SparseCCL: Connected Components Labeling and Analysis for sparse images

Arthur Hennequin[1,2], Ben Couturier[2], Vladimir V. Gligorov[3], Lionel Lacassagne[1]

[1]LIP6, Sorbonne Université, CNRS, Paris, France [2]CERN, Switzerland
[3]LPNHE, Sorbonne Université, Paris Diderot Sorbonne Paris Cité, CNRS/IN2P3, Paris, France
email: arthur.hennequin@lip6.fr, lionel.lacassagne@lip6.fr

***Abstract*—Connected components labeling and analysis for dense images have been extensively studied on a wide range of architectures. Some applications, like particles detectors in High Energy Physics, need to analyse many small and sparse images at high throughput. Because they process all pixels of the image, classic algorithms for dense images are inefficient on sparse data. We address this inefficiency by introducing a new algorithm specifically designed for sparse images. We show that we can further improve this sparse algorithm by specializing it for the data input format, avoiding a decoding step and processing multiple pixels at once. A benchmark on Intel and AMD CPUs shows that the algorithm is from $\times 1.6$ to $\times 2.5$ faster on sparse images.**

I. Introduction

In computer vision, Connected Component labeling (CCL) is a common and wide spread algorithm. It consists in assigning a unique label to each group of connected pixels. These groups of pixels, called Connected Components (CC), are then used for higher level tasks, like tracking, motion detection or optical character recognition. First instances of this algorithm were proposed by pioneers like Rosenfeld [16] or Haralick [6]. In High Energy Physics (HEP), CCL is used in the tracking of particles by labeling hits on the detectors' sensors to extract the real impact positions.

A CCL algorithm by itself, only provides the association of pixels, this is why it is followed by an analysis algorithm. The purpose of the analysis is to compute features of each CC, like the bounding box or the first statistical moments i order to to compute the center of gravity. If naive algorithm perform the labeling first and then the analysis, the optimized algorithms do the analysis during the labeling. These algorithms are called *Connected Component Analysis* (CCA).

Most of CCL algorithms used to be sequential ones developed on single-core processors [7] [3]. Recently, new parallel algorithms were developed for multi-core processors [13] [5], SIMD processors [18] [11] [8] and GPUs [14] [9].

These algorithms are very efficient for natural images but not for very low density images (very few pixels set to one) like those generated in HEP experiment.

The case considered here is similar to matrix algebra. When a matrix has very few non-zero value, the classical dense structure and the classical dense algorithms turn out to be inefficient. The dense structure is replaced by various flavours of lists that hold the non-zero value and specialized or dedicated algorithms are designed to process these data efficiently. In the case of tracking hits on detectors' sensors, the same phenomenon happens: even if CCA algorithms are very fast, they are inefficient to cluster and label matrices of hits with around 0.5% of hits.

The section II presents some classic connected components labeling algorithms. Section III introduces a new algorithm for connected components labeling of sparse images and its specialization for the pattern recognition of CERN's LHCb experiment. Finally, in section IV, we evaluate this new algorithm and compare it to state-of-the-art.

II. Classic algorithms for dense images

In this section, we present three classes of connected components labeling algorithms.

A. One component at a time

In this first class of algorithm, we process one connected component at a time. The image is scanned one time and, for every foreground pixel encountered, a traversal of the connected component is done to label all the pixels. This algorithm and its variants are often called *flood fill* or sometimes *seed fill*. The traversal can be done using a stack in depth-first order or a queue, in breadth-first order. Implementations of algorithms of this class are found in [17] and [1]. This algorithm can be optimized by only adding, on top of the stack, the branching pixels – ie. the pixels that have more than one non-visited neighbour – and directly processing the others. Doing so, we avoid a store and a load for these pixels. If the image is sparse and if we have a list of pixel coordinates, we can directly start at known pixel positions avoiding the read of many background pixels. However, this does not prevent the test of every pixel on the contour of the connected component and the visited pixels have to be removed from the list.

