How to speed Connected Component Labeling up with SIMD RLE algorithms

Florian Lemaitre, Arthur Hennequin, Lionel Lacassagne
firstname.name@lip6.fr
LIP6 – Sorbonne University, CNRS

ABSTRACT

The research in Connected Component Labeling, although old, is still very active and several efficient algorithms for CPUs and GPUs have emerged during the last years and are always improving the performance. This article introduces a new SIMD run-based algorithm for CCL. We show how RLE compression can be SIMDized and used to accelerate scalar run-based CCL algorithms. A benchmark done on Intel, AMD and ARM processors shows that this new algorithm outperforms the State-of-the-Art by an average factor of ×1.7 on AVX2 machines and ×1.9 on Intel Xeon Skylake with AVX512.

INTRODUCTION & STATE OF THE ART

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.

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][2]. After an era on single-core processors, where many sequential algorithms were developed [3] and codes were released [4], new parallel algorithms were developed on multicore processors [5][6].

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

But direct algorithms [7] are dominated by their control flow and can not be easily parallelized – unlike the iterative ones. That is the reason why the first algorithms designed for SIMD processors [8][9] and GPUs [10] were iterative. Now these algorithms are also direct [11] [12][13] and thanks to the parallelism of these architectures, they are very fast.

In the meantime, two new single-threaded algorithms for CPUs named DRAG [14] and Spaghetti [15] were published. Thanks to advanced algorithmic tricks based on block-processing and large decision trees to avoid comparisons and memory accesses, these algorithms outperformed all published pixel-based algorithms on CPUs.

All these recent algorithms for CPUs, SIMD CPUs and GPUs share a common denominator: they all process pixels one by one (DRAG and Spaghetti use 2×2-pixel blocks). Only LSL [16][17] uses RLE and run-based processing to reduce the amount of memory accesses. This approach was reused for HA, the first run-based algorithm for GPU [18].

In this article, we introduce a new run-based algorithm for SIMD processors which rely on classical SIMD instructions available on all SIMD family and without specific scatter-gather instructions.

Section 1 deals with binary RLE compression and present a way to improve it with SIMD and how to use it for labeling. Section 2 sums up pixel-based algorithms and introduces our new run-based SIMD algorithm. Section 3 presents an exhaustive benchmark to compare the latest pixel-based and run-based algorithms for CPUs where the results are detailed and analyzed.

1 RUN-LENGTH ENCODING

Run-Length Encoding (RLE)[19][20] is a basic lossless data compression initially used to compress binary signals and images. It is the basis of more advanced compression algorithms like LZ78[21] or bitmap index for databases[22]. It can also be used for non-compression algorithms like auto-correlation[23]. In the last years, people started to study again RLE-based algorithms[24][25][26] to take advantage of the recent SIMD instruction sets.

RLE consists in grouping consecutive elements with the same value in segments (or runs) and keeping only the information about the segments (position, size and value). As RLE was primarily designed for element streams, position is usually not kept and just inferred from the sum of the previous segment sizes.

In this paper, we consider binary image inputs. Therefore, we do not need to store explicitly values of the segments, and we store start and stop positions of the segments instead of length in order to speed partial decoding up. Note that the start position of a segment is the stop position of the previous segment, thus no space is wasted compared to actual length encoding.

We use semi-open intervals (left-closed, right-opened) in order to increase the symmetry between 0-segments and 1-segments and thus simplify both the implementations and the explanations. In addition, as the input image is the result of thresholding on a 8-bit gray image, the input is kept with 1 pixel per byte. Finally, as we encode line of images, we restrict our segment positions to 16-bit integers. All our implementations are adjustable to wider arrays and non-binary inputs.

Most RLE encoders follow the same pattern: detect edges (when the value changes), compute the position of the edges, and store them contiguously. The edge detection can be done by testing if the current element has the same value as its predecessor. This can be achieved using a $\oplus$ (`xor`) in binary.

1.1 Scalar RLE encoder

The code of the scalar implementation is given in Algorithm 1. It is a simple code that has few optimizations. Namely, we applied scalarization by keeping `X[i]` in register and rotating registers to

Algorithm 1: Scalar RLE (LSL step 1a)

```
1  er ← 0
2  // prolog
3  x_1 ← 0    // previous value of X (x_1 = X_i[j − 1])
4  for j = 0 to w − 1 do
5      x_0 ← X_i[j]    // current value
6      f ← x_0 ⊕ x_1    // edge detection
7      if f ≠ 0 then
8          RLC_i[er] ← j
9          er ← er + 1
10     ER_i[j] ← er
11     x_1 ← x_0    // register rotation
12 // epilog
13 x_0 ← 0    // next value of X (x_1 = X_i[w])
14 f ← x_0 ⊕ x_1
15 RLC_i[er] ← w
16 er ← er + f
17 ner ← er
18 return ner
```

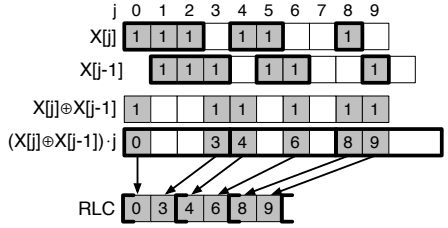

Figure 1: Example of a segment and its associated run-length encoding with a semi-open interval $[0, 3[4, 6[8, 9[$ with a SIMD4 `compress`

keep previous value in register. Note how the previous value is initialized to 0 and thus effectively skipping the first segment if its value is 0.

1.2 SIMD RLE encoder

An RLE encoder is not trivially "SIMDizable". The complexity of SIMD implementations is due to the irregularity of the problem: fixed-length input produces variable-length output. This is visible in the scalar code where the store and increment is within a branch.

