Designing efficient SIMD algorithms for direct Connected Component Labeling

Arthur Hennequin[1,2], Ian Masliah[1], Lionel Lacassagne[1]
firstname.name@lip6.fr
LIP6[1] – Sorbonne University, CNRS
LHCb experiment[2] – CERN

ABSTRACT

Connected Component Labeling (CCL) is a fundamental algorithm in computer vision, and is often required for real-time applications. It consists in assigning a unique number to each connected component of a binary image. In recent years, we have seen the emergence of direct parallel algorithms on multicore CPUs, GPUs and FPGAs whereas, there are only iterative algorithms for SIMD implementation. In this article, we introduce new direct SIMD algorithms for Connected Component Labeling. They are based on the new Scatter-Gather, Collision Detection (CD) and Vector Length (VL) instructions available in the recent Intel AVX512 instruction set. These algorithms have also been adapted for multicore CPU architectures and tested for each available SIMD vector length. These new algorithms based on SIMD Union-Find algorithms can be applied to other domains such as graphs algorithms manipulating Union-Find structures.

1 INTRODUCTION

Connected Component Labeling (CCL) is a central algorithm between low-level image processing (filtering and pre-processing) and high-level image processing (scene analysis, pattern recognition and decision). CCL consists in providing a unique number to each connected component of a binary image. There are several applications in computer vision that use CCL such as Optical Character Recognition, motion detection or tracking.

CCL algorithms are often required to be optimized to run in real-time and have been ported on a wide set of parallel machines [1][13]. After an era on single-core processors, where many sequential algorithms were developed [7] and codes were released [2], new parallel algorithms were developed on multicore processors [15][6].

The majority of CCL algorithms for CPUs are direct, by opposition to iterative ones – where the number of iterations (the number of image pass to process it) depends on the image structure – and require only 2 passes thanks to an equivalence table.

The first algorithms designed for GPUs [10][9][5] and SIMD [19][12] were iterative. Furthermore, the number of iterations required to reach a solution cannot be predicted beforehand and can reach a large number. In recent years, some direct algorithms for GPUs were designed and are better suited for real-time image processing in embedded systems [11][3][16][8].

In this article, we introduce a new direct CCL algorithm for SIMD processors which relies on *scatter-gather* operations. These algorithms are only available using Intel AVX512 extension and the upcoming ARM/Fujitsu SVE.

Section 2 explains algorithmic design and the required SIMD instructions. Section 3 describes the classic Rosenfeld algorithm [18] and its derivatives, as well as concurrency issues that can appear using SIMD. Section 4 introduces two new SIMD algorithms with implementation details. The first one is based on a classic pixel-based approach, while the second one is based on sub-segments and uses bit manipulation and conflict detection to reduce memory accesses. Section 5 presents the benchmark protocol for reproducible experiments, the results and their analysis.

2 ALGORITHMIC DESIGNS TO EXPLOIT ARCHITECTURES

As clock frequencies of modern processors are expected to stay near their current levels, or even to get lower, the primary method to improve the computational capability of a chip is to increase either the number of processing units (cores) or the intrinsic parallelism of a core (SIMD). In this work, we target the most recent Intel architecture with the AVX512 SIMD extension set. The AVX512F extension has added the scatter instruction for each available vector size (128, 256, 512) which is required to develop an SIMD Rosenfeld algorithm. We have used the instructions *compress* and *expand* which are available in the AVX512VL extension (algo 8). From AVX512CD, we also used the *conflict detection* (CD) and *lzcnt* (count leading zeros) instructions (algo 11).

Each of these instructions are required to implement efficient SIMD CCL algorithms and are only available in the recent Intel AVX512 extension set. Developing new algorithms that use the architecture efficiently requires heavy algorithmic modifications and can only be done efficiently if the correct instructions are available. Due to this complex process, it is impossible for a compiler to provide any vectorization support. Domain Specific Languages (DSLs) such as Halide [17] would also require additional modifications from the DSL and the application programmer due to the new available instructions.

In this context, we developed a set of AVX512 C++ templates code that can be instanciated in 128-bit, 256-bit and 512-bit in order to evaluate the impact of SIMD width on the performance. The required sub-sets of AVX512 are AVX512F, AVX512CD and AVX512VL. We have further extended our portable template code to support the new ARM SVE instruction set but the code has not been tested in the scope of this paper.

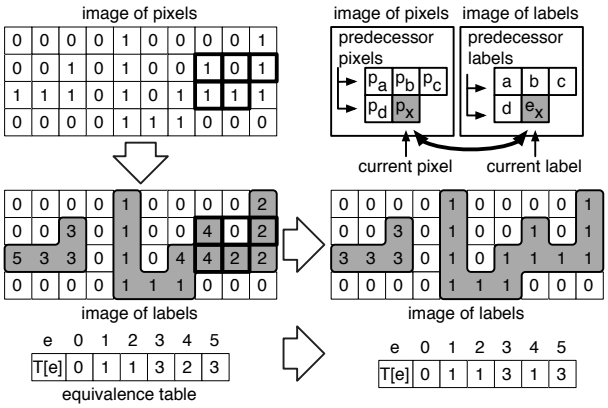


Figure 1: Example of 8-connected CCL with Rosenfeld algorithm: binary image (top), image of temporary labels (bottom left), image of final labels (bottom right) after the transitive closure of the equivalence table.

3 CLASSIC ALGORITHMS

Usually, CCL algorithms are split into three steps and perform two image scans (like the pioneer algorithm of Rosenfeld [18]). The first scan (or first labeling) assigns a temporary/provisional label to each connected component (CC) and some label equivalences are built if needed. The second step solves the equivalence table by computing the transitive closure (TC) of the graphs (in fact a forest) associated to the label equivalences. The third step performs a second scan (or second labeling) that replaces temporary labels of each CC by its final label.

