Taming Voting Algorithms on Gpus for an Efficient Connected Component Analysis Algorithm

Florian Lemaitre, Arthur Hennequin, Lionel Lacassagne

TAMING VOTING ALGORITHMS ON GPUS FOR AN EFFICIENT CONNECTED COMPONENT ANALYSIS ALGORITHM

Florian Lemaitre[1] *Arthur Hennequin*[1,2] *Lionel Lacassagne*[1]

1: LIP6, Sorbonne Université, CNRS, Paris, France email: firstname.name@lip6.fr
2: LHCb Experiment, CERN, Geneva, Switzerland email: firstname.name@cern.ch

ABSTRACT

Connected Component Analysis is vastly used as a building block for many Computer Vision algorithms from many fields like medical image processing, surveillance, or autonomous driving. It extends Connected Component Labeling by computing some features of the connected components like their bounding box or their surface. As such, Connected Component Analysis is a voting algorithm just like histogram computation or Hough transform. Voting algorithms are difficult on many-core architectures like GPUs because of the serialization of atomic memory accesses. The trend to increase the number of cores makes this issue even more critical.

This paper explores multiple ways to reduce those conflicts for voting algorithms and especially for Connected Component Analysis. We show that our new algorithm is from 4 up to 10 times faster than State-of-the-Art on average on an Nvidia A100.

Index Terms— Voting algorithm, Connected Component Analysis, GPU, Cuda, Histogram

1. INTRODUCTION

Connected Component Labeling (CCL) is a crucial part of Computer Vision and is as old as the field [1] [2] [3]. Many applications using CCL require to compute some features for each connected component like its bounding box, its surface or its centroid. This can be used directly by the application or just used to filter out small connected components. This evolution of CCL algorithm is called *Connected Component Analysis* (CCA). CCA is used by many medical applications [4] [5] [6] [7] [8], surveillance [9] [10], autonomous driving [11] [12] and other Computer Vision applications [13] [14].

CCL on CPUs has been heavily studied and optimized [15] [16] [17] [18]. Early GPU CCL algorithms were iteratives [19] [20] [21]. The first direct CCL algorithm for GPUs was introduced by Komura [22] and improved by Playne [23] by limiting the number of unions performed.

On the other hand, parallelization of CCA is much harder as it is a voting algorithm just like histogram computation or Hough transform. This issue arises from the serialization of memory accesses and is amplified by the high number of cores of the GPUs. Therefore, while there are many hardware algorithms [24] [25] [26] [27], there is only few algorithms for multi-core CPUs [28] [29] and GPUs [30] [31].

Our contribution is the exploration of three novel ways to reduce serialization of voting algorithms on GPUs and their application to a new CCA algorithm, faster than State-of-the-Art.

2. STATE-OF-THE-ART

Concurrent voting algorithms rely on atomic read-modify-write instructions. When multiple threads vote in the same cell (ie: same memory location), their accesses are serialized in order to keep the atomic aspect of the access. For CCA, voting happens when the features of a connected component are computed by multiple threads in parallel. When a connected component is big, many threads will update the features of this component, and thus perform atomic memory accesses at the same location. This algorithm, even if parallel, performs in the same way as the equivalent sequential algorithm would, and loses all benefits from the parallelism of GPUs. The HA algorithm [31] is, to our knowledge, the only CCA algorithm that tackles this issue. Figure 1 shows the processing time to process random images on an Nvidia A100 depending on the foreground pixel density for the naive CCA approach and the optimized HA algorithm. We can clearly see that the processing time for the naive CCA algorithm is very slow after the percolation threshold at $d = 60\%$ and keeps getting worse. In fact, the maximum processing time (at $d = 100\%$) is $19\times$ higher than before the percolation threshold. At this point, the naive algorithm is fully serialized, and adding more cores will not improve the processing time. HA partially solves this issue, but an elongated peak remains at the percolation threshold: the maximum processing time (at $d = 65\%$) is still $8\times$ higher than before the percolation threshold.