B. Iterative algorithms

The second class was introduced by Haralick [6]. Each pixel is initialized with a unique temporary label, then this label is propagated to the pixel's neighbors using local minimum or maximum propagation. The propagation step is repeated until the image of labels reaches stabilization, ie. there is no more change within the image. This algorithm was particularly

fitted for implementations on parallel architectures, due to its high regularity, before the apparition of fast scatter and gather operations needed for union-find based algorithms. The number of iterations, and thus the processing time, of iterative algorithms depends on the longest path in the image. For an $n \quad n$ spiral, the number of iterations is equal to $\frac{n^2}{2}$.

C. Direct two-pass algorithms

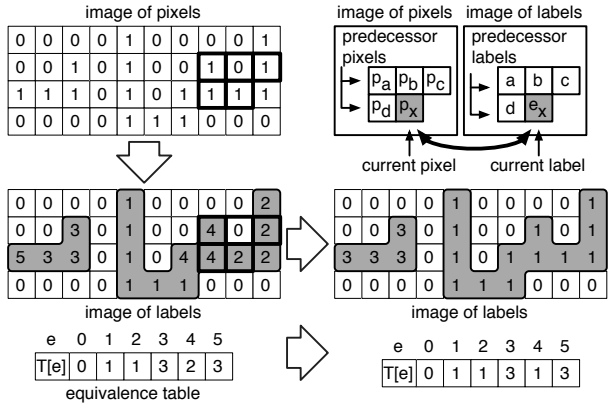

Fig. 1. Example of 8-connected CCL with Rosenfeld algorithm: binary image (top), image of temporary labels (bottom left), image of final labels (bottom right) after the transitive closure of the equivalence table.

The two-pass CCL algorithms are split into three steps and perform two image scans (like the pioneer algorithm of Rosenfeld [16]). The first scan (or first labeling) assigns a temporary label to each CC and some label equivalences are built if needed. The second step solves the equivalence table by computing the transitive closure of the graphs 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, by doing a simple look up: $I(i;j) \quad L[I(i;j)]$.

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. L is the equivalence table, e a label and r its root. The first scan of Rosenfeld is described in algorithm 1, the transitive closure in algorithm 2, while the classical union-find algorithms are provided in algorithms 3 & 4. 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 L. At the end of first scan, the equivalence table is complete and applied to the image.

III. Context and SparseCCL

In this section we present SparseCCL, a parameterizable connected components labeling and analysis algorithm for sparse images. Then, we present a specialization of the algorithm in the context of the LHCb experiment, where the very few hits of high-energy particles are scattered across the detector's sensors.

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

Input: $a;b;c;d$, four labels, p_x, the current pixel in $(i;j)$
if p_x 6= 0 **then**
  $a \quad E[i \quad 1][j \quad 1]; \quad b \quad E[i \quad 1][j]$;
  $c \quad E[i \quad 1][j + 1]; \quad d \quad E[i][j \quad 1]$;
  if $(a = b = c = d = 0)$ **then**
    $ne \quad ne + 1; \quad e_x \quad ne$;
  else
    $r_a \quad Find(L;a); \quad r_b \quad Find(L;b)$;
    $r_c \quad Find(L;c); \quad r_d \quad Find(L;d)$;
    $e_x \quad min^+(r_a;r_b;r_c;r_d)$;
    if $(r_a$ 6= 0 **and** r_a 6= $e_x)$ **then** $Union(L;e_x;r_a)$;
    if $(r_b$ 6= 0 **and** r_b 6= $e_x)$ **then** $Union(L;e_x;r_b)$;
    if $(r_c$ 6= 0 **and** r_c 6= $e_x)$ **then** $Union(L;e_x;r_c)$;
    if $(r_d$ 6= 0 **and** r_d 6= $e_x)$ **then** $Union(L;e_x;r_d)$;
else
  $e_x \quad 0$

Algorithm 2: Sequential solve of equivalences (step 2)

for e 2 $[1:n]$ **do**
  $L[e] \quad L[L[e]]$

A. General parameterizable ordered SparseCCL

In this first version of the algorithm, we assume that the image is represented by a list of active pixels ordered by their coordinates. This kind of representation allows the algorithm to take advantage of the sparse nature of the data. This is due to the size of the list scaling directly with the number of pixels to label and not the total number of pixels. Another case where this representation is useful is when the coordinate ranges are too large or if the number of dimensions makes the storage not practical. Figure 2 gives an example of such an image and its list representation.