Figure 1 defines some notations and gives an example of a classic Rosenfeld algorithm execution. Let p_x, e_x, the current pixel and its label. Let p_a, p_b, p_c, p_d, the neighbor pixels, and a, b, c, d, the associated labels. T is the equivalence table, e a label and r its root. The first scan of Rosenfeld is described in algorithm 3, the transitive closure in algorithm 4, while the classical union-find algorithms are provided in algorithms 1 & 2. In the example (Fig. 1), we can see that the rightmost CC requires three labels (1, 2 and 4). When the mask is in the position seen in figure 1 (in bold type), the equivalence between 2 and 4 is detected and stored in the equivalence table T. At the end of the first scan, the equivalence table is complete and applied to the image (like a Look-Up-Table).

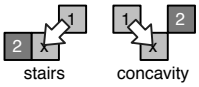


Figure 2: Basic patterns generating an equivalence: stairs & concavity

3.1 Algorithmic and SIMD optimisation

Decision Tree (DT) based algorithms for CCL [20] have been proved to be very efficient to enhance scalar implementations. The DT reduces the number of Union and Find function calls to the strict minimum, i.e. when there is an equivalence between two different

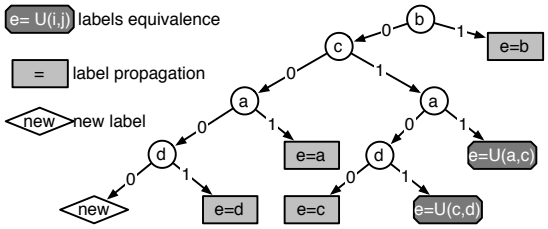


Figure 3: 8-connected Decision Tree for a 4-pixel mask. Labels equivalence (call to Union) in dark gray

Algorithm 1: Find(e, T)

Input: e a label, T an equivalence table
Result: r, the root of e

```
r ← e
while T[r] ≠ r do
    r ← T[r]
return r
```

Algorithm 2: Union(e_1, e_2, T)

Input: e_1, e_2 two labels, T an equivalence table
Result: r, the least common ancestor of the e's

```
r1 ← Find(e1)
r2 ← Find(e2)
if r1 < r2 then
    r ← r1, T[r2] ← r
else
    r ← r2, T[r1] ← r
return r
```

Algorithm 3: Rosenfeld algorithm – first labeling (step 1)

Input: a, b, c, d, four labels, p_x, the current pixel in (i, j)

```
if p_x ≠ 0 then
    a ← E[i−1][j−1],   b ← E[i−1][j]
    c ← E[i−1][j+1],   d ← E[i][j−1]
    if (a = b = c = d = 0) then
        ne ← ne + 1,   e_x ← ne
    else
        r_a ← Find(a, T),   r_b ← Find(b, T)
        r_c ← Find(c, T),   r_d ← Find(d, T)
        e_x ← min+(r_a, r_b, r_c, r_d)
        if (r_a ≠ 0 and r_a ≠ e_x) then Union(e_x, r_a, T)
        if (r_b ≠ 0 and r_b ≠ e_x) then Union(e_x, r_b, T)
        if (r_c ≠ 0 and r_c ≠ e_x) then Union(e_x, r_c, T)
        if (r_d ≠ 0 and r_d ≠ e_x) then Union(e_x, r_d, T)
else
    e_x ← 0
```

labels. There are only two patterns generating an equivalence between labels: stairs and concavities which are depicted in figure 2. DT is more efficient than path-compression as it reduces the number of memory accesses. Considering the classical implementation of the Rosenfeld algorithm (algo. 3), there are four calls to Find and one to Union. The calls to find in the union can be omitted because the input labels are already equivalence trees' roots. Using a DT (algo. 5), there are at most one call to Union and two calls to Find as we have at most one equivalence between labels.

If we consider the extended topology of the mask in the SIMD Union Find (Fig. 4), there are, for an SIMD vector of cardinality *card*,

Algorithm 4: Sequential solve of equivalences (step 2)

```
for e ∈ [1 : ne] do
    T[e] ← T[T[e]]
```

Algorithm 5: Rosenfeld with DT – optimized first labeling (step1)

```
Input: a, b, c, d, four labels, p_x, the current pixel in (i, j)
if p_x ≠ 0 then
    b ← E[i − 1][j]
    if (b ≠ 0) then
        e_x ← b
    else
        c ← E[i − 1][j + 1]
        if (c ≠ 0) then
            a ← E[i − 1][j − 1]
            if (a ≠ 0) then
                e_x ← U(a, c)
            else
                d ← E[i][j − 1]
                if (a ≠ 0) then
                    e_x ← U(c, d)
                else
                    e_x ← c
        else
            a ← E[i − 1][j − 1]
            if (a ≠ 0) then
                e_x ← a
            else
                d ← E[i][j − 1]
                if (a ≠ 0) then
                    e_x ← d
                else
                    n_e ← n_e + 1
                    e_x ← n_e
else
    e_x ← 0
```

at most $card/2 + 1$ different labels, $card/2$ calls to Union, and thus, $card$ calls to Find, with $card/2 - 1$ which are redundant. If we take a vector with a cardinal of 8, we get $1/2$ Union per pixel and one call to Find. In the classical Union Find approach, we first perform a call to $Union(e_0, e_1)$ which calls $Find(e_0)$ and $Find(e_1)$. Then, we call $Union(e_1, e_2)$ which calls $Find(e_1)$ and $Find(e_2)$, and so on, until a call to $Union(e_3, e_4)$ which calls $Find(e_3)$ and $Find(e_4)$. As there can be more than one Union in an SIMD vector, concurrency issues can appear.

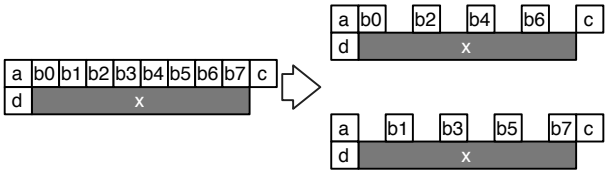


Figure 4: SIMD Union Find: number of different labels is equal to $card/2 + 1$ (here $card = 8$)

3.2 Algorithmic and SIMD concurrency issue

Depending on the neighboring labels of a pixel, concurrency issues can appear while performing simultaneous union operations. Figure 5 provides some examples (and the associated graphs) built with the basic patterns of figure 2. In the first row, there are no concurrency issues. There is one CC on the left ($T[3] \leftarrow 1, T[4] \leftarrow 1$)

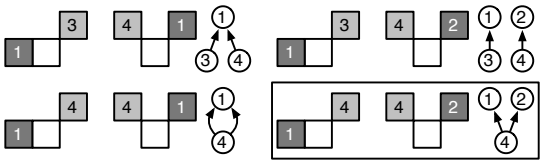


Figure 5: Composition of basic patterns leading to redundant accesses (bottom left) and concurrency issue (bottom right)

and two CCs on right ($T[3] \leftarrow 1, T[4] \leftarrow 2$).