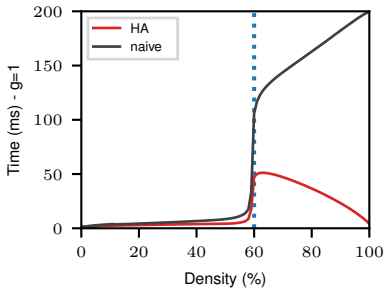


Fig. 1: Time per image as a function of image density. State-of-the-Art algorithms were run on 8192×8192 random images on a A100. Dotted line is the percolation threshold at $d = 60\%$.

The best solution to speed the histogram computation up is to have a private copy of the histogram for each thread (or at least warp), and merge them together at the end. However, this technique cannot be used for CCA as the number of cells is much higher (one cell per connected component).

3. REDUCING CONFLICTS

3.1. Full runs (FLSL)

HA [31] algorithm processes lines per bloc of 64 pixels per warp. It groups pixels into sub-runs within those blocs in order to reduce merge conflicts. Therefore a single warp processes exactly 64 pixels

Algorithm 1: Kernel for FLSL segment detection

$n \leftarrow 0$ ▷ Number of runs on the line y
$m_p \leftarrow 0$ ▷ Previous pixel mask
▷ Detect runs
for $x \leftarrow$ `laneid()` **to** *width* **by** *warp_size* **do**
  $p \leftarrow I[y \cdot width + x]$
  $m_c \leftarrow$ `__ballot_sync(ALL, p)`
  ▷ Detect edges
  $m_e \leftarrow m_c \ \hat{} \ $ `__funnelshift_l(`$m_p, m_c, 1$`)`
  $m_p \leftarrow m_c$
  ▷ Count edges before current index
  $er \leftarrow n +$ `__popc(`m_e `& lanemask_le())`
  $ER[y \cdot width + x] \leftarrow er$
  ▷ "Compress store"
  if m_e & m_l **then** $RLC[y \cdot width + er - 1] \leftarrow x$
  $n \leftarrow n +$ `count_edges(`m_e`)` ▷ same n for the whole warp
if n is odd **then**
  if $tx = 0$ **then** $RLC[y \cdot width + n] \leftarrow w$
  $n \leftarrow n + 1$
if $tx = 0$ **then** $N[y] \leftarrow n$

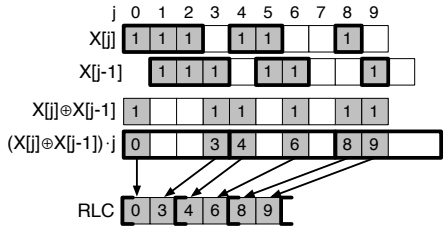


Fig. 2: Example of a segment and its associated run-length encoding with a semi-open interval $[0, 3[4, 6[8, 9[$ with a 4-wide warp `compress`.

per iteration, even if there is a single run within this bloc. Consequently, the longer the runs, the less parallelism is used. Moreover, if a run spans multiple blocks, features for this run will be updated multiple times (once per block).

In order to avoid those problems, it is possible to use *full* runs, and assign a thread per run. If a run spans the entire row, it will still be processed once and by only one thread. There are already CPU algorithms implementing those ideas: the LSL [29] and derivatives. We re-designed FLSL [18], a variant of LSL for SIMD CPU (SSE, AVX512, Neon), to target GPUs and address their architectural constraints. The crucial part is to first do a segment detection that consists in an RLE encoder and relies on "compress-store" (Figure 2). Indeed, the run boundaries are the position of the edges (when pixels change value). Compress-store can be implemented rather easily on GPUs thanks to `__ballot_sync` and `__popc` (Algorithm 1). 2-pass are required to process a single row: first detect segments (one thread per pixel), and then label segments (one thread per segment). Those passes are done in the same kernel. Like HA or naive, feature updates are done once for each segment in a dedicated kernel after the image as been labeled.

3.2. On-the-fly feature merge (OTF)

The second method to reduce contention is to take advantage of the fact that we can start to compute the features while the connected components are discovered. This requires a concurrent way to move features from a location to another while two labels are merged together.