We tested multiple ways to write SIMD RLE encoders: with a finite state machine, with `scatter` and with `compress`. For brevity and because its the fastest in most cases, we will focus on `compress` only.

The first step is to detect the edges (the positions where the value changes). This is done by comparing the input vector with itself shifted by one element to the "future" (we compare each element with its predecessor). The resulting vector contains ones where the input value at this position is different than at the previous position, indicating where the edges are.

Then, the vector of positions ($[0, 1, 2, 3\ldots]$) is `compressed` according to the edge vector. Meaning: the positions that are not edges (whose corresponding element in the edge vector is 0) are removed, and the remaining positions are packed contiguously on left of the resulting vector. This resulting vector is directly the RLE of the input. A diagram showing this algorithm is provided in Figure 1.

Listing 1: `compress`-based RLE encoder targeting SSE

```
int16_t* detect_segment(
                    int w, uint8_t* X, int16_t* RLC) {
  __m128i last = _mm_setzero_si128();
  __m128i incr8 = _mm_set1_epi16(8);
  __m128i J = _mm_set_epi16(7, 6, ..., 1, 0);
  for (int i = 0; i < w; i += 8) {
    __m128i in = _mm_loadl_epi64((__m128i*)(X+i));
    in = _mm_cvtepi8_epi16(in);
    // edge detection
    __m128i prev = _mm_alignr_epi8(in, last, 14);
    __m128i f = _mm_cmpeq_epi8(in, prev);
    f = _mm_xor_si128(f, _mm_set_epi8(-1));
    last = in;
    // compress and store indices
    RLC = compress_storeu_m16_epi16(RLC, J, f);
    J = _mm_add_epi16(J, incr8);
  }
  *RLC++ = w;
  return RLC;
}
```

There are several ways to implement this algorithm. The simplest one is given in Listing 1. This implementation loads 8-bit elements and first converts them into 16-bit elements. Then, the whole processing is done on 16-bit elements.

One can do the edge detection and the conversion into a mask on 8-bit elements, and then split the mask in two and do two separate `compress` on 16-bit elements like above.

We have actually tested the two version above, and a third one where the `compress` is done directly on 8-bit elements. But only the version using 8-bit masks and 16-bit positions is considered in this paper for the sake of brevity and simplicity and because it appears to be the fastest one.

1.3 Emulating `compress`

No current hardware has native support for 8-bit and 16-bit element `compress`, and few have native support for wider elements (AVX512 and SVE). So we need to emulate those narrow element `compress` on all architecture. We tested many ways to implement `compress` like scalar emulation, Lookup Tables (LUT)[27] or hierarchical shifts[28, Chapter 7-4].

We report here the best versions for each considered platforms. For all the versions considered here, `popcount` is used to know the number of active elements, and the store is full-width and unaligned. This full-width store implies that some junks elements will be stored, but they will either be overwritten by the next store or they will never be read (past the end).

1.3.1 AVX512. We can `compress` 32-bit elements with the AVX512 instruction `_mm512_maskz_compress_epi32`. This can be used to `compress` narrower elements. The way to do it is to widen elements from 16 bits to 32 bits, use the native 32-bit `compress`, and then narrow 32-bit elements back to 16 bits. Compressing 8-bit elements can be done in the same way. Note that this implementation requires wider vectors to apply the `compress`, and thus can only process up to 16 elements (AVX512 registers can hold 16 32-bit elements). This technique would be directly applicable to SVE as well.

Listing 2: LUT-based compress for SSE

```
int16_t* _mm_compress_storeu_m16_epi16(
                    int16_t* p, __m128i v, __m128i m) {
  m = _mm_packs_epi16(m, _mm_setzero_si128());
  int mi = _mm_movemask_epi8(m);
  __m128i perm = _mm_load_si128((__m128i*)LUT16x8[mi]);
  v = _mm_shuffle_epi8(v, perm);
  _mm_storeu_si128((__m128i*)p, v);
  return p + __builtin_popcount(mi);
}
```

1.3.2 SSSE3 → AVX2. Without native support for `compress`, the best way is to use a LUT to compute the permutation that has the same effect as the `compress`. The LUT-based implementation converts the edge mask into a scalar integer, then uses this integer to find the right permutation rule from within the LUT. This permutation is then applied using `_mm_shuffle_epi8`. The code is shown in Listing 2. No AVX or AVX2 instruction seems to profit to such a code.

1.3.3 ARMv8. On ARM, the best way is also to use a LUT. The instruction `vqtbl1q_u8` is used to apply a run-time permutation.

However the implementation is more complex due to the absence of an instruction converting a mask vector into an integer (`_mm_movemask_epi8` on SSE). This conversion needs to be emulated by setting elements to $1, 2, 4, 8\ldots$, apply the mask and either reduce add (`vaddv_u16`) or reduce `or` (no specific instruction). Extra care is required for `uint8x16_t` as there are more elements in the vector than bits per elements.

1.4 SIMD RLE decoder

In its simplest form, RLE decoding just consists of iterating over the segments and `memset`ing the segment value from the start position to the stop position of the segment. If the segment value is encoded in a type larger than 8 bits, this cannot be done with an actual call to `memset`, but the idea remains the same.

One way to accelerate the decoding is to approximate `memset` by rounding the segment size up to a multiple of the SIMD cardinal. In that way, the expansion of a single segment involves less tests and branches. This code remains correct if the segments are iterated in order as the eventual extra elements written by the previous SIMD store are overridden by the current SIMD store. A diagram of this algorithm is provided in Figure 2.

The efficiency of this optimization highly depends on segment size and cannot be faster than the scalar `memset`-like code for segments of size 1.