The algorithm parameterization is done through the functions `is_adjacent` / `is_far_enough` for the labeling and `init_features` / `accumulate_features` for the analysis. The algorithm is generic enough to be adapted to n-dimension and every type of connectivity and pixel format. Algorithm 5 gives the complete algorithm.

Algorithm 3: find(L, e)

Input: e a label, L an equivalence table
Result: r, the root of e
$r \quad e$
while $L[r]$ 6= r **do**
  $r \quad L[r]$
return r

Algorithm 4: union(L, e_1, e_2)

Input: e_1, e_2 two labels, L an equivalence table
Result: e, the least common ancestor of the e's
if $e_1 < e_2$ **then**
  $e \quad e_1, L[e_2] \quad e$
else
  $e \quad e_2, L[e_1] \quad e$
return e

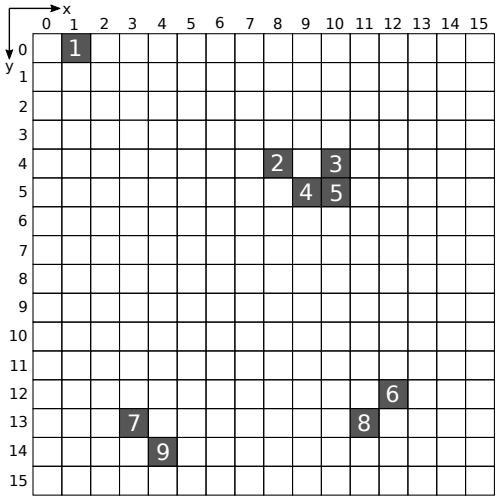

Fig. 2. Sparse binary image and its list representation. Each entry in the list is a tuple of pixel coordinates (x, y). The list is sorted in column major order and contains $n = 9$ pixels.

Algorithm 5: SparseCCL

```
1  // First scan: pixel association
2  start_j ← 0
3  for i ← 0 to n − 1 do
4      L[i] ← i
5      ai ← i
6      for j ← start_j to i − 1 do
7          if is_adjacent(pixel[i], pixel[j]) then
8              ai ← union(L, ai, find(L, j))
9          else if is_far_enough(pixel[i], pixel[j]) then
10             start_j ← start_j + 1
11 // Second scan: transitive closure and analysis
12 labels ← 0
13 for i ← 0 to n − 1 do
14     if L[i] = i then
15         labels ← labels + 1
16         l ← labels
17         init_features(l, pixel[i])
18     else
19         l ← L[L[i]]
20         accumulate_features(l, pixel[i])
21     L[i] ← l
```

SparseCCL is designed to minimize the memory footprint in order to have the best data locality and to fit in the L1 cache. The only internal memory it needs is an integer table of size n to store the equivalences. The input table containing the pixels and the output tables containing the connected components features are allocated outside of the algorithm. The algorithm is divided into two parts: the first scan where pixels are associated using an equivalence table and the second scan where we resolve equivalences by performing a transitive closure of the graph embedded in the equivalence table. We can also do an *on-the-fly* analysis of connected components in the second scan.

