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

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

LIP6, Sorbonne University, CNRS, France [1]
LHCb experiment, CERN, Switzerland [2]

ICASSP 2021

SCIENCES
SORBONNE
UNIVERSITÉ

cnrs

CERN

Voting algorithms

- A voting algorithm, for each piece of data, updates a counter which depends on the piece of data being processed
  - Histogram, Hough transform, Connected Component Analysis
- Parallel voting algorithms require concurrent counter updates
  - atomic Read-Modify-Write instructions
  - if multiple accesses are on the same counter, they are serialized
- Common techniques to accelerate voting algorithms:
  - privatization: threads have local counters they can update without serialization → only for low number of counters
  - caching: threads can keep a recently accessed counter in a software cache in case it is accessed soon. The global counter is updated only when the cached counter is evicted, but has a high overhead
  - partial Access: all threads process the whole data, but update only a part of the counters → low parallel efficiency if data is large

What are Connected Component Labeling and Analysis ?

Connected Components **L**abeling (CC**L**) consists in assigning a unique number (label) to each connected component of a binary image to cluster pixels

Connected Components **A**nalysis (CC**A**) consists in computing some features associated to each connected component like the bounding box $[x_{min}, x_{max}]$ x $[y_{min}, y_{max}]$, the sum of pixels S, the sums of x and y coordinates Sx, Sy

gray level image

binary level image (segmentation by (motion detection)

connected component labeling

connected component analysis

- seems easy for a human being who has a global view of the image
- ill-posed problem: the computer has only a local view around a pixel (neighborhood)

Direct Connected Component Labeling

Direct algorithms are based on Union-Find structure (represent equivalences by a forest of trees stored in the table T):

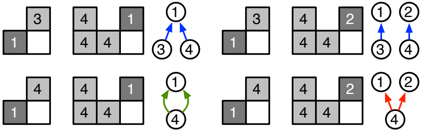
- find(e, T) search for the root of e
- union(e_1, e_2, T) join the trees containing e_1 and e_2
- flatten(T) flatten all the trees in T (all vertices point to their root)

Rosenfeld algorithm [1] is the first 2-pass algorithm with an equivalence table:

- First pass: scan the image (raster order) to create temporary labels and build the equivalence table
- Transitive closure: flatten T
- Second pass: relabel the image (replace temporary labels with their root)

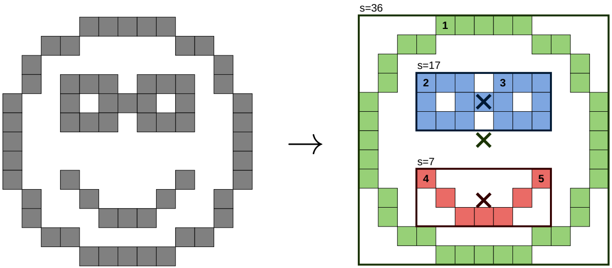
Parallel merge in union-find can lead to concurrency issues.

- Bottom-right case: 4 has to take the value 1 and 2 simultaneously: conflict!
- lock-free union by Komura [2] and improved by Playne and Hawick [3]



Connected Component Analysis

- Compute features for each connected component
  - Surface (number of pixels): S
  - Bounding box: $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$
  - Centroid: $(x_G, y_G) = (S_x, S_y)/S$
- Features are stored per label in separate arrays (Struct of Arrays)
  - Temporary labels make “holes” within feature tables



		1	2	3	4	5
	T	1	2	3	4	5
F	S	36	17		7	
	S_x	216	102		42	
	S_y	216	68		64	
	X_{min}	0	3		3	
	Y_{min}	0	3		8	
	X_{max}	12	9		9	
	Y_{max}	12	5		10	

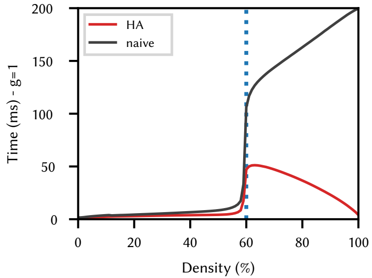
For the following explanations and examples, only S is shown.