In the second row, there is one CC on the left with twice the same equivalence ($T[4] \leftarrow 1, T[4] \leftarrow 1$) which is redundant but not an issue. However, on the right happens a concurrency, as we have $4 \equiv 1$ and $4 \equiv 2$. A possible correction could be $2 \equiv 1$ ($T[4] \leftarrow 1$, $T[2] \leftarrow 1$). Note that this problem can happen starting from a 3-pixel wide SIMD register if the two patterns are merged. The SIMD implementation of Union Find has to address these concurrency issues, which is – by far – non trivial to solve and to implement efficiently in SIMD.

4 NEW ALGORITHMS

4.1 SIMD Union-Find

The biggest challenge when designing an SIMD CCL algorithm is to design a fast and concurrency-free union-find algorithm to manage equivalences. The union algorithm must take into account the conflicts from multiple equivalence tree roots involved in simultaneous merge operations. Figure 6 shows an example of such complex merges. The top row shows the evolution of the root label pointers and the bottom row shows the pending merge operations in black and the finished merge operations in grey.

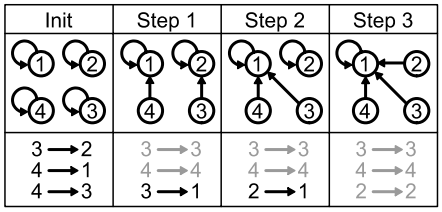


Figure 6: Execution of algorithm 7 *VecUnion* with arguments $\vec{e_1} = [3, 1, 2], \vec{e_1} = [4, 4, 3]$ (example of simultaneous unions in 3 steps and their serialization).

Algorithms presented in this section and the followings are written for a cardinality of 8 (for the sake of clarity) but can easily be extended to 4 or 16 elements. Unmasked equalities and inequalities between vectors are tested using the intrinsics *cmpeq_epi32_mask* and *cmpneq_epi32_mask*, but are written as mathematical comparison to be more readable. Masked comparisons are expressed using their corresponding intrinsic.

Algorithm 6 uses gather loads to find the roots of all labels in $\vec{e}$. The loop must run until all roots have been found, so the number of iterations is equal to the maximum distance between involved

labels and their roots.

Algorithm 6: VecFind($\vec{e}$, T, m)

Input: $\vec{e}$ a vector of label, T an equivalence table, m a mask
Result: $\vec{r}$, the roots of $\vec{e}$

$\vec{r} \leftarrow \vec{e}$
$done \leftarrow 0$ // mask
while $done \neq m$ **do**
  $\vec{l} \leftarrow$ mask_i32gather_epi32($\vec{r}$, $\neg done \wedge m$, $\vec{r}$, T, sizeof(uint32))
  $done \leftarrow$ mask_cmpeq_epi32_mask($\vec{l}$, $\vec{r}$, m) // mask
  $\vec{r} \leftarrow \vec{l}$
return $\vec{r}$

Algorithm 7 is inspired by the Playne-Equivalence reduce function designed for parallel CCL on GPU [16]. The main difference with a GPU algorithm is that we have to consider the parallelism within an SIMD vector instead of the parallelism between GPU threads in memory. To solve the concurrency issue the same way the GPU does, we introduce the *VecScatterMin* function that emulates the behavior of the CUDA *atomicMin* function. This function takes two vectors $\vec{idx}$ and $\vec{val}$ and tries to perform the operation: $T[\vec{idx}] \leftarrow min(T[\vec{idx}], \vec{val})$. It then returns the old value of $T[\vec{idx}]$ if the operation succeeded or the current value if another vector element has written it first. Using this function, only one value is written to a memory address at a time, but it is always the minimum value of the concurrent store operations. This allows the *VecUnion* operation to retry until all involved equivalence trees have been merged. Because of the pixel topology, there can be at most $card/2$ simultaneous equivalence tree merges. In practice, the merge vectors are very sparse, allowing us to reduce the numbers of operations needed by compressing the vectors. The *VecScatterMin* function is described in Algorithm 8.

Algorithm 7: VecUnion($\vec{e_1}$, $\vec{e_2}$, T, m)

Input: $\vec{e_1}$, $\vec{e_2}$ two vectors of labels, T an equivalence table, m a mask

$\vec{r_1} \leftarrow$ VecFind($\vec{e_1}$, T, m)
$\vec{r_2} \leftarrow$ VecFind($\vec{e_2}$, T, m)
$m \leftarrow$ mask_cmpneq_epi32_mask(m, $\vec{e_1}$, $\vec{e_2}$) // mask
while m **do**
  $\overrightarrow{r_{max}}, \overrightarrow{r_{min}} \leftarrow$ max_epu32($\vec{r_1}$, $\vec{r_2}$), min_epu32($\vec{r_1}$, $\vec{r_2}$)
  $\vec{r_1}, \vec{r_2} \leftarrow \overrightarrow{r_{max}}, \overrightarrow{r_{min}}$
  $\vec{r_3} \leftarrow$ VecScatterMin($\vec{r_1}$, $\vec{r_2}$, T, m)
  $done \leftarrow$ mask_cmpeq_epi32_mask(m, $\vec{r_1}$, $\vec{r_3}$) // mask
  $\vec{r_1} \leftarrow \vec{r_3}$
  $m \leftarrow \neg done \wedge m$ // mask

4.2 SIMD Rosenfeld pixel algorithm

Like its scalar counterpart, the SIMD Rosenfeld pixel algorithm (v1) is a two pass direct CCL algorithm. In order to simplify the algorithm as well as improving the memory footprint and the performance, we embed the equivalence table into the image. The label creation can now easily be done in parallel as the new label is equal to the linear address of the pixel plus 1 to differentiate the background: $i \times w + j + 1$, where (i, j) are the pixel coordinates and w the width of the image. This bijection also allows for a faster relabeling as it can be done during the transitive closure step.