The first scan iterates over the pixels in the list and adds them one by one to the equivalence table. The equivalence table is an index table implementing a forest of equivalence trees. Each cell of the table corresponds to one pixel, the content of the cell is the index of the parent pixel. A pixel is a root if its entry in the equivalence table is its own index. For each pixel, the algorithm checks on previously added pixels for adjacency and merge their two equivalence trees if they are adjacent. The merging is done by calling the `union` and `find` functions described in algorithms 4 and 3. Because the list is ordered, we can keep track of a start index to avoid testing pixels that are too far from each other. With this optimization, the first scan complexity becomes $O(kn)$ instead of $O(n(n-1)/2)$, with $k << n$. The `is_adjacent` and `is_far_enough` parameterization for 2-dimensions 8-connectivity labeling is described in algorithms 6 and 7.

Algorithm 6: is_adjacent(p_1, p_2)

```
1  return |p1.x − p2.x| ≤ 1 and |p1.y − p2.y| ≤ 1
```

Algorithm 7: is_far_enough(p_1, p_2)

```
1  return p1.y − p2.y > 1
```

The second scan iterates over each temporary label in the equivalence table. If the label is a root, it creates a new connected component label by incrementing the *labels* counter and initialises the features for this connected component. If the label has a parent, it takes the parent's label and accumulates the features. Because the temporary label of a parent is always smaller than the one of the child, we know that the parent is already processed. Before continuing to the next label, we update the equivalence table with the new label. Algorithms 8 and 9 give an example of the features needed to compute the connected components center of gravity $(G_x; G_y) = (S_x/S; S_y/S)$. $(S_x; S_y)$ are the sums of x and y coordinates and S the number of pixels.

Algorithm 8: init_features(label, pixel)

```
1  sum_x[label] ← pixel.x
2  sum_y[label] ← pixel.y
3  sum_n[label] ← 1
```

Algorithm 9: accumulate_features(label, pixel)

```
1  sum_x[label] ← sum_x[label] + pixel.x
2  sum_y[label] ← sum_y[label] + pixel.y
3  sum_n[label] ← sum_n[label] + 1
```

B. Acceleration structure for unordered pixels

In some scenarios, we might not receive an ordered list of pixels as input and sorting them would already take too much time. We also cannot afford accessing a full pixel image buffer because of data locality and the time it will take to reset such

buffer between two images labeling. We compromise by doing dimension reduction: we use a table for each row and add the pixels to its corresponding row when we encounter it in the first scan. Now, when checking for adjacency, we only have to check the previous, the current and the next row table. The pixels within each row are not sorted so they have to be all checked. Each row table has a size property (N_{row}) that keeps track of how many pixels were added to the row. When resetting the tables, we only have to set the size of used rows to zero. Table I shows an example of such structure.

Row	N_{row}	Pixels (index, column)
0	1	(5, 1)
1..3	0	
4	2	(9, 8), (7, 10)
5	2	(4, 9), (1, 10)
6..11	0	
12	1	(8, 12)
13	2	(3, 3), (6, 11)
14	1	(2, 4)
15	0	

TABLE I
STRUCTURE REPRESENTING THE RECEIVED PIXELS

C. Case study: specialization for LHCb VELO Upgrade

LHCb, one of the four major experiments at the Large Hadron Collider (LHC) [2], is a general purpose spectrometer optimized for the study of particles containing b and anti-b quarks (B mesons). The experiment's detector is specifically designed to filter out these particles and the products of their decay. Each LHCb sub-detector is specialized in measuring a different characteristic of the particles produced by colliding protons. Collectively, the detector's components gather information about the identity, trajectory, momentum and energy of each generated particle, and can single out individual particles from the billions that spray out from the collision point each second. After the upcoming high luminosity upgrade, the LHC will collide protons 30 million times per second at LHCb. These collision "events" must be analysed in real time by a system called the High Level Trigger 1 (HLT1), in order to find and keep the small fraction of these events which contain B mesons.

The VELO (VErtex LOcator) sub-detector, shown in figure 3, is a high precision pixel detector surrounding the beamline where the collision occurs. It is divided into 52 L-shaped modules. Each module is itself composed by 4 sensors of 3 chips each. The chips have 256 256 pixels, so the sensors have 256 rows and 768 columns. Each pixel is a square with a length of 55 microns. The sensor pixels are packed into Super-Pixels (SP) of size 2 4 pixels, so the sensors have 64 SP rows and 384 SP columns [10] [15].