2 CONNECTED COMPONENTS LABELING

2.1 Pixel-based Algorithms: Rosenfeld

Usually, CCL algorithms are split into three steps and perform two image scans like the pioneer algorithm by Rosenfeld [7]. Let us define some notations (Figure 3). Let p_x, e_x be the current pixel and its label. Let p_a, p_b, p_c, p_d be the neighbor pixels, and a, b, c, d their associated labels. T is the equivalence table, e a label and r its root. The first scan (or first labeling) assigns a temporary/provisional label to each CC and some equivalences between labels are

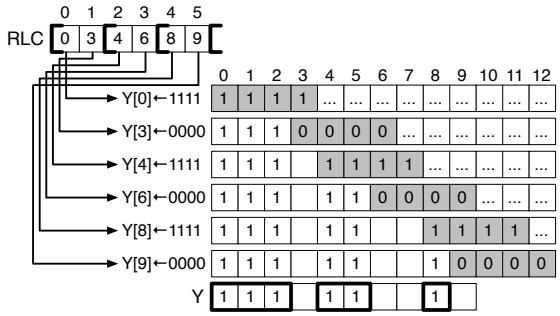

Figure 2: RLE decoder with 4-block SIMD

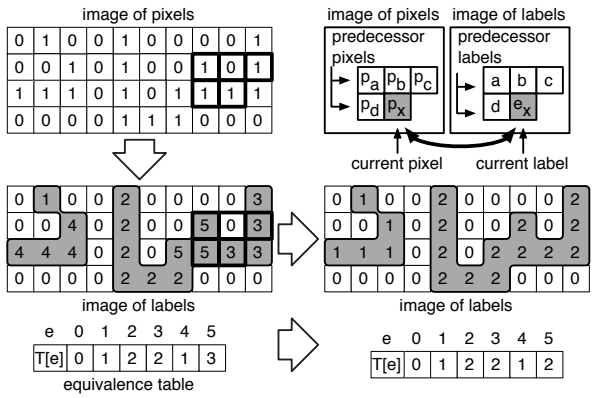

Figure 3: Example of 8-C connected component labeling with Rosenfeld algorithm: binary image (top), image of temporary labels (bottom left), image of final labels (bottom righ) after the transitive closure of the equivalence table

built if needed (Figure 3 and Algorithm 2). The second step solves the equivalence table (Algorithm 3) by computing the transitive closure (TC) of the graph associated to the label equivalence. The third step performs a second scan (or second labeling) that replaces temporary labels of each CC with their root (Algorithm 4).

In Figure 3, we can see that the right CC requires three labels (2, 3 and 5). At the mask position the equivalence between 3 and 5 is detected and stored into the equivalence table T. At the end of the first scan, the equivalence table is solved and applied to the image (like a Look-Up-Table).

2.2 Run-based algorithm: classic LSL

LSL is also composed of three steps. The first step is a two-fold step: there is first a segment detection (Algorithm 1) and then a segment Unification (Algorithm 6). The specificity of LSL is to be run-based and to use a line-relative labeling (Figure 4). From two consecutive lines X_{i-1}, X_i, two relative labelings (ER_i, ER_{i-1}) are produced where runs (or segments) have odd numbers. In the same way, the associated "*run-length coding*" are produced (tables RLC_{i-1}, RLC_i). The table ERA_i holds the translation between *Relative* and *Absolute* labels: $ea = ERA_i[er/2]$. To find out how which labels of the previous line are connected to the current segment (Unification), one has to read the value of relative labels from table ER_{i-1} at the positions given by RLC_i, and translates them into absolute labels

Algorithm 2: Rosenfeld – first labeling (step 1)

Input: a, b, c, d, four labels, p_x, the current pixel in (i, j)
if $p_x \neq 0$ **then**
  $a \leftarrow E_{i-1}[j-1]$
  $b \leftarrow E_{i-1}[j]$
  $c \leftarrow E_{i-1}[j+1]$
  $d \leftarrow E_i[j-1]$
  if $(a = b = c = d = 0)$ **then**
    $ne \leftarrow ne + 1$
    $e_x \leftarrow ne$
  else
    $r_a \leftarrow Find(a)$
    $r_b \leftarrow Find(b)$
    $r_c \leftarrow Find(c)$
    $r_d \leftarrow Find(d)$
    $e_x \leftarrow min^+(r_a, r_b, r_c, r_d)$ // ignore values $\leqslant 0$
    if $(r_a \neq 0$ ***and*** $r_a \neq e_x)$ **then** $T[r_a] \leftarrow e_x$
    if $(r_b \neq 0$ ***and*** $r_b \neq e_x)$ **then** $T[r_b] \leftarrow e_x$
    if $(r_c \neq 0$ ***and*** $r_c \neq e_x)$ **then** $T[r_c] \leftarrow e_x$
    if $(r_d \neq 0$ ***and*** $r_d \neq e_x)$ **then** $T[r_d] \leftarrow e_x$
else
  $e_x \leftarrow 0$
$E_i[j] \leftarrow e_x$ // temporary root

Algorithm 3: sequential solve of equivalences (step 2)

for $e = 1$ **to** ne **do**
  $T[e] \leftarrow T[T[e]]$

Algorithm 4: relabeling (step 3)

for $i = 0$ **to** $h - 1$ **do**
  for $j = 0$ **to** $w - 1$ **do**
    $E_i[j] \leftarrow T[E_i[j]]$

Algorithm 5: Find(e)

Input: e a label, T the equivalence table
Result: r, the root of e
$r \leftarrow e$
while $T[r] \neq r$ **do**
  $r \leftarrow T[r]$
return r

to update the equivalence table T. An alternate algorithm to Unification is the finite state machine of Figure 5. It is described in the next section.