Algorithm 8: VecScatterMin($\overrightarrow{idx}$, $\overrightarrow{val}$, T, m)

Input: $\overrightarrow{idx}$, $\overrightarrow{val}$ two vectors of labels, T an equivalence table, m a mask
Result: $\vec{r}$, the old values of $T[\overrightarrow{idx}]$

$\overrightarrow{rotate} \leftarrow$ set_epi32(0, 7, 6, 5, 4, 3, 2, 1)
$\overrightarrow{idx_c} \leftarrow$ maskz_compress_epi32(m, $\overrightarrow{idx}$)
$\overrightarrow{val_c} \leftarrow$ maskz_compress_epi32(m, $\overrightarrow{val}$)
$n \leftarrow$ popcnt_u32(m)
$\overrightarrow{rotate} \leftarrow$ maskz_move_epi32(0xFFFF >> (17 − n), $\overrightarrow{rotate}$)
$\overrightarrow{idx_r}, \overrightarrow{val_r} \leftarrow \overrightarrow{idx_c}, \overrightarrow{val_c}$
for $i \leftarrow 0$ *to* n **do**
  $\overrightarrow{idx_r} \leftarrow$ permute_epi32($\overrightarrow{rotate}$, $\overrightarrow{idx_r}$)
  $\overrightarrow{val_r} \leftarrow$ permute_epi32($\overrightarrow{rotate}$, $\overrightarrow{val_r}$)
  $same\_addr \leftarrow \overrightarrow{idx_c} = \overrightarrow{idx_r}$ // mask
  $\overrightarrow{val} \leftarrow$ mask_min_epu32($\overrightarrow{val}$, $same\_addr$, $\overrightarrow{val}$, $\overrightarrow{old}$)
$\overrightarrow{old} \leftarrow$ mask_i32gather_epi32($\overrightarrow{idx}$, m, $\overrightarrow{idx}$, T, sizeof(uint32))
$\overrightarrow{new} \leftarrow$ maskz_expand_epi32(m, $\overrightarrow{val_c}$)
$\overrightarrow{new} \leftarrow$ min_epu32($\overrightarrow{new}$, $\overrightarrow{old}$)
mask_i32scatter_epi32(T, m, $\overrightarrow{idx}$, $\overrightarrow{new}$, sizeof(uint32))
$\vec{r} \leftarrow$ mask_mov_epi32($\overrightarrow{new}$, cmpeq_epi32_mask($\overrightarrow{new}$, $\overrightarrow{val}$), $\overrightarrow{old}$)
return $\vec{r}$

Algorithm 9 describes the processing of a pixel vector during the first pass. The pixels are processed in a sequential natural reading order. Neighbor vectors $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$ and the current pixel $\vec{x}$ can be obtained by doing aligned loads or by register rotation. Border and corners cases can be handled by setting the out of image pixel vectors to zero. We start by constructing unaligned vectors $\vec{ab}$ and $\vec{bc}$ from $\vec{a}$, $\vec{b}$, $\vec{c}$ by doing some element shifting (lines 2 and 3). We also compute the bitmask m corresponding to $\vec{x}$: a bit is set to 1 for a foreground pixel and to 0 for a background pixel (line 4). The next step is to initialize the labels in $\vec{x}$ as shown in figure 8. Each pixel either points to itself or to its left neighbour (lines 6 to 9). We can now compute $\vec{dx}$ from $\vec{d}$ and $\vec{x}$ and propagate the neighbor labels into $\vec{x}$ (lines 10 and 11). The $vec\_maskz\_min_n^+$ function can be implemented using the property of unsigned integers overflow (−1 = MAX_UINT) and the *maskz_min_epu32* intrinsics. Finally, $\vec{x}$ can be stored to memory (line 13) and we can call the *VecUnion* function described in subsection 4.1 (lines 15 to 18). As previously said, only the stairs and concavity patterns can lead to an union operation. The other configurations are handled with the label propagation step. In our implementation, the equivalence table pointer is moved by 1 pixel to account for the +1 in the labels and simplify memory accesses.

The second pass is the simultaneous transitive closure and relabeling. In this pass we don't need the neighbor information making the loading pattern straightforward. For each vector of pixel $\vec{e}$, we find the corresponding equivalence tree root and write it back to memory. Processing the pixels in the same order as in first pass allows the algorithm to capitalize on previous iterations to find the roots faster. We can compute the true number of label n, using the *popcnt_u32* (population count in a 32-bit integer) instruction and some vector comparisons. We use the property that by definition a root label points to itself. Algorithm 10 describes this process.

Figure 7 represents the execution of the SIMD Rosenfeld pixel algorithm on a 12×4 image and SIMD register of size of 4. The outlined area shows the steps of algorithm 9. The *VecUnion* operation is not detailed here but the pattern is similar to the one in figure 6. The modifications in the image from the scan (9 -> 4, 18 ->4) in figure 7 are due to the equivalence table being embed in the image.