Naive Feature Computation

- Post-processing of regular CCL
  - Each pixel vote in an array *S* at the index given by its label

Algorithm 1: Naive Feature Computation

for $y = 0 : h - 1$ **do** $\triangleright$ parallel
  for $x = 0 : w - 1$ **do** $\triangleright$ parallel
    if $I[y \cdot width + x] \neq 0$ **then**
      $e \leftarrow E[y \cdot width + x]$
      `atomicAdd`$(\&S[e], 1)$



State-of-the-Art Feature Computation on 8192×8192 random images on an A100

- serialization of *atomic* accesses on same label are as slow as sequential for the full image (all ones): atomics do not scale

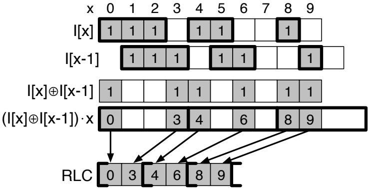
- We propose and explore three ways to reduce serialization of votes for CCA:
  - Run-Length Encoding (full segments, RLE)
  - Conflict detection
  - On-the-fly Feature Computation

Full runs: FLSL (Faster LSL)

Based on the CPU algorithm with the same name [4] and expands the use of runs from HA [5].

- labels and features are shared with all pixels of a run: one single vote per run
- full runs allow even more update reduction compared to HA
- does not lose parallelism with long runs

- performs a per-line RLE compression
- "compress-store"



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`.

Algorithm 2: Kernel for FLSL segment detection

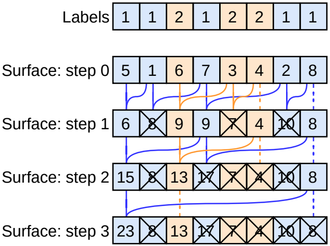
```
n ← 0 ▷ Number of runs on the line y
m_p ← 0 ▷ Previous pixel mask
▷ Detect runs
for x ← laneid() to width by warp_size do
    p ← I[y · width + x]
    m_c ← __ballot_sync(ALL, p)
    ▷ Detect edges
    m_e ← m_c ^ __funnelshift_l(m_p, m_c, 1)
    m_p ← m_c
    ▷ Count edges before current index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

Conflict Detection

- When threads vote to update features, we can detect which threads of a warp access the same label thanks to `__match_any_sync`
- Perform an in-register reduction for all threads updating the same label
  - tree-based reduction with non-contiguous lanes (eg: [6])
- Only a single thread per label will update the feature in global memory

Algorithm 3: Function for feature update with conflict detection

```
operator feature_update_cd(mask, e, s)
    peers ← __match_any_sync(mask, e)
    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) ▷ Reduction step
        if next ≠ 0 then s ← s + s'
        peers ← peers & __ballot_sync(mask, rank is even)
        rank ← rank >> 1
    ▷ Only the leader updates the features
    if leader then atomicAdd(&S[e], s)
```

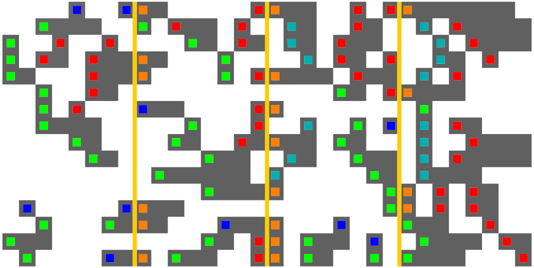


Parallel masked tree-based reduction for conflict detection during surface computation.

Conflict Detection: example

Example showing the different number of updates for various algorithms

- HA and FLSL vote only once per segment
  - HA segments are limited by the tile border (yellow line)
- Conflict Detection remove redundant updates on the same line
- "lower bound" is one single vote per connected component



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

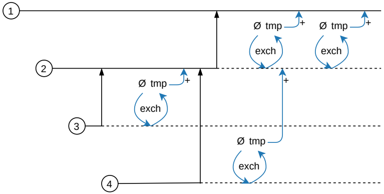
On-the-fly Feature update: concurrent algorithm

Algorithm 4: Concurrent on-the-fly feature update

```
operator otf_merge(e1, e2)
    e1 ← Find(e1)
    e2 ← Find(e2)
    __threadfence()
    while e1 ≠ e2 do
        if e2 < e1 then swap e1, e2
        e ← atomicMin(&T[e2], e1)  ▷ label merge
        __threadfence()
        s ← atomicExch(&S[e2], 0)  ▷ feature extraction
        atomicAdd(&S[e1], s)  ▷ feature merge in current root
        __threadfence()
        if e = e2 then break
        e2 ← e