The modules are positioned along the z axis. More information about the geometry can be found in the LHCb VELO Upgrade Technical Design Report [12]. Figure 4 shows the format of a Super-Pixel (SP) encoded in a 32 bit integer. The less significant byte is a bitmask representing the pixels. Then we find, from bit 8 to 13, the row of the SP and, from bit 14

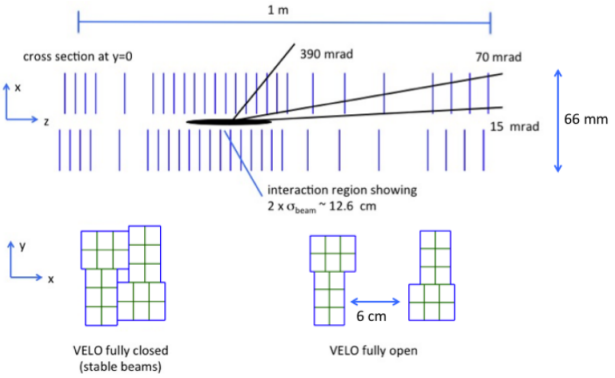

Fig. 3. CERN LHCb VELO geometry

to 22, the column of the SP. The 31^{th} bit is a flag indicating if the SP is isolated, ie. if it doesn't have any neighbor. The SP are delivered in raw banks. There is one raw bank per sensor and each one contains the number of SP in the bank followed by the encoded SP.

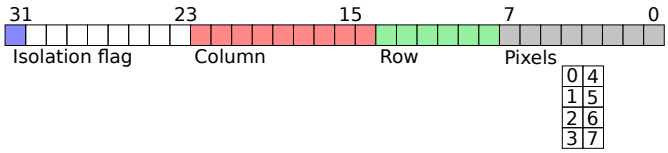

Fig. 4. CERN LHCb VELO Super-pixel format

To take advantage of the data format, we specialized the SparseCCL algorithm to label SP instead of pixels. This allows to further reduce the amount of memory needed and to skip a decoding step. We first start by preparing the data: we remove the SP that are known to be isolated and resolve them using lookup tables, for the remaining SP, we test if there is more than one CC inside and split them if necessary. Figure 5 shows the two possible configurations for a SP: one CC or two CCs. Once the SP list is prepared, we run the algorithm using a combination of bitwise operations and a lookup table to test the adjacency. Another lookup table is used for a fast computation of the first statistical moment and the number of pixels within a SP.

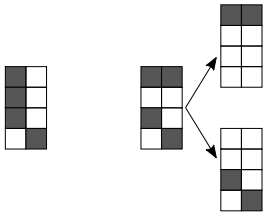

Fig. 5. A Super-Pixel containing one CC (left) and a Super-Pixel containing two CCs, split in two (right)

In 8-connectivity CCL there are eight directions of adjacency, but by using symmetries we can reduce their number to four. Figure 6 shows the configurations of SP and the pixels we have to test. Configurations a, b and c are quickly tested

using only bitwise operations. While configuration d could be tested the same way, it was found faster to use a 256-entry lookup table using the dark pixels bit pattern as the address. Configuration e shows the pixels required to take the decision to split the SP in two.

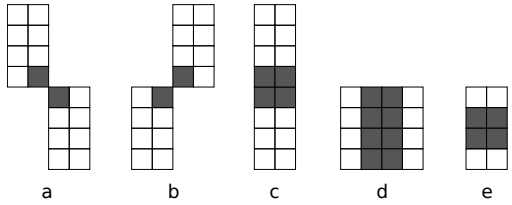

Fig. 6. a. Diagonal "forward" link, b. Diagonal "backward" link, c. Vertical link, d. Horizontal link, e. Two clusters condition

IV. Experimental Evaluation

Evaluating CCL algorithms has always been a challenge as the speed of such algorithms is data-dependent. To model natural images, we follow the approach which is proposed in [4] and generate pseudo-random noise images of varying densities and granularity. The density parameter controls, at low density, the number of pixels in the image. The granularity is the size of a macro-pixel side, it controls how clustered the pixels are and thus the minimum size of a connected component.