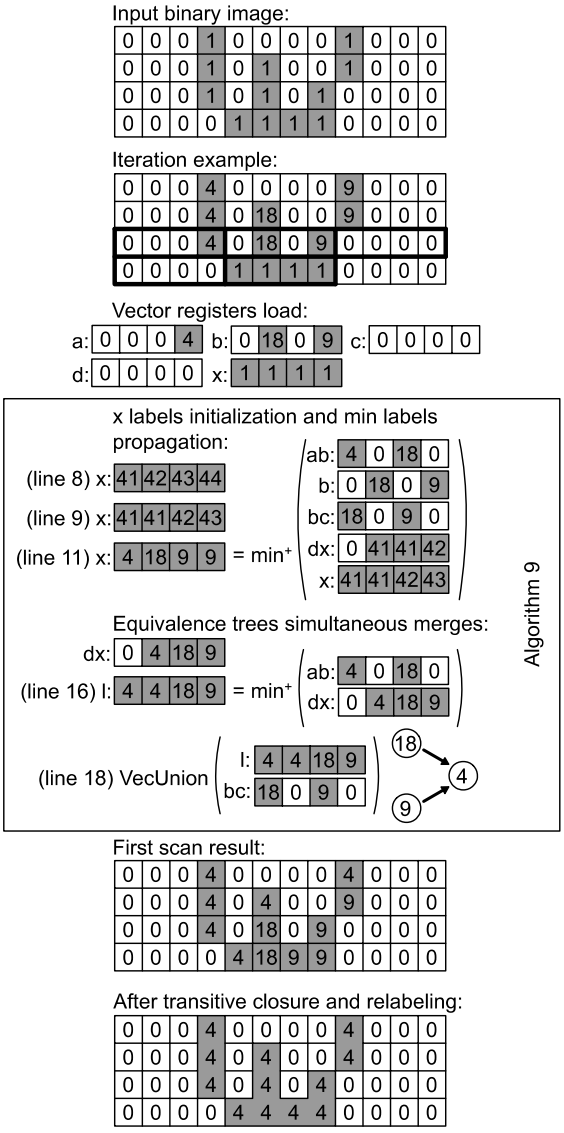


Figure 7: Example of an iteration of the SIMD Rosenfeld pixel algorithm.

4.3 SIMD Rosenfeld sub-segment algorithm

The SIMD rosenfeld pixel algorithm works well for low and medium image densities but for high image densities the performance collapses due to the pixel-recursive labels leading to more iterations in the *VecFind* while loop. To address this issue at the cost of a few more operations, we introduce the SIMD Rosenfeld sub-segment algorithm (v2). The only difference with the SIMD Rosenfeld pixel algorithm lies in the way we produce new labels (Fig. 8).

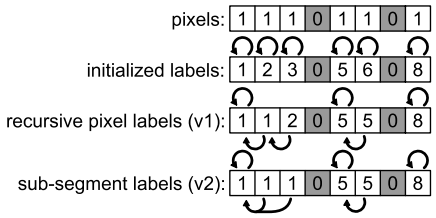


Figure 8: Creation of labels in SIMD Rosenfeld pixel algorithm (v1) and SIMD Rosenfeld sub-segment algorithm (v2).

Algorithm 9: SIMD Rosenfeld pixel (v1)

Input: T, the image / equivalence table, $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

1 // Shuffles :
2 $\vec{ab} \leftarrow$ vec_shift_right_epi32($\vec{a}$, $\vec{b}$)
3 $\vec{bc} \leftarrow$ vec_shift_left_epi32($\vec{b}$, $\vec{c}$)
4 $m \leftarrow \vec{x} \neq \vec{0}$ // mask
5 // x labels initialization and min labels propagation :
6 $\overrightarrow{inc} \leftarrow$ set_epi32(7, 6, 5, 4, 3, 2, 1, 0)
7 $\overrightarrow{base} \leftarrow$ set1_epi32($i \times w + j + 1$)
8 $\vec{x} \leftarrow$ maskz_add_epi32(m, $\overrightarrow{base}$, $\overrightarrow{inc}$)
9 $\vec{x} \leftarrow$ mask_sub_epi32($\vec{x}$, $m \wedge (m << 1)$, $\vec{x}$, $\vec{1}$)
10 $\vec{dx} \leftarrow$ vec_shift_right_epi32($\vec{d}$, $\vec{x}$)
11 $\vec{x} \leftarrow$ vec_maskz_min$^{+}_{5}$(m, $\vec{ab}$, $\vec{b}$, $\vec{bc}$, $\vec{dx}$, $\vec{x}$)
12 // Store x :
13 mask_store_epi32(&$T[i][j]$, m, $\vec{x}$)
14 // Equivalence trees simultaneous merges :
15 $\vec{dx} \leftarrow$ vec_shift_right_epi32($\vec{d}$, $\vec{x}$)
16 $\vec{l} \leftarrow$ vec_maskz_min^{+}(m, $\vec{ab}$, $\vec{dx}$)
17 $merge \leftarrow (\vec{bc} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge (\vec{l} \neq \vec{0}) \wedge m$ // mask
18 VecUnion($\vec{l}$, $\vec{bc}$, &$T[0][-1]$, $merge$)

Algorithm 10: SIMD Transitive closure (solve equivalences)

Input: T, the image / equivalence table, w, the width of the image
Result: n, the true number of labels in the image (optional)

1 $n \leftarrow 0$
2 **for** ***each*** $\vec{e} \in T$ **do**
3 $m \leftarrow \vec{e} = \vec{0}$ // mask
4 $\vec{e} \leftarrow$ VecFind($\vec{e}$, T, m)
5 mask_store_epi32(&$T[i][j]$, m, $\vec{e}$)
6 // (i, j) are the coordinates of $\vec{e}$
7 $\overrightarrow{inc} \leftarrow$ set_epi32(7, 6, 5, 4, 3, 2, 1, 0)
8 $\overrightarrow{base} \leftarrow$ set1_epi32($i \times w + j + 1$)
9 $\vec{l} \leftarrow$ maskz_add(m, $\overrightarrow{base}$, $\overrightarrow{inc}$)
10 $n \leftarrow n +$ popcnt_u32(mask_cmpeq_epi32_mask(m, $\vec{l}$, $\vec{e}$))

We define a segment as a sequence of same value pixels. A sub-segment is a segment bounded by the size of a vector. We use the conflict detection instructions from AVX512CD to compute a conflict-free subset ($c\vec{f}ss$) which allows us, with the count leading zero instruction (*lzcnt_epi32*) and some bit manipulation, to retrieve the index of the first element of a sub-segment. By making the pixel point directly to the sub-segment start, we can reduce the number of *VecFind* iterations to only 1 jump per sub-segment. Algorithm 11 describes theses changes and figure 9 shows the key states of the sub-segment start computation code. Lines 14 to 16 of algorithm 11

are optional for the correctness of the algorithm but improve the performance.