The second step is the same as the pixel-based algorithm (Algorithm 3). The third step is also RLE-accelerated: it is a simple RLE decoder where each label is replaced by its root (Algorithm 7). Compared to the implementation in [17], table *RLC* encodes semi-open intervals and the associative table *ERA* is packed and does not contain zero (so accesses look like $ERA[er/2]$ instead of $ERA[er]$) and *LEA* is no more used.

2.3 New SIMDized Faster LSL

For our new implementation of a Faster LSL, the SIMDized segment detection (RLE encoder: step 1a) and Segment expansion (RLE decoder: step 3) are used. We have also developed an alternative

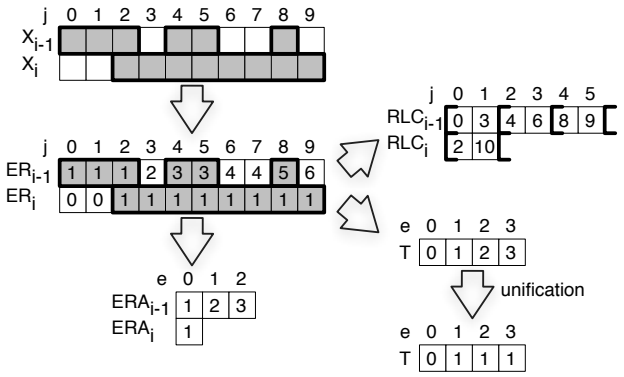

Figure 4: LSL tables

Algorithm 6: LSL Unification U0 (step 1b)

for $er = 1$ **to** $ner - 1$ **step** 2 **do**
  // semi open interval segment extraction $[j_0, j_1[$
  $j_0 \leftarrow RLC_i[er-1]$
  $j_1 \leftarrow RLC_i[er]$
  // check extension in case of 8-connect algorithm
  if $j_0 > 0$ **then** $j_0 \leftarrow j_0 - 1$
  if $j_1 < w$ **then** $j_1 \leftarrow j_1 + 1$
  $e_{r0} \leftarrow ER_{i-1}[j_0]$
  $e_{r1} \leftarrow ER_{i-1}[j_1 - 1]$ // right compensation
  // check label parity: segments are odd
  if e_{r0} *is even* **then** $e_{r0} \leftarrow e_{r0} + 1$
  if e_{r1} *is even* **then** $e_{r1} \leftarrow e_{r1} - 1$
  if $e_{r1} \geq e_{r0}$ **then**
    $e_a \leftarrow ERA_{i-1}[e_{r0}/2]$
    $a \leftarrow Find(e_a)$
    for $e_{rk} = e_{r0} + 2$ **to** e_{r1} **step** 2 **do**
      $ea_k \leftarrow ERA_{i-1}[er_k/2]$
      $a_k \leftarrow Find(ea_k)$
      // Union: min extraction and propagation
      if $a < a_k$ **then**
        $T[a_k] \leftarrow a$
      if $a > a_k$ **then**
        $T[a] \leftarrow a_k$
        $a \leftarrow a_k$
    $ERA_i[er/2] \leftarrow a$ // the global min
  else
    // new label
    $ne \leftarrow ne + 1$
    $ERA_i[er/2] \leftarrow ne$

Algorithm 7: LSL relabeling (step 3)

for $i = 0$ **to** $h - 1$ **do**
  $j_1 \leftarrow 0$
  for $er = 1$ **to** $ner - 1$ **step** 2 **do** // number of segment on line i
    $j_0 \leftarrow RLC_i[er-1]$
    $E_i[j_1 : j_0[\leftarrow 0$ // The first BG segment might be empty
    $ea \leftarrow ERA_i[er/2]$
    $r \leftarrow T[ea]$
    $j_1 \leftarrow RLC_i[er]$
    $E_i[j_0 : j_1[\leftarrow r$ // FG segment
  $E_i[j_1 : w[\leftarrow 0$ // The last BG segment might be empty

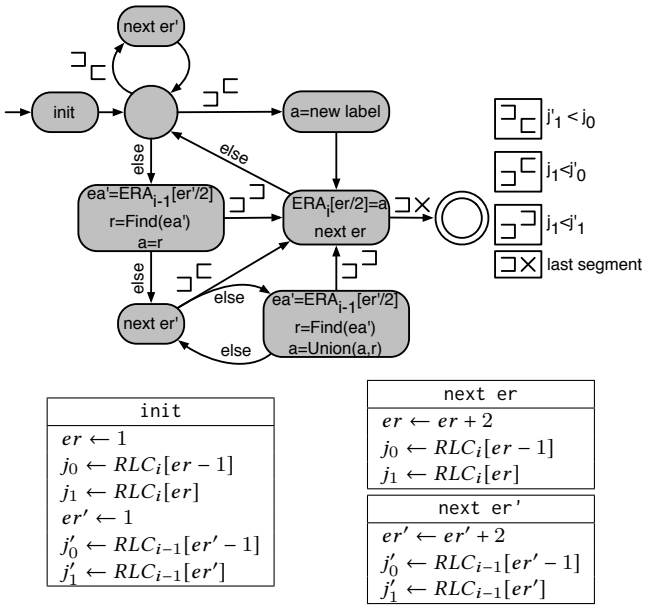

Figure 5: LSL Unification U1 with finite state machine (step 1b)