Algorithm 2: Function for on-the-fly merge

operator `otf_merge`(l_1, l_2)
  $l_1 \leftarrow$ `find(`l_1`)`
  $l_2 \leftarrow$ `find(`l_2`)`
  `__threadfence()`
  while $l_1 \neq l_2$ **do**
    if $l_2 < l_1$ **then** swap l_1, l_2
    $l \leftarrow$ `atomicMin(`$L[l_2], l_1$`)` ▷ label merge
    `__threadfence()`
    $s \leftarrow$ `atomicExch(`$S[l_2], 0$`)` ▷ feature extraction
    `atomicAdd(`$S[l_1], s$`)` ▷ feature merge in current root
    `__threadfence()`
    if $l = l_2$ **then break**
    $l_2 \leftarrow l$
  ▷ Ensure the features have reached an actual root
  $a \leftarrow$ `find(`l_1`)`
  `__threadfence()`
  while $a \neq l_1$ **do**
    $s \leftarrow$ `atomicExch(`$S[l_1], 0$`)`
    `atomicAdd(`$S[a], s$`)`
    `__threadfence()`
    $l_1 \leftarrow a$
    $a \leftarrow$ `find(`l_1`)`
    `__threadfence()`

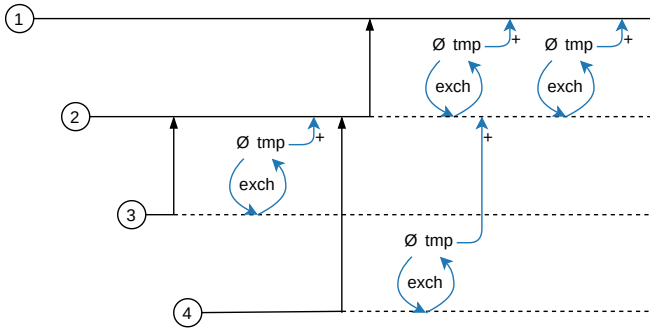


Fig. 3: Lifelines of labels during OTF merge. Solid black lines are lifelines of labels as root. Lifelines are dashed when label is no longer a root. Black arrows are equivalence recording (Unions). Blue arrows are feature movements. Chronological order is from left to right.

To do so, their instantaneous roots are retrieved, and the higher one is made point to the lower one. The features from the higher one are extracted with an `atomicExch` with 0, preventing them from being extracted multiple times. Those extracted features are then merged with the features of the lower root with an `atomicAdd`. Similarly to Komura's equivalence building [22], those steps need to be repeated if the roots have been altered by another thread (Algorithm 2).

Figure 3 shows an example of such an on-the-fly merge with a representation of the lifelines of each labels, the equivalences between labels and feature movements. On this example, the equivalence ② ≡ ① is recorded after ④ ≡ ②, but before features from ④ were merged into ②. Therefore, the thread merging ④ into ② needs to merge ② into its new root ①, otherwise, features will remain in a non-root node (ie: the accumulation will be incomplete).

While the number of updates required with OTF is actually higher than without, the number of conflicts is reduced. Indeed, the updates are done while the connected components are not yet fully discovered: threads accumulate into provisional labels rather than final roots.

Algorithm 3: Function for feature update with conflict detection

```
operator feature_update_cd(mask, l, s)
    peers ← __match_any_sync(mask, l)
    rank ← __popc(peers & lanemask_lt())
    leader ← rank = 0
    peers ← peers & lanemask_gt()
    ▷ Reduce features among peers
    while __any_sync(mask, peers) do
        next ← __ffs(peers)
        s' ← __shuffle_sync(mask, s, next)
        if next ≠ 0 then s ← s + s'
        peers ← peers & __ballot_sync(mask, rank is even)
        rank ← rank » 1
    if leader then atomicAdd(S[l], s)
```

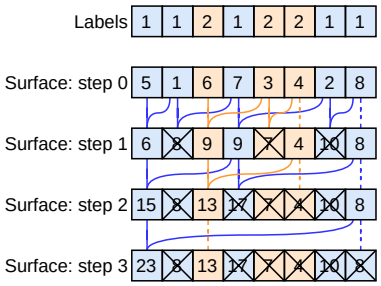


Fig. 4: Parallel masked reduction for conflict detection during surface computation.

3.3. Conflict detection (CD)

To reduce the number of collisions during feature updates, we propose the conflict detection (CD) variant, that transparently replaces the naive way of voting for computing features. Before merging the features in memory, each thread will check which thread of the warp is processing the same component. This is done with `__match_any_sync` primitive introduced in Volta GPUs. Threads will elect a leader per component, and accumulate into the leader their features with `__shfl_sync` (Figure 4). Then, only the leader actually accumulates the features for the component in memory. This way, only a single thread per warp accumulates features for a component, but multiple components can still be processed in parallel by the warp. The whole process is detailed in Algorithm 3.