	0	1	2	3	4	5	6	7
pixels:	1	1	1	0	1	1	0	1
cfss:	00000000	00000001	00000011	00000000	00000111	00010111	00001000	00110111
bitmask:	00000000	00000001	00000011	00000111	00001111	00011111	00111111	01111111
andnot:	00000000	00000000	0000000	00000111	00001000	00001000	00110111	01001000
lzct:	8	8	8	5	4	4	2	1
x:	0	0	0	3	4	4	6	7

Figure 9: Use of conflict detection to find the sub-segment's start indices. In this example, elements have 8 bits and *lzct* count from the 8^{th} bit instead of the 32^{th}.

Algorithm 11: SIMD Rosenfeld sub-segment (v2)

Input: T, the image / equivalence table, $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

// Shuffles :
$\vec{ab} \leftarrow$ vec_shift_right_epi32($\vec{a}$, $\vec{b}$)
$\vec{bc} \leftarrow$ vec_shift_left_epi32($\vec{b}$, $\vec{c}$)
$m \leftarrow \vec{x} \neq \vec{0}$
// x labels initialization and min labels propagation :
$\overrightarrow{bitmask} \leftarrow$ set_epi32(0x7F, 0x3F, 0x1F, 0xF, 7, 3, 1, 0)
$\overrightarrow{cfss} \leftarrow$ maskz_conflict(m, x)
$\overrightarrow{lzct} \leftarrow$ lzcnt_epi32(andnot_epi32($\overrightarrow{cfss}$, $\overrightarrow{bitmask}$))
$\overrightarrow{base} \leftarrow$ set1_epi32($i \times w + j + 1 + 32$)
$\vec{x} \leftarrow$ maskz_sub_epi32(m, $\overrightarrow{base}$, $\overrightarrow{lzct}$)
$\vec{dx} \leftarrow$ vec_shift_right_epi32($\vec{d}$, $\vec{x}$)
$\vec{x} \leftarrow$ vec_maskz_min$^{+}_{5}$(m, $\vec{ab}$, $\vec{b}$, $\vec{bc}$, $\vec{dx}$, $\vec{x}$)
// Optional propagation:
$\overrightarrow{perm} \leftarrow$ maskz_sub_epi32(m, 32, $\overrightarrow{lzct}$)
$\vec{x_p} \leftarrow$ permute_epi32($\overrightarrow{perm}$, $\vec{x}$)
$\vec{x} \leftarrow$ vec_maskz_min^{+}(m, $\vec{x}$, $\vec{x_p}$)
// Store:
mask_store_epi32(&$T[i][j]$, m, $\vec{x}$)
// Equivalence trees simultaneous merges :
$\vec{dx} \leftarrow$ vec_shift_right_epi32($\vec{d}$, $\vec{x}$)
$\vec{l} \leftarrow$ vec_maskz_min^{+}(m, $\vec{ab}$, $\vec{dx}$)
$merge \leftarrow (\vec{bc} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge (\vec{l} \neq \vec{0}) \wedge m$
VecUnion($\vec{l}$, $\vec{bc}$, &$T[0][-1]$, $merge$)

4.4 Parallel SIMD algorithms

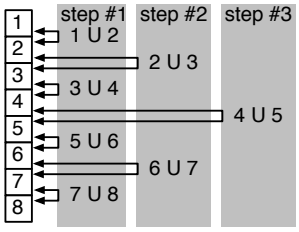


Figure 10: Pyramidal border merging of disjoint sets

We use OpenMP for the parallel implementation and make the assumption that the memory model is NUMA with shared memory between processors. SIMD algorithms have an increased pressure on memory bandwidth which tends to reduce the multicore-parallelism efficiency if the application is not compute bound.

The approach we follow in our parallel implementations of the SIMD Rosenfeld pixel and sub-segment algorithms is based on a divide-and-conquer method described in [4]. The image is split into p sub-images, with p the number of cores. This parallel algorithm minimizes the number of merges required by taking a pyramidal approach but diminishes the number of active cores at a given time. It needs $\log_2(p)$ steps to complete the merge. Each step is fully parallel and does not require *atomic* instructions to update the equivalence table as this scheme merges borders of disjoint sets (Fig. 10).

First, each processor core takes a sub-image horizontal strip and applies the first pass of either algorithm. Except for the modified loop indexes, there is no difference between the sequential and parallel code. Then, the next step consists in applying a pyramidal merge. As we divide the image by the number of cores available p, the total number of merges required is equal to $p-1$. For each merge step, we divide the number of active cores doing a two way merge until we have only one core left. On the border line between two sub-images, we apply a merging pass on each column. For each SIMD vector of pixels, we have at most two calls to *VecUnion*. Algorithm 12 describes the body of the border merge loop. Finally, we apply the second pass which does not require any code modification compared to the sequential version.

Algorithm 12: SIMD Border merging

Input: T, the image / equivalence table, $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

// Shuffles :
$\vec{ab} \leftarrow$ vec_shift_right_epi32($\vec{a}$, $\vec{b}$)
$\vec{bc} \leftarrow$ vec_shift_left_epi32($\vec{b}$, $\vec{c}$)
$\vec{dx} \leftarrow$ vec_shift_right_epi32($\vec{d}$, $\vec{x}$)
$m \leftarrow \vec{x} \neq \vec{0}$
// Equivalence trees simultaneous merges :
$\vec{l} \leftarrow$ vec_maskz_min^{+}(m, $\vec{ab}$, $\vec{b}$)
$merge \leftarrow (\vec{l} \neq \vec{0}) \wedge (\vec{dx} = \vec{0}) \wedge m$ // mask
VecUnion($\vec{x}$, $\vec{l}$, &$T[0][-1]$, $merge$)
$\vec{l} \leftarrow$ vec_maskz_min^{+}(m, $\vec{b}$, $\vec{bc}$)
$merge \leftarrow (\vec{l} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge m$ // mask
VecUnion($\vec{x}$, $\vec{l}$, &$T[0][-1]$, $merge$)