    ▷ Ensure the features have reached an actual root
    a ← Find(e1)
    __threadfence()
    while a ≠ e1 do
        s ← atomicExch(&S[e1], 0)
        atomicAdd(&S[a], s)
        __threadfence()
        e1 ← a
        a ← Find(e1)
        __threadfence()
```

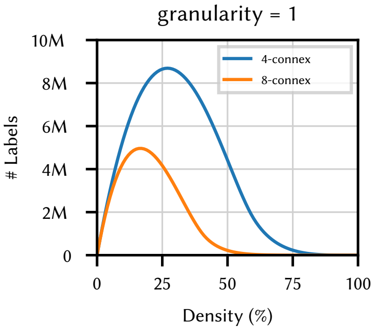
- Compute features for temporary labels and move features along the way when label unions are recorded
- Enhancement of Komura/Playne equivalence to support feature moves: same lock-free guarantee
- Tree based reduction that follows the Union-Find structure
- Correctness of the algorithm rely on precise `__threadfence` positioning

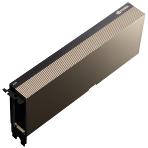


Example of 3 concurrent merges: 3 ≡ 2, 4 ≡ 2 and 2 ≡ 1. Lifelines of labels during OTF merge. Solid black lines are lifelines of labels as root. Lines are dashed when label is no longer a root. Black arrows are equivalence recording (Unions). Blue arrows are feature movements. Chronological order is from left to right.

Benchmark methodology

- random 8192×8192 (8k) images of varying density (0% - 100%), granularity (1 - 16, granularity = 4 close to natural image complexity)
- percolation threshold: transition from many smalls CCs to few larges CCs
  - 8C: density = 40%
  - 4C: density = 60%

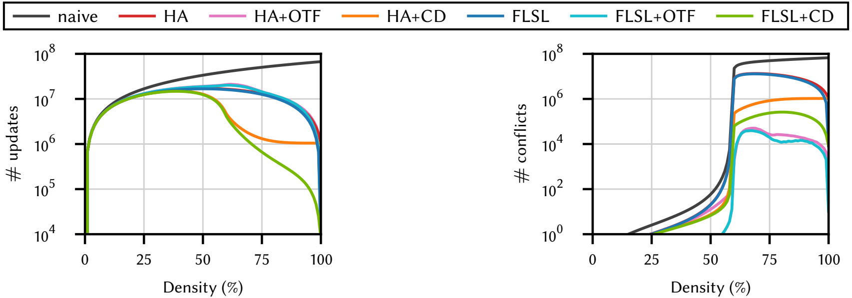




A100
2020

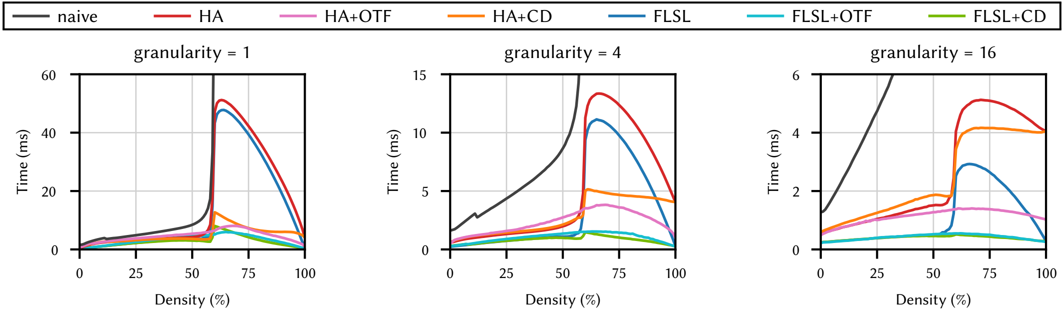
Ampere
6912 CUDA cores
1.41 GHz

Number of conflicts: theoretical analysis



- Naive number of updates is linear with the density
- HA and FLSL have roughly the same number of updates/conflicts
  - For density $\sim 100\%$, FLSL have less updates
- Number of conflicts is low before the percolation threshold ($d = 60\%$)
- OTF is the most effective to reduce the number of conflicts
  - Despite the small increase in number of updates
- CD highly reduce both updates and conflicts after the percolation threshold
  - it has almost no impact before it

A100 Density performance



- FLSL alone is effective only for high granularity (low detail images)
- Both CD and OTF are effective at mitigating serialization
- OTF shows a small overhead
- Even combined with either CD or OTF, HA still suffers from the lost of parallelism due to its partial segment nature.

$\Rightarrow$ **FLSL+CD is the most effective combination**

Average throughput

Algorithm	$g = 1$		$g = 4$		$g = 16$		full image	
naive	**0.966**	($\times$0.23)	**0.994**	($\times$0.08)	**0.985**	($\times$0.04)	**0.337**	($\times$0.02)
HA	4.22	($\times$1)	13.2	($\times$1)	25.8	($\times$1)	16.6	($\times$1)
HA+OTF*	14.6	($\times$3.5)	28.7	($\times$2.2)	59.3	($\times$2.3)	66.2	($\times$4.0)
HA+CD*	13.8	($\times$3.3)	23.9	($\times$1.8)	27.4	($\times$1.1)	16.6	($\times$1.0)
FLSL*	4.85	($\times$1.1)	19.1	($\times$1.4)	61.9	($\times$2.4)	**244**	($\times$15)
FLSL+OTF*	20.8	($\times$4.9)	65.1	($\times$4.9)	160	($\times$6.2)	238	($\times$14)
FLSL+CD*	**24.5**	($\times$5.8)	**83.2**	($\times$6.3)	**170**	($\times$6.6)	**244**	($\times$15)

* : our contributions

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