In the case of LHCb's VELO detector, simulation data have shown that densities of hits in the sensors are very low and granularity is around 2. This is due to charge sharing in the silicon, when a particle hits the border between 2 or 4 pixels.

Table II shows the time, in microseconds to process one image of 768 256 pixels, for state-of-the-art dense CCL algorithms [4] and sparse CCL algorithms. We observe, that while these algorithms are well optimized, a simple flood fill looking only at active pixels is 10 times faster at a density of 1%.

	0%	1%
LSL(dense)	235.3	319.9
Rosenfeld+DT (dense)	270.3	315.4
Flood fill (sparse)	0.0	29.9
SparseCCL (ordered)	0.0	31.5
SparseCCL (row table)	0.0	17.4
SparseCCL (Super-pixels)	0.0	25.0

TABLE II
Processing time of 768 256 pixels images – in microseconds – of dense and sparse algorithms at granularity $g = 1$, on an Intel Xeon Gold 6126 @2.6GHz.

In this benchmark, we measure the time in cycles per pixel (cpp). Normalizing by the number of active pixels allows to see the real impact of the increasing density: the more the connection of pixels impacts the speed, the bigger the slope of the plot will be. Normalizing by the frequency of the machine allows to abstract the frequency of the CPU for a better comparison of architectures.

We evaluate four algorithms. The first one is a flood fill algorithm as described in section II-A. While the flood fill is generally inefficient on dense images, we found that it outperforms fast implementations of iterative and two-pass algorithms on sparse images. This is due to its ability to use the pixel list information as a starting point for its connected component mapping. The other three algorithms we evaluated are variant of our algorithm: SparseCCL. The ordered by row variant is the simple case where the input is an ordered list of single pixels. The row table variant is the algorithm described in section III-B that can take an unordered list of single pixels as input. The last variant is the specialization of the algorithm for super-pixel encoding described in section III-C where we assume the list of super-pixel ordered.

Figure 7 shows the measured time in cpp for the four algorithms, for 364 768 pixels images of varying densities from 0% to 2.5% and a granularity g=1. The number of pixel n is given by the formula $n = \frac{d}{100} \quad w \quad h$, where d is the density, w the number of columns and h the number of rows. In our test configuration, the number of pixels ranges from 0 to 6988. We observe that for a granularity of 1, the ordered SparseCCL working on pixels has the best behavior at low density. The Super-Pixels variant is slower at low densities because each SP is more likely to contain only 1 pixel. It presents no advantage over the pixel versions. The row table variant starts higher than the ordered one because of the cost of table reset. The flood fill algorithm is significantly slower than other version at low density, but scales better with the number of pixels and eventually becomes faster for densities $>$ 1:8% on Intel and $>$ 2:5% on AMD.

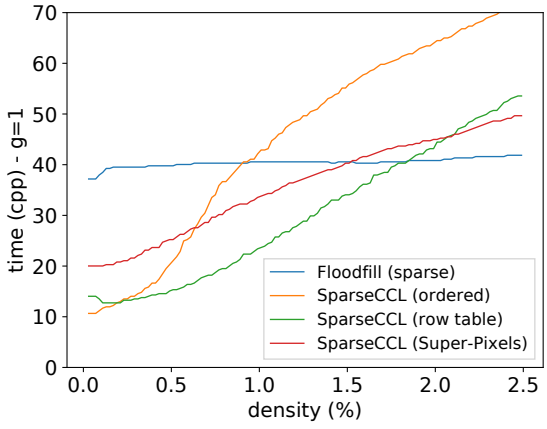

Fig. 7. Cycles Per Pixel (cpp) for the four algorithms depending on the density of the image, with a granularity g=1 on an Intel Xeon Gold 6126 @2.6GHz

Figure 8 shows the same algorithms, but with a granularity g=2. The flood fill algorithm benefits from the data locality induced by the increased granularity. On the contrary, the pixel based SparseCCL variants are slowed down by it as the

number of tests they have to perform is slightly higher. Thanks to the Super-Pixel encoding, the last variant of SparseCCL specialized for the LHCb experiment speeds up.