5 BENCHMARK

5.1 Benchmark protocol

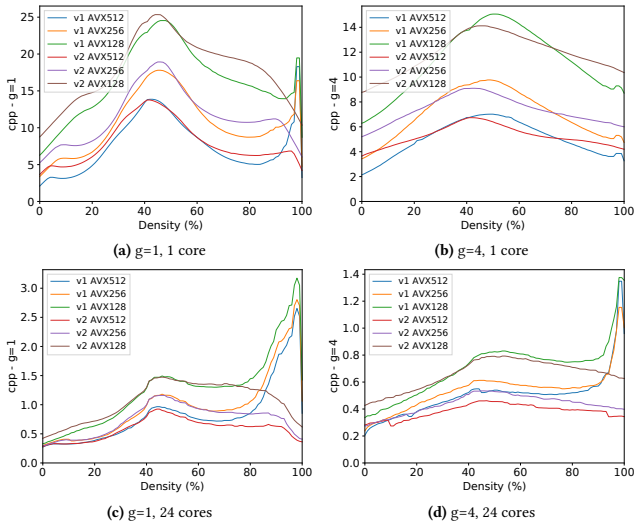
In our benchmark protocol, we aim to study the behavior of CCL algorithms in the widest range of configurations available, and to see how they compare to each other. For reproducible results, MT19937 [14] was used to generate images of varying density ($d \in [0\% - 100\%]$) and granularity ($g \in \{1 - 4\}$). The image density is the ratio of black and white pixels in the image: a black image (with zero pixel set) has a density of 0% and a white image (with all pixel set) has a density of 100%. An image of granularity g

is composed of $g \times g$ macro-pixels set to 1 or 0. Granularity increases the local homogeneity of an image. Natural and structured images lead to processing times roughly equal to those of random images with $g = 4$, while $g = 1$ provides difficult and unstructured configurations to stress the algorithms and especially those manipulating segments or run-length's. This benchmark methodology for CCL algorithms was previously used in [12], [3], [4], [8].

We tested the performance of the available SIMD vector length (128, 256, 512) in single and multicore on a dual socket Intel Xeon(R) Gold 6126 running at 2.6 Ghz (turbo-boost off). Before any testing, it was unclear whether the new 512 vector length would have a positive impact on the performance due to the additional stress on the memory bandwidth and the frequency throttle. This is especially true while fully exploiting SIMD and multicore due to bandwidth saturation. Our results are summarized in table 1, 2 and figures 11, 12 and are discussed in the following section. The performance of these new algorithms is compared to the classic pixel-based algorithms with DT (Rosenfeld and Rosenfeld+DT) and to the fastest run-length based segment labeling algorithm (LSL_{STD} and LSL_{RLE}) [4].

5.2 Results analysis

Figure 11 shows the execution time (in cycles per pixels) on images of varying densities. We observe that for both versions, there is a bump around 45% which corresponds to the percolation threshold in 8-connectivity. We can also see that the main difference in performance between the pixel and sub-segment algorithms happens for higher densities. The pixel version being slower because of the recursive pixel labels leading to a longer while loop in *VecFind* (as seen in Sec. 4.3). This performance difference grows with the number of cores due to the added cost of find operations in the border merging step.



(a) g=1, 1 core **(b)** g=4, 1 core

(c) g=1, 24 cores **(d)** g=4, 24 cores

Figure 11: Cycles per pixels for SIMD Rosenfeld pixel (v1) and SIMD Rosenfeld sub-segment (v2), applied to 2048×2048 images

SIMD vs scalar: In the single-threaded case, SIMD versions are two times faster than the Rosenfeld scalar versions. They are also faster than the LSL versions. In the multi-threaded case, this ratio grows to ×3. Compared to LSL, the SIMD versions are faster only for g=1, which is the worst case for full-segment labeling like LSL (the strategy to save memory accesses, especially with the RLE version becomes more profitable than in the mono-threaded case where the pressure on the memory is lower).

threading	1-core mono thread			24-core multi thread		
granularity	g=1	g=2	g=4	g=1	g=2	g=4
LSL_{STD}	10.47	6.98	5.91	0.71	0.36	0.28
LSL_{RLE}	16.96	9.31	6.05	0.95	0.45	0.24
Rosenfeld	30.61	19.01	12.49	2.56	1.69	1.27
Rosenfeld+DT	20.01	11.81	8.30	1.95	1.51	1.09
SIMD $v1_{512}$	**7.61**	**6.07**	**5.06**	0.84	0.52	0.50
SIMD $v1_{256}$	10.65	8.64	7.08	0.99	0.57	0.55
SIMD $v1_{128}$	16.40	13.23	11.33	1.26	0.76	0.71
SIMD $v2_{512}$	7.97	6.54	5.26	**0.58**	**0.41**	**0.38**
SIMD $v2_{256}$	11.54	9.22	7.35	0.74	0.53	0.44
SIMD $v2_{128}$	17.89	14.15	11.89	1.07	0.71	0.66

Table 1: Average cycles per pixels for 2048×2048 images - best SIMD in bold (lower values is better)

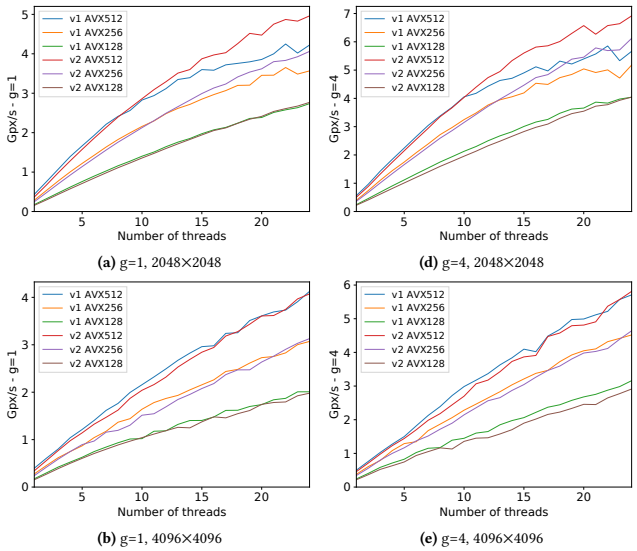
threading	1-core mono thread			24-core multi thread		
granularity	g=1	g=2	g=4	g=1	g=2	g=4
LSL_{STD}	0.27	0.38	0.44	4.12	7.53	9.45
LSL_{RLE}	0.21	0.32	0.46	3.56	6.91	11.80
Rosenfeld	0.13	0.17	0.24	1.15	1.70	2.17
Rosenfeld+DT	0.17	0.25	0.33	1.47	2.02	2.46
SIMD $v1_{512}$	**0.44**	**0.49**	**0.56**	4.22	5.06	5.66
SIMD $v1_{256}$	0.29	0.34	0.40	3.57	4.21	5.18
SIMD $v1_{128}$	0.18	0.21	0.24	2.74	3.37	4.04
SIMD $v2_{512}$	0.37	0.42	0.51	**4.96**	**5.84**	**6.92**
SIMD $v2_{256}$	0.25	0.30	0.36	4.06	4.98	6.12
SIMD $v2_{128}$	0.16	0.19	0.22	2.77	3.48	4.05

Table 2: Average throughput (Gpx/s) for 2048×2048 images - best SIMD in bold (higher is better).