When the image is completely white (foreground), the naive version becomes completely serial

- Naive version poorly uses the parallelism of high-end GPUs due to the extreme serialization of atomic memory accesses
- All feature updates are fully serialized and all the benefits from parallelism have vanished
- compared to the first direct (and naive) algorithm, FLSL+CD achieves a $\times$700 speedup and is always the most effective in average

Conclusion

- we achieved our goal to overcome the serialization when computing the features by reducing the number of conflicting memory accesses
- three new techniques:
  - FLSL: *Faster LSL* with RLE, which is the natural extension of HA with full runs
  - OTF: merging features *On-The-Fly* during the merging of the connected components
  - CD: *Conflict Detection* within a warp
- FLSL+CD outperforms all existing implementations
  - from $\times 5$ up to $\times 15$ faster than State-of-the-Art
- As the CCA is finally very efficient for all granularities and densities, we plan to develop a 3D version for medical imaging.

Thank you!

Parallel State-of-the-art on CPU

- Parallel Light Speed Labeling(LSL) [7](L. Cabaret, L. Lacassagne, D. Etiemble) (2018)
  - parallel algorithm for CPU
  - based on RLE (Run Length Encoding) to speed up processing and save memory accesses
  - current fastest CC**A** algorithm on CPU
- FLSL = Faster LSL [4](F. Lemaitre, A. Hennequin, L. Lacassagne) (2020)
  - SIMD algorithm for CPU
  - based on RLE (Run Length Encoding) to speed up processing and save memory accesses
  - current fastest CC**L** algorithm on CPU

Parallel State-of-the-art on GPU

- Playne-Equivalence [3](D. P. Playne, K.A. Hawick) (2018)
  - *direct* CC**L** algorithm for GPU (2D and 3D versions)
  - based on the analysis of local pixels configuration to avoid unnecessary and costly atomic operations to save memory accesses.
- HA32/64 [5](A. Hennequin, Q. L. Meunier, L. Lacassagne, L. Cabaret) (2018)
  - *direct* CC**L** and CC**A** algorithm for GPU (2D 4-connexe)
  - use warp level intrinsics and sub-segment data structure to save memory accesses.
- BKE [8](S. Allegretti, F. Bolelli, and C. Grana) (2019)
  - *direct* CC**L** for GPU (8-connexe)
  - use 2×2 blocks

➧ only HA tackles CC**A** implementation

References I

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

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.

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

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*, pp. 1–8, 2020.

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.

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

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*, no. 15,1, pp. 173–196, 2018.

S. Allegretti, F. Bolelli, and C. Grana, "Optimized block-based algorithms to label connected components on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 423–438, 2019.

Direct algorithms are based on Union-Find structure

Algorithm 5: Rosenfeld labeling algorithm

