecture for connected component labeling of streamed images. In *International Conference on Field Programmable Technology (FPT)*, pages 159–165. IEEE, 2012.

[15] M. Klaiber, D. Bailey, S. Ahmed, Y. Baroud, and S. Simon. A high-throughput FPGA architecture for parallel connected components analysis based on label reuse. In *International Conference on Field Programmable Technology (FPT)*, pages 302–305. IEEE, 2013.

[16] 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.

[17] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle. High level transforms for SIMD and low-level computer vision algorithms. In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 49–56, 2014.

[18] R. Leissa, I. Haffner, and S. Hack. Sierra: a SIMD extension for C++. In *Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 17–24. ACM, 2014.

[19] 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.

[20] N. Ma, D. Bailey, and C. Johnston. Optimised single pass connected component analysis. In *International Conference on Field Programmable Technology (FPT)*, pages 185–192. IEEE, 2008.

[21] 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.

[22] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: patterns for efficient computation*. Morgan Kaufmann, 2012.

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

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

[25] P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible research in signal processing. *Signal Processing Magazine*, 26,3:37–47, 2009.

[26] H. Wang, P. Wu, I. Tanase, M. Serrano, and J. Moreira. Simple, portable and fast SIMD intrinsic programming: generic simd library. In *Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 9–16. ACM, 2014.

[27] F. Wende and T. Steinke. Swendsen-wang multi-cluster algorithm for the 2d/3d Ising model on Xeon Phi and GPU. technical report 13-44, Zuse Institue Berlin, 2013.

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

[29] G. Ziegler. Connected components revisited on Kepler. In Nvidia, editor, *GPU Technology Conference*, pages 1–56, 2013.