SIMD Scalability (128 / 256 / 512) In the single-threaded case, doubling the SIMD size provides a speedup around 1.5. In multi-threaded case, this ratio still exists but only for 128 / 256 registers. For 256 / 512 registers, the ratio drops to 1.2, due to bandwidth saturation.

Thread scalability: Depending on the SIMD size, the scalability of the algorithms varies in the interval ×10−×13 for 512-bit registers and up to ×15 − ×18 for 128-bit registers. That is an efficiency between 40% (for 512-bit registers) up to 75% (for 128-bit registers). Considering all the instructions for the control-flow and the label propagation within a register, for such an irregular algorithm, we consider the results acceptable.

Performance with image size: The best performance is achieved when the image fits in the cache for all algorithms. (see Tab. 3). For a granularity equal to 4, LSL_{RLE} is still the fastest algorithm. On



(a) g=1, 2048×2048 **(d)** g=4, 2048×2048

(b) g=1, 4096×4096 **(e)** g=4, 4096×4096

Figure 12: Average throughput (Gpx/s) for SIMD Rosenfeld pixel (v1) and SIMD Rosenfeld sub-segment (v2)

lower granularities, the new algorithms outperform Rosenfeld+DT by a factor ×2.3 up to ×2.8 for g=4, and from ×2.6 up to ×3.4 for g=1. We also observe that, for larger images, the pixel-based algorithm SIMD v1 becomes faster than the sub-segment-based algorithm SIMD v2. These two new algorithms are specially well-suited to very complex / un-structured / quasi-random images.

image size	2k images		4k images		8k images	
granularity	g=1	g=4	g=1	g=4	g=1	g=4
LSL_{RLE}	3.56	**11.80**	3.45	**9.16**	3.25	**7.55**
Rosenfeld+DT	1.47	2.46	1.41	2.22	1.37	1.94
$SIMD\ v1_{512}$	4.22	5.66	**4.13**	5.71	**3.83**	4.89
$SIMD\ v2_{512}$	**4.96**	6.92	4.07	5.81	3.59	4.45

Table 3: Average throughput (Gpx/s) for 2k, 4k, 8k images on 24 cores (best performance in bold, for each column).

6 CONCLUSION

This paper presented two new CCL direct algorithms for SIMD multi-core architectures. The work is motivated by a large number of applications in computer vision and could also be of interest for the graph community with a new Union-Find SIMD algorithm. The use of SIMD instructions in CCL and more generally in computer vision is scarce due to the complexity of the code vectorization process. We have developed a performance efficient and portable SIMD algorithms for the Rosenfeld algorithm that outperform scalar pixel-based algorithms by a factor of ×2.3 up to ×3.4.

We found out that the new optimized SIMD algorithms are performing better than the State-of-the-Art algorithms on small images of fine granularity and on single core CPU. While it does not scale as well as run-length based algorithms, a lot of applications which do not require big images but instead search for high throughput could benefit from it. In future work, we plan to test our algorithms on new ARM/Fujisu SVE architectures.

REFERENCES

[1] D. A. Bader and J. Jaja. Parallel algorithms for image histogramming and connected components with an experimental study. *Parallel and Distributed Computing*, 35,2:173–190, 1995.

[2] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana. Toward reliable experiments on the performance of connected components labeling algorithms. *Journal of Real-Time Image Processing (JRTIP)*, pages 1–16, 2018.

[3] L. Cabaret, L. Lacassagne, and D. Etiemble. Distanceless label propagation: an efficient direct connected component labeling algorithm for GPUs. In *IEEE International Conference on Image Processing Theory, Tools and Applications (IPTA)*, pages 1–8, 2017.

[4] 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*, 15(1):173–196, 2018.

[5] M. Ceska. Computing strongly connected components in parallel on cuda. In Nvidia, editor, *GPU Technology Conference*, 2010.

[6] 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.

[7] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao. The connected-component labeling problem: a review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[8] A. Hennequin, Q. L. Meunier, L. Lacassagne, and L. Cabaret. A new direct connected component labeling and analysis algorithm for GPUs. In *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–6, 2018.

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

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

[11] Y. Komura. Gpu-based cluster-labeling algorithm without the use of conventional iteration: application to swendsen-wang multi-cluster spin flip algorithm. *Computer Physics Communications*, pages 54–58, 2015.

[12] L. Lacassagne, L. Cabaret, F. Hebache, and A. Petreto. A new SIMD iterative connected component labeling algorithm. In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 1–8, 2016.

[13] A. Lindner, A. Bieniek, and H. Burkhardt. PISA-Parallel image segmentation algorithms. pages 1–10. Springer, 1999.

[14] 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.

[15] M. Niknam, P. Thulasiraman, and S. Camorlinga. A parallel algorithm for connected component labeling of gray-scale images on homogeneous multicore architectures. *Journal of Physics - High Performance Computing Symposium (HPCS)*, 2010.

[16] D. P. Playne and K. Hawick. A new algorithm for parallel connected-component labelling on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

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

[19] F. Wende and T. Steinke. Swendsen-wang multi-cluster algorithm for the 2d/3d Ising Model on Xeon Phi and GPU. In ACM, editor, *International Conference on High Performance Computing (SuperComputing)*, pages 1–12, 2013.

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