This method is classified as "Opportunistic Warp-level Programming" [32] and is made possible by the `__match_any_sync` instruction. The other crucial part of this algorithm is the reduction of features into the leader. A warp can process multiple components and thus it is necessary to perform multiple reductions on distinct partitions of the warp in parallel. Few resources are available to perform such a "masked reduction" and the only mention we found is from an Nvidia blog post [33].

4. RESULTS

In order to characterize how algorithms perform, we ran all the variants proposed as well as the State-of-the-Art algorithm HA [31] on random images. For reproducible results, MT19937 [34] was used to generate images of varying density ($d \in [0\% - 100\%]$) and granularity ($g \in \{1 - 16\}$) like in [17]. A smaller granularity means more complex images with finer details.

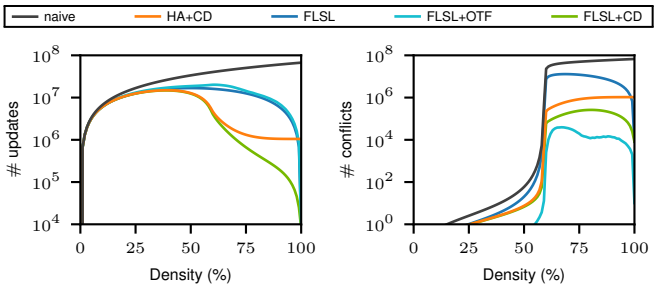


Fig. 5: Number of atomic updates and conflicts for all versions on 8192×8192 random images at $g = 1$ as a function of density. Number of conflicts is estimated from a very simple probabilistic model. Logarithmic scale is used to accommodate the wide range of values.

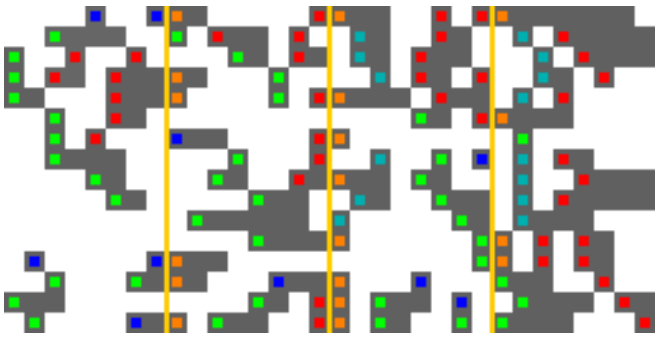
We also chose a representative set of features to benchmark. The most commonly used features are the number of pixels (ie: the surface) of the connected component (S) and the bounding box (BB), which gives the extents (min_x, min_y, max_x, max_y) of the component in the x and y directions. Those features are encoded as 32-bit integers which is enough for images smaller than 2^{32} pixels (65 536×65 536 images). Another feature of interest is the centroid that can be computed from the surface S and the first statistical raw moments S_x and S_y (respectively the sum of x and y coordinates of the pixels). Those extra features require 64-bit integers in order to handle images larger than 2048×2048 without overflows. We chose to compute all those 7 features (five 32-bit integers and two 64-bit integers) as it represents a useful set of features for many applications. Moreover, the function `connectedComponentsWithStats` from OpenCV [35] computes exactly those features which is unfortunately unavailable for GPU.

4.1. Number of updates and conflicts

We present here how the different schemes affect the number of atomic updates and more importantly the number of conflicting updates. The number of atomic updates has been precisely measured for each connected component. The number of conflicts is estimated from the number of atomic updates as the probability that two feature updates picked at random are to the same label (ie: the same memory location) multiplied by the total number of updates. If $\mathcal{U}_l$ is the number of updates of a label l, then the estimated number of conflicts is $\left(\sum_l \mathcal{U}_l^2\right) / \left(\sum_l \mathcal{U}_l\right)$.