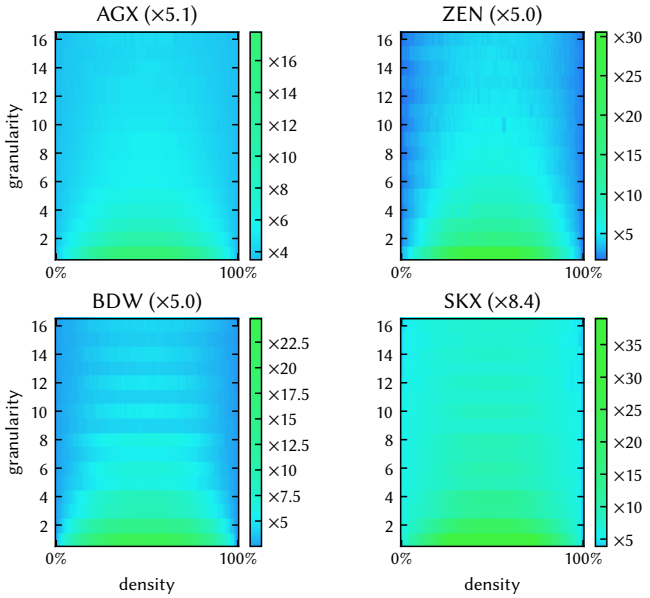

Figure 6: Speedup of the SIMD RLE encoder compared to the scalar RLE encoder on all benchmarked architectures for g ∈ [1, 16]

Unification step. It is a Finite state machine (Figure 5) that is close to the merge-sort algorithm: it iterates over segments of both current and previous lines at the same time.

This FSM uses `gotos` for its transitions for two reasons: first it avoids to compute more complex tests associated to a centralized dispatch (with a `switch-cases` construction) and second, distributed dispatches are more friendly for the branch predictor of the CPU that can learn the most probable transition for each state.

In order to skip a test within `next er'`, we insert a virtual segment at the end of the previous line that is located after the end of the line. Therefore, there is always a " next er' " that is after any segment of the current line and we do not need to test if there is remaining segments on the previous line. Indeed, once we reached this virtual segment, the state machine will never get into a `next er'` state.

3 BENCHMARKS

3.1 Protocol

In order to compare our new LSL implementation with State-of-the-Art algorithms and former LSL implementations, the benchmark uses the protocol shared by all recent publications: binary images with various granularity. First because it simplifies the analysis of results: OpenMP (or pthread) combined with SIMD put a high level of stress on memory resulting in a loss of parallelism for compute-bound algorithms like CCL. Second, because DRAG [14] and Spaghetti [15], the current State-of-the-Art algorithms, are only available in a mono-threaded implementation within the open-source YACCLAB library [29].

For reproducible results, MT19937 pseudo-random generator was used to generate 2048×2048 images of varying density $d \in [0\%; 100\%]$ and granularity $g \in [1; 16]$. The image density is the ratio of black and white pixels in the image. An image of granularity g is composed of $g \times g$ macro-pixels set to 1 or 0. Natural and structured images lead to processing times roughly equal to those of random images with $g = 4$, while $g = 1$ generates unstructured images to stress the algorithms and especially those manipulating run length structures.

All the versions (including YACCLAB ones) have been compiled with GCC 8 at `-O3`. The time is measured with `rdtsc` on x86 platforms and with `clock_gettime` on ARM platforms. Each image is processed multiple times, and the minimum time per image is kept. In order to have comparable measurements on different machines, the timings are given in cycles per pixel (cpp = $\frac{time \cdot freq}{width \cdot height}$).

A first benchmark was done to evaluate the SIMD speedup for segment detection (step 1a). A second benchmark was done to evaluate the three steps of each algorithms.

The evaluated architectures are the Nvidia Jetson AGX ARMv8.2 Neon (AGX) for embedded system, Intel Broadwell Xeon E5-2640 v4 with AVX2 (BDW), Intel Skylake Xeon Xeon Gold 6126 with AVX512 (SKX) and AMD Epyc EPYC 7301 with AVX2 (ZEN) for workstation and server. The frequencies have been forced fixed to their nominal value.

3.2 RLE encoder

In Figure 6, we present the speedup of our best SIMD RLE encoder compared to the scalar RLE encoder that was used in the former LSL. This plot really shows all the benefits of our SIMD approach: in average most machines get a ×5 speedup. On Skylake Xeon, we achieved ×8 thanks to AVX512 (×6 without).

The SIMD RLE encoder is *constant* time while the scalar is not: the more random the input is, the slower the encoder is. It is mainly due to the branch in the scalar implementation. This explains that

the highest speedup is achieved for granularity $g = 1$ and density around $d = 50\%$, where the input values are the most random. For the same reason, the speedup is minimal for $d = 0\%$ or $d = 100\%$.

The AVX512 implementation needs 5 cycles per iteration on the shuffle unit (1 for mask `cmp`, 2 for widening and 2 for `compress`). This gives us a lower bound at $5/16 \simeq 0.31$ cpp, where our actual code runs at 0.34 cpp on SKX. This makes the AVX512 RLE encoder compute bound. *A fortiori*, the other SIMD encoders are also compute bound as they are slower.

3.3 Step processing time

The contribution of each step of the processing to the total time is shown in Figure 7 for both the reference LSL and our new SIMD LSL. We can clearly see the non-constant run time of the scalar RLE encoder mentioned above, while our SIMD RLE processing time is completely flat.

In scalar, we can see that the RLE, the unification and the relabeling steps are faster at higher granularity, while on our SIMD version, RLE is at the same level, and the relabeling is only slightly faster. In all cases, the SIMD relabeling (RLE decoder) is at least as fast as the scalar one. Both relabeling are equally fast at high granularity ($g \geqslant 8$): the compiler was able to vectorize the scalar version of the RLE decoder.

Surprisingly, our new unification U1 used in SIMD seems slower than U0 used in the reference, despite the full CCL with U1 being faster than with U0 as explained in Section 3.4. This is a consequence of the *ER* table absence. To process the U0 unification, the RLE encoder is required to compute this table *ER*. When using U1, we can skip this part of the computation in the RLE encoder. As a matter of fact, the RLE encoder without *ER* is almost 2× faster than the RLE encoder with *ER*.