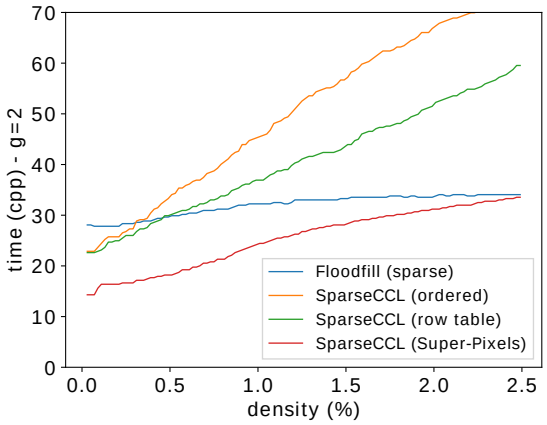

Fig. 8. Cycles Per Pixel (cpp) for the four algorithms depending on the density of the image, with a granularity g=2 on an Intel Xeon Gold 6126 @2.6GHz

Following the method described in [8], we wrote an AVX-512 version of sparseCCL. The SIMD version of the algorithm uses AVX-512's scatter and gather instructions to implement the union-find operations. For low densities, the cost of control flow is greater than the gain due to SIMD parallelism, making this SIMD version slower than the scalar one.

V. Conclusion

This article introduced a new parameterizable connected component labeling and analysis algorithm for sparse images, and a specialization of this algorithm for a practical application. We studied the impact of different versions on images of varying densities and granularities and showed that at low densities our algorithm performed better than state-of-the-art dense CCL algorithms and flood fill optimized for sparse images.

Acknowledgements

The authors would like to thank the LHCb computing and simulation teams for their support and for producing the simulated LHCb samples used in the paper. VVG is supported by ERC-CoG-724777 "RECEPT".

References

[1] A. A. AbuBaker, R. Qahwaji, S. S. Ipson, and M. S. Saleh. One scan connected component labeling technique. *2007 IEEE International Conference on Signal Processing and Communications*, pages 1283–1286, 2007.

[2] A. A. Alves, Jr. et al. The LHCb Detector at the LHC. *JINST*, 3:S08005, 2008.

[3] 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.

[4] 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.

[5] 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.

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

[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, I. Masliah, and L. Lacassagne. Designing efficient simd algorithms for direct connected component labeling. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'19, pages 4:1–4:8, New York, NY, USA, 2019. ACM.

[9] 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.

[10] K. Hennessy and LHCb VELO Upgrade Collaboration. LHCb VELO upgrade. *Nuclear Instruments and Methods in Physics Research A*, 845:97–100, Feb 2017.

[11] 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.

[12] LHCb Collaboration. LHCb VELO Upgrade Technical Design Report. Technical Report CERN-LHCC-2013-021. LHCB-TDR-013, CERN, Nov 2013.

[13] 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.

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

[15] T. Poikela, M. D. Gaspari, J. Plosila, T. Westerlund, R. Ballabriga, J. Buytaert, M. Campbell, X. Llopart, K. Wyllie, V. Gromov, M. van Beuzekom, and V. Zivkovic. VeloPix: the pixel ASIC for the LHCb upgrade. *Journal of Instrumentation*, 10(01):C01057–C01057, jan 2015.

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

[17] L. Vincent and P. Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(6):583–598, June 1991.

[18] 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.