Figure 5 shows the number of atomic updates of the features as well as the estimation of the number of conflicting updates. HA and HA+OTF have been omitted from Figure 5 as they are almost identical to FLSL and FLSL+OTF respectively. The number of atomic updates of the naive version is linear with the number of foreground pixels, and all other versions reduce the number of updates. *Full* runs (FLSL) decreases the number of updates slightly more than HA, but this effect is mainly visible for high density images. On-the-fly merges (OTF) actually increase the number of updates, especially around the percolation threshold (at $d = 60\%$). Conflict detection (CD) highly reduces the number of updates, especially after the percolation threshold. Figure 6 is a visual example of the reduction of the update number (OTF is excluded from this example because of its parallel nature).

Looking at the number of conflicts, the picture is drastically different. First, the number of conflicts before the percolation threshold is tiny for all versions, even the naive one. Then, despite the higher number of updates, OTF actually has the lowest conflict count af-



algorithm	updates count	pixels generating feature updates
naive	229	
HA	119	
FLSL	101	
HA+CD	80	
FLSL+CD	48	
lower-bound	10	

Fig. 6: Example showing the difference in feature updates of the algorithms. For the sake of demonstration, 8-connectivity is used and warps are 8-pixel wide and their vertical boundaries are represented with yellow lines (relevant only for HA algorithms).

ter the percolation threshold. Indeed, the updates are performed on different labels, hence the probability that 2 threads are conflicting decreases. The behavior of the other versions is similar to the number of atomic updates.

According to these numbers, CD and OTF are effective to reduce the number of conflicts. CD is also great at reducing the total number of updates, especially when combined with FLSL.

4.2. Processing time

We ran all versions on an Nvidia Tesla A100 GPU for 8192×8192 random images at several granularities ($g \in \{1, 4, 16\}$). For each image, the algorithm was run 20 times and the minimum processing time was taken. Throughputs are averaged over the density range $[0\% - 100\%]$.

If we look at the processing time of all the variants as a function of the density (Figure 7, left column), we can see the peak at the percolation threshold for HA and FLSL, like expected from the estimated number of conflicts. OTF is efficient after the percolation threshold ($d \geqslant 60\%$) where most conflicts are expected, but seems to suffer from having more updates before as it is slower than without. CD is efficient both before the percolation where it basically does not change the processing time, and after the percolation where the conflict reduction is useful. Looking at higher granularities (less detailed images), it appears that HA+CD is not much different from HA alone, and suffers from processing sub-runs instead of *full* runs. FLSL+CD appears to be the most effective because the conflict detection is applied on more updates than with HA+CD. The picture is mostly the same for all sizes (Figure 7, right column). In particular, FLSL+CD remains the fastest for all sizes and granularities.

In Table 1, we summarized the average throughput of various algorithms and different configurations. The naive and HA algorithms, discussed in section 2, are shown as a reference point. We first showed the impact of the OTF and CD transformations to the HA algorithm and three versions of our proposed new algorithm. The configurations have varying granularity to represent images containing different component sizes. The full image configuration represent an

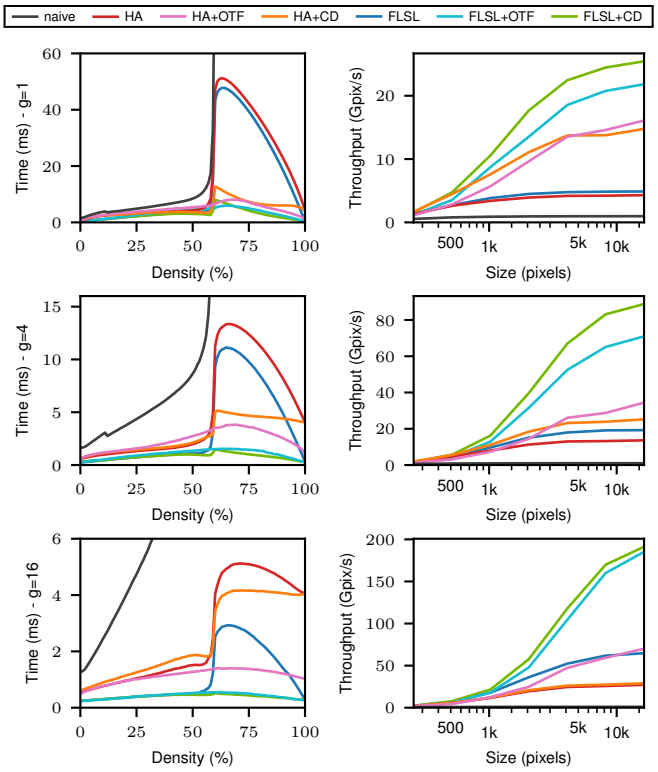


Fig. 7: Time per image for $g = \{1, 4, 16\}$ on Nvidia A100. Time versus density for 8192×8192 images (left). Average throughput versus size from 256×256 to $16\,384\times16\,384$ (right).