We can observe that the allocation takes roughly as much time as our new RLE encoder. This means that in practice, our new version can be a few dozen percents faster in most cases. For production code, this is not an issue because the processed images will all have the same size. Actually, even if they are not the same size, we could reuse the same buffers for smaller images avoiding reallocations in most cases.

The longest step of our new SIMD implementation is the relabeling. However, this step is often not required in practice as we are usually interested in the computation of features associated to the connected components (those can be computed during unification) like the bounding box, and the first statistical raw moments. Consequently, one should be interested only in the RLE + unification timing. the relabeling step has to be considered for CC labeling in order to compare to State-of-the-Art, and should not for CC analysis.

As a side note, the relabeling step is mostly a streaming algorithm with a light processing and thus is naturally memory bound. The relabeling of 32000×32000 images achieves 10.7 GB/s on SKX, which is exactly the STREAM write bandwidth on a single core. The unification processing time is dominated by the control structure of the algorithm.

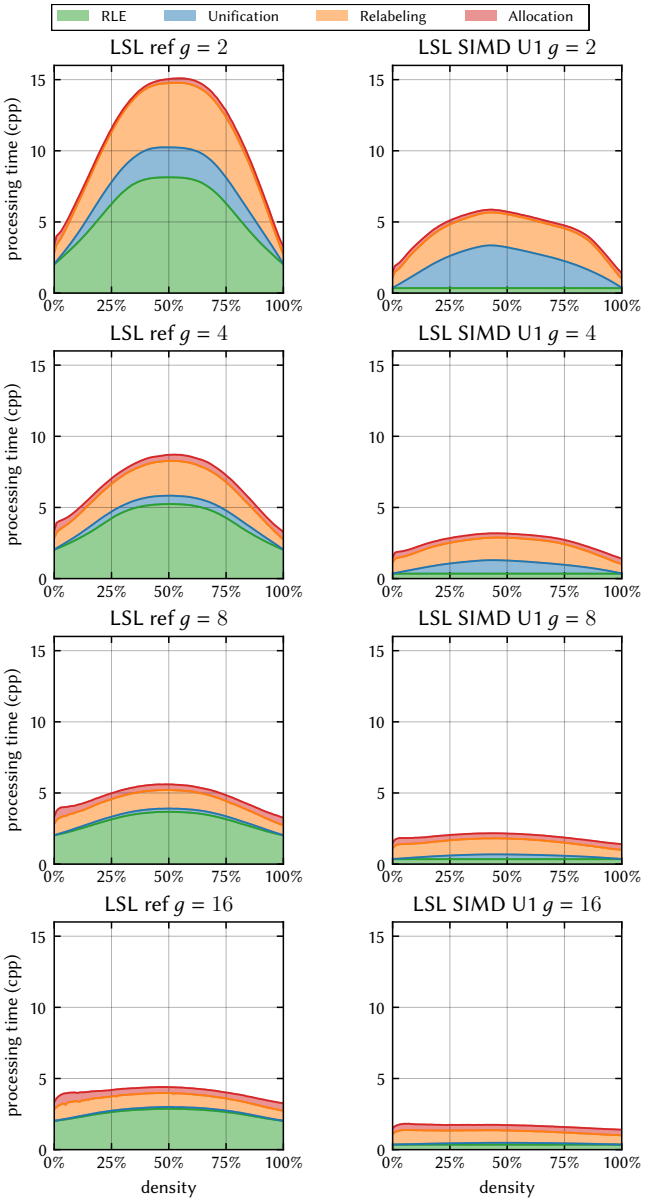

Figure 7: Impact of each step on the processing time for the reference LSL and the SIMD LSL U1 on Skylake Xeon for $g \in \{2, 4, 8, 16\}$

3.4 Granularity comparison

The processing time of the full CCL on SKX is given by Figure 8 for $g \in \{2, 4, 8, 16\}$. We can clearly see that the scalar LSL is slower than both DRAG and Spaghetti, which are themselves outperformed by our new SIMD LSL at any density. LSL SIMD is faster than Rosenfeld SIMD for $g \geqslant 2$. Rosenfeld SIMD is actually 1.30× faster than LSL SIMD for $g = 1$ in average. Indeed, lower granularity stresses run-based algorithms much more.

Our new unification U1 has roughly the same speed than U0 at $g = 2$, while it is always faster for higher granularity. As explained in the previous section, this is not because U1 is intrinsically faster,

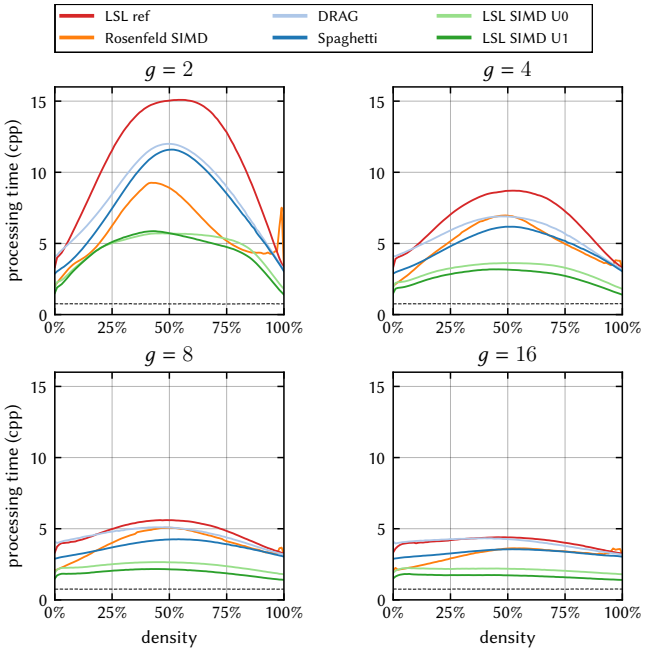

Figure 8: Processing time of the different CCL algorithms on a Skylake Xeon for 2048×2048 **with** $g \in \{2, 4, 8, 16\}$ **(the dotted line is the practical speed limit associated to memory bandwidth)**