Algorithm	$g = 1$	$g = 4$	$g = 16$	full image
naive	0.966	0.994	0.985	0.337
HA	4.22	13.2	25.8	16.6
HA+OTF*	14.6	28.7	59.3	66.2
HA+CD*	13.8	23.9	27.4	16.6
FLSL*	4.85	19.1	61.9	**244**
FLSL+OTF*	20.8	65.1	160	238
FLSL+CD*	**24.5**	**83.2**	**170**	**244**

* : our contributions

Table 1: Average CCA throughput (Gpix/s) for 8192×8192 on an Nvidia A100

image with only one big component, filling the whole space: it is the worst case for the naive CCA algorithm as it maximize the number of memory conflicts. Indeed, both naive and HA struggle on full images, while OTF and FLSL variants achieve best throughput for full images. FLSL+CD appears to be the fastest for all the benchmarked configurations and is from 4 to 10 times faster than HA on average.

5. CONCLUSION

This article explored three new ways to overcome serialization issues of voting algorithms on GPUs. Applied to Connected Component Analysis (CCA), the combination of full runs (FLSL) and conflict detection (CD) achieves the most effective conflict reduction. Memory conflict issues are crucial to solve as the number of cores increases at each hardware generation. As a matter of facts, our new CCA algorithm (FLSL+CD) is from 4 up to 10 times faster on average than State-of-the-Art algorithm HA on an Nvidia A100.

6. REFERENCES

[1] A. Rosenfeld and J.L. Platz, "Sequential operator in digital pictures processing," *Journal of ACM*, vol. 13,4, pp. 471–494, 1966.

[2] F. Veillon, "One pass computation of morphological and geometrical properties of objects in digital pictures," *Signal Processing*, vol. 1,3, pp. 175–179, 1979.

[3] R.M. Haralick, "Some neighborhood operations," in *Real-Time Parallel Computing Image Analysis*. Plenum Press, 1981, pp. 11–35.

[4] Weijie Chen, Maryellen L Giger, and Ulrich Bick, "A fuzzy c-means (FCM)-based approach for computerized segmentation of breast lesions in dynamic contrast-enhanced mr images," *Academic radiology*, vol. 13, no. 1, pp. 63–72, 2006.

[5] Sedat Nazlibilek, Deniz Karacor, Tuncay Ercan, Murat Husnu Sazli, Osman Kalender, and Yavuz Ege, "Automatic segmentation, counting, size determination and classification of white blood cells," *Measurement*, vol. 55, pp. 58–65, 2014.

[6] Omar Abuzaghleh, Buket D Barkana, and Miad Faezipour, "Noninvasive real-time automated skin lesion analysis system for melanoma early detection and prevention," *IEEE journal of translational engineering in health and medicine*, vol. 3, pp. 1–12, 2015.

[7] Geert Litjens, Clara I Sánchez, Nadya Timofeeva, Meyke Hermsen, Iris Nagtegaal, Iringo Kovacs, Christina Hulsbergen-Van De Kaa, Peter Bult, Bram Van Ginneken, and Jeroen Van Der Laak, "Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis," *Scientific reports*, vol. 6, 2016.

[8] Nazish Khan, Imran Ahmed, Mahreen Kiran, Hamoodur Rehman, Sadia Din, Anand Paul, and Alavalapati Goutham Reddy, "Automatic segmentation of liver & lesion detection using h-minima transform and connecting component labeling," *Multimedia Tools and Applications*, vol. 79, no. 13, pp. 8459–8481, 2020.