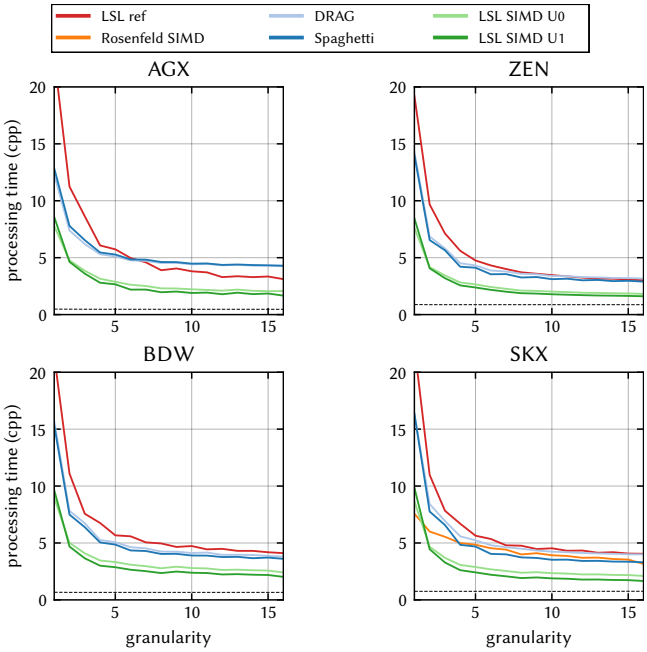

Figure 9: Average processing time (cpp) for 2048×2048 **images with** $g \in [1, 16]$ **(the dotted line is the practical speed limit associated to memory bandwidth)**

but because U1 does not require the computation of *ER*. For $g = 1$, the unification time is predominant and U0 is then 1.14× faster than U1. We can also see that the timings of all versions at $d = 50\%$ are lower at higher granularity: the input values are more predictable for the branch predictor, and the segments are fewer (each one is longer). On the opposite, the timings at $d = 0\%$ and $d = 100\%$ do not change with the granularity: an empty image remains empty at other granularity. The timings are flattened by the higher granularity.

3.5 Average processing time

Figure 9 shows for each granularity and target architecture the average processing time of the CCL algorithms. We can clearly see the following ordering: LSL ref is slower than both DRAG and Spaghetti which are both slower than SIMD U0 and U1 on all x86 machines. However, we can see that LSL ref gets faster than both DRAG and Spaghetti on AGX for $g > 6$. We believe that those algorithms under-perform on ARM because they were developed and optimized exclusively on Intel platform. One can note that on AGX, DRAG is actually a bit faster than Spaghetti while it is the opposite on all x86 platforms.

As stated earlier, we can see that Rosenfeld SIMD is faster than LSL SIMD for $g = 1$, and slower afterwards. This algorithm gets close to Spaghetti but slower for $g \geqslant 4$. We can also see that U0 is faster than U1 for $g = 1$ on all platforms. For $g > 2$, U1 is faster than U0 as already explained. The average average-timings in cpp are given in Table 1 for all densities and $g \in [1, 16]$. The same table in ms/image is given in Table 2. We can see that our best algorithm – LSL SIMD U1 – outperforms all the others on all tested platforms.

cycle/pixel	AGX	ZEN	BDW	SKX
LSL ref [16]	5.99	5.28	6.46	6.41
Rosenfeld SIMD [11]	N/A	N/A	N/A	4.47
DRAG [14]	5.33	4.60	5.38	5.59
Spaghetti [15]	5.47	4.33	5.14	4.98
LSL SIMD U0	2.94	2.69	3.44	3.04
LSL SIMD U1	**2.72**	**2.53**	**3.10**	**2.70**
LSL SIMD U1 -alloc	2.38	2.29	2.61	2.36
LSL SIMD U1 -relabeling	1.18	0.99	1.18	1.09

Table 1: Average processing time (cpp) for the full CCL for 2048×2048 **images with** $g \in [1, 16]$

ms/image	AGX	ZEN	BDW	SKX
LSL ref [16]	11.1	10.1	7.97	10.3
Rosenfeld SIMD [11]	N/A	N/A	N/A	7.21
DRAG [14]	9.87	8.76	6.64	9.02
Spaghetti [15]	10.1	8.26	6.33	8.03
LSL SIMD U0	5.45	5.12	4.24	4.90
LSL SIMD U1	**5.04**	**4.81**	**3.82**	**4.36**
LSL SIMD U1 -alloc	4.41	4.36	3.21	3.81
LSL SIMD U1 -relabeling	2.18	1.89	1.46	1.76

Table 2: Average processing time (ms) for the full CCL for 2048×2048 **images with** $g \in [1, 16]$

We achieve the complete CCL with allocations under *3 cycles per pixel* on recent platforms. If one is only interested in the features of the connected components, then they do not need relabeling and can get RLE + unification in just under 1.2 cycles per pixel, while the features will be computed on the fly.