[9] Kinjal A Joshi and Darshak G Thakore, "A survey on moving object detection and tracking in video surveillance system," *International Journal of Soft Computing and Engineering*, vol. 2, no. 3, pp. 44–48, 2012.

[10] Jennifer Salau and Joachim Krieter, "Analysing the space-usage-pattern of a cow herd using video surveillance and automated motion detection," *Biosystems Engineering*, vol. 197, pp. 122–134, 2020.

[11] Huai-Mao Weng and Ching-Te Chiu, "Resource efficient hardware implementation for real-time traffic sign recognition," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 1120–1124.

[12] Wajdi Farhat, Hassene Faiedh, Chokri Souani, and Kamel Besbes, "Real-time embedded system for traffic sign recognition based on ZedBoard," *Journal of Real-Time Image Processing*, vol. 16, no. 5, pp. 1813–1823, 2019.

[13] SL Happy and Aurobinda Routray, "Automatic facial expression recognition using features of salient facial patches," *IEEE transactions on Affective Computing*, vol. 6, no. 1, pp. 1–12, 2014.

[14] Brent E Tweddle and Alvar Saenz-Otero, "Relative computer vision-based navigation for small inspection spacecraft," *Journal of Guidance, Control, and Dynamics*, vol. 38, no. 5, pp. 969–978, 2015.

[15] S. Gupta, D. Palsetia, M.M. Ali Patwary, A. Agrawal, and A. Choudhary, "A new parallel algorithm for two-pass connected component labeling," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2014, pp. 1355–1362.

[16] 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.

[17] 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*, pp. 1–16, 2018.

[18] F. Lemaitre, A. Hennequin, and L. Lacassagne, "How to speed connected component labeling up with SIMD RLE algorithms," in *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*, 2020, pp. 1–8.

[19] G. Ziegler and A. Rasmusson, "Efficient volume segmentation on the GPU," in *GPU Technology Conference*, Nvidia, Ed., 2010, pp. 1–44.

[20] J. Barnat, P. Bauch, L. Brim, and M. Ceška, "Computing strongly connected components in parallel on CUDA," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 544–555.

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

[22] 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.

[23] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1217–1230, 2018.

[24] M. Klaiber, D. Bailey, and S. Simon, "A single cycle parallel multi-slice connected components analysis hardware architecture," *Journal of Real-Time Image Processing*, 2016.

[25] Chen Zhao, Guodong Duan, and Nanning Zheng, "A hardware-efficient method for extracting statistic information of connected component," *Journal of Signal Processing Systems*, vol. 88, no. 1, pp. 55–65, 2017.

[26] Jia Wei Tang, Nasir Shaikh-Husin, Usman Ullah Sheikh, and Muhammad N Marsono, "A linked list run-length-based single-pass connected component analysis for real-time embedded hardware," *Journal of Real-Time Image Processing*, vol. 15, no. 1, pp. 197–215, 2018.

[27] Fanny Spagnolo, Stefania Perri, and Pasquale Corsonello, "An efficient hardware-oriented single-pass approach for connected component analysis," *Sensors*, vol. 19, no. 14, 2019.

[28] Eril Mozef, Serge Weber, Jamal Jaber, and Etienne Tisserand, "Parallel architecture dedicated to connected component analysis," in *Proceedings of 13th International Conference on Pattern Recognition*. IEEE, 1996, vol. 4, pp. 699–703.

[29] 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*, vol. 15, pp. 173–196, 2018.

[30] L. Riha and M. Mareboyana, "GPU accelerated one-pass algorithm for computing minimal rectangles of connected components," in *IEEE Workshop on Applications of Computer Vision*, 2010, pp. 479–484.

[31] A. Hennequin, L. Lacassagne, L. Cabaret, and . Meunier, "A new direct connected component labeling and analysis algorithms for GPUs," in *Conference on Design and Architectures for Signal and Image Processing*. IEEE, 2018, pp. 76–81.

[32] Yuan Lin and Vinod Grover, "`https://devblogs.nvidia.com/using-cuda-warp-level-primitives/`," 2018.

[33] Elmar Westphal, "`https://developer.nvidia.com/blog/voting-and-shuffling-optimize-atomic-operations/`," 2015.

[34] M. Matsumoto and T. Nishimura, "Mersenne twister web page: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`," .

[35] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.