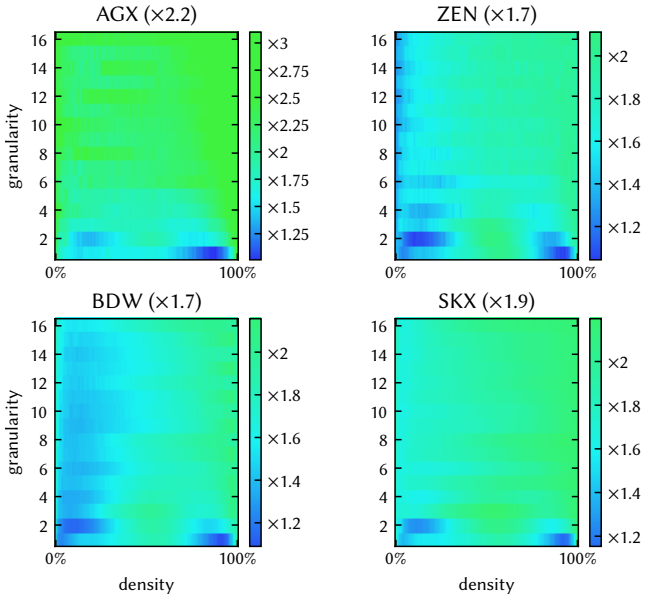

Figure 10: Speedup of LSL SIMD U1 compared against Spaghetti (with memory allocations) for $g \in [1, 16]$

3.6 Speedup against State-of-the-Art

Figure 10 shows the speedup of our best LSL SIMD against the best State-of-the-Art algorithm Spaghetti. On AVX2 machines (ZEN and BDW), we achieve an average speedup about ×1.7. On SKX, we can take advantage of AVX512 and thus achieve ×1.9. On AGX, we achieve a higher speedup about ×2.2 due to the suspected underperformance of YACCLAB algorithms on this platform.

We can see that the lowest speedup is for $g = 1$ and $d \sim 90\%$. The highest speedups are achieved for relatively high granularity ($g > 3$) and high densities.

CONCLUSION

In this paper we have introduced a SIMD version of LSL that is accelerated thanks to SIMD RLE compression and decompression. This algorithm is available for SSE, AVX, AVX512 and Neon and can be easily ported on Altivec thanks to `vec_perm` instruction or SVE (by emulating missing compress store). It has been evaluated on Intel, AMD and ARM architectures. In average, our binary RLE encoder is ×5 faster and has a data-independent processing time. The complete CCL is then from ×1.7 up to ×1.9 faster than the 2019 State-of-the-Art algorithms.

REFERENCES

[1] D. A. Bader and J. Jaja, "Parallel algorithms for image histogramming and connected components with an experimental study," *Parallel and Distributed Computing*, vol. 35,2, pp. 173–190, 1995.
[2] A. Lindner, A. Bieniek, and H. Burkhardt, "PISA - parallel image segmentation algorithms," pp. 1–10, Springer, 1999.
[3] 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*, vol. 70, pp. 25–43, 2017.
[4] 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)*, pp. 1–16, 2018.
[5] 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.
[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)*, pp. 1355–1362, IEEE, 2014.
[7] A. Rosenfeld and J. Platz, "Sequential operator in digital pictures processing," *Journal of ACM*, vol. 13,4, pp. 471–494, 1966.
[8] F. Wende and T. Steinke, "Swendsen-wang multi-cluster algorithm for the 2d/3d Ising Model on Xeon Phi and GPU," in *International Conference on High Performance Computing (SuperComputing)* (ACM, ed.), pp. 1–12, 2013.
[9] 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)*, pp. 1–8, 2016.
[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*, vol. 71, pp. 615–620, 2011.
[11] A. Hennequin, I. Masliah, and L. Lacassagne, "Designing efficient SIMD algorithms for direct connected component labeling," in *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pp. 1–8, 2019.
[12] 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*, pp. 54–58, 2015.
[13] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.
[14] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected components labeling on DRAGs," in *International Conference on Pattern Recognition (ICPR)* (IEEE, ed.), pp. 121–126, 2018.
[15] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti labeling: Directed acyclic graphs for block-based connected components labeling," *Transactions on Image Processing*, vol. PP, pp. 1–14, 2019.
[16] L. Lacassagne and A. B. Zavidovique, "Light speed labeling for RISC architectures," in *IEEE International Conference on Image Analysis and Processing (ICIP)*, 2009.
[17] 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 (JRTIP)*, vol. 15, no. 1, pp. 173–196, 2018.
[18] 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)*, pp. 1–6, 2018.
[19] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proceedings of the IEEE*, vol. 55,3, pp. 8–19, 1967.
[20] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17,6, pp. 8–19, 1984.
[21] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Transactions on Information Theory*, vol. 24,5, pp. 530,536, 1978.
[22] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," in *ACM SIGMOD Record*, vol. 27, pp. 355–366, ACM, 1998.
[23] J. Willms, "Autocorrelations of binary sequences and run structure," *Transactions on Information Theory*, vol. 59,8, pp. 4985–1993, 2013.
[24] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai, "Roaring bitmaps: Implementation of an optimized software library," *Software: Practice and Experience*, vol. 48, no. 4, pp. 867–895, 2018.
[25] A. Ungethum, J. Pietrzyk, P. Damme, D. Habich, and W. Lehner, "Conflict detection-based run-length encoding - avx-512 cd instruction set in action," in *International Conference on Data Engineering Workshops (ICDEW)*, pp. 96–101, IEEE, 2019.
[26] H. Lang, L. Passing, A. Kipf, P. Boncz, T. Neumann, and A. Kemper, "Make the most out of your simd investments: counter control flow divergence in compiled query pipelines," *Journal on Very Large Data Bases (VLDB)*, pp. 1–18, 2019.
[27] D. Lemire, "Lemire's simdprune https://github.com/lemire/simdprune," 2019.
[28] H. S. Warren, *Hacker's Delight.* Addison-Wesley Professional, 2nd ed., 2012.
[29] C. Grana, "YACCLAB https://github.com/prittt/YACCLAB," 2016.