```
for y = 0 : h − 1 do
    for x = 0 : w − 1 do
        if I[y][x] ≠ 0 then
            e1 ← E[y − 1][x]
            e2 ← E[y][x − 1]
            if (e1 = e2 = 0) then
                ne ← ne + 1
                e ← ne
            else
                r1 ← Find(e1, T)
                r2 ← Find(e2, T)
                e ← min+(r1, r2)
                if (r1 ≠ 0 and r1 ≠ e) then T[r1] ← e
                if (r2 ≠ 0 and r2 ≠ e) then T[r2] ← e
        else
            e ← 0
        E[y][x] ← e
```

Algorithm 6: Find(e, T)

```
while T[e] ≠ e do
    e ← T[e]
return e ▷ the root of the tree
```

Algorithm 7: Union(e_1, e_2, T)

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

Algorithm 8: Transitive Closure

```
for i = 0 : ne do
    T[e] ← T[T[e]]
```

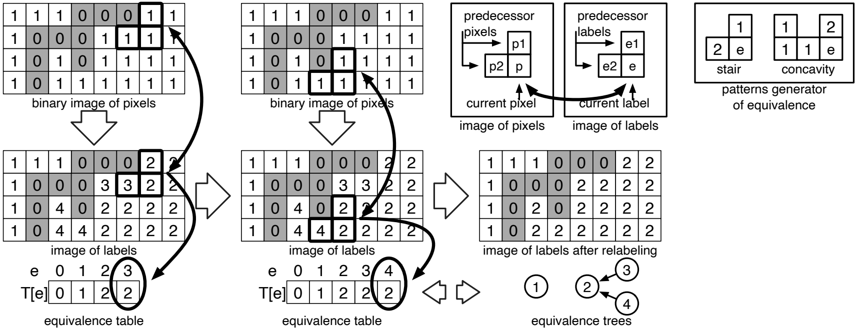
Parallel algorithms **have to do**:

- sparse addressing ⇒ scatter/gather SIMD instructions (AVX512/SVE)

Classic direct algorithm: Rosenfeld

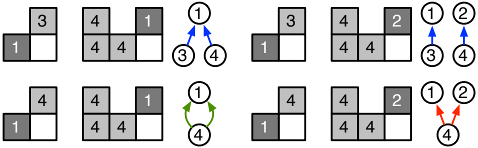
Rosenfeld algorithm is the first 2-pass algorithm with an equivalence table

- when two labels belong to the same component, an equivalence is created and stored into the equivalence table T
- eg: there is an equivalence between 2 and 3 (stair pattern) and between 4 and 2 (concavity pattern)
- stair and concavity are the only two two patterns generating equivalence
- here, background in gray and foreground in white, 4-connectivity algorithm



Equivalence merge & concurrency issue

The direct CC**L** algorithms rely on Union-Find to manage equivalences
A parallel merge operation can lead to concurrency issues:



- 1^{st} example (top-left): no concurrency, $T[3] \leftarrow 1$, $T[4] \leftarrow 1$
- 2^{nd} example (top-right): no concurrency, $T[3] \leftarrow 1$, $T[4] \leftarrow 2$
- 3^{rd} example (bottom-left): benign concurrency, $T[4] \leftarrow 1$, $T[4] \leftarrow 1$
- 4^{th} example (bottom-right): concurrency issue, $T[4] \leftarrow 1$, $T[4] \leftarrow 2$
  - 4 can't be equal to 1 and 2
  - $\Rightarrow$ 4 has to point to 1 *and* 2 has to point to 1 too...

Equivalence merge: lock-free based *concurrent* implementation

The merge function, introduced by Komura and enhanced by Playne and Hawick, solves the concurrency issues by *iteratively* merging labels using atomic operations in a lock-free scheme

Algorithm 9: merge(T, e_1, e_2)

while $e_1 \neq e_2$ **and** $e_1 \neq T[e_1]$ **do**
    $e_1 \leftarrow T[e_1]$ ▷ root of e_1
while $e_1 \neq e_2$ **and** $e_2 \neq T[e_2]$ **do**
    $e_2 \leftarrow T[e_2]$ ▷ root of e_2

▷ "Compare And Swap" loop
while $e_1 \neq e_2$ **do**
    if $e_2 < e_1$ **then** swap e_1, e_2
    $e \leftarrow$ `atomicMin`($\&T[e_2], e_1$) ▷ Convergence is faster with `atomicMin` than `atomicCAS`
    if $e = e_2$ **then** $e_2 \leftarrow e_1$
    else $e_2 \leftarrow e$

By definition, $e \leqslant T[e_2]$, so:

- if $e = e_2$: no concurrent write, update of T is successful, terminates the loop
- if $e < e_2$: concurrent write, T was updated by another thread, need to merge e and $e1$

State-of-the-Art: Hardware Accelerated (HA)

The algorithm is divided into 3 kernels:

- strip labeling: the image is split into horizontal strips of 4 rows. Each strip is processed by a block of 32×4 threads (one warp per row). Only the head of a sub-run (sub-segment) is labeled
- border merging: to merge the labels on the horizontal borders between strips
- relabeling / features computation: to propagate the label of each sub-run to the pixels or to compute the features associated to the connected components

HA algorithm uses sub-runs (compared to pixel-based algorithms) to reduce number of updates, but:

- runs cannot span multiple tiles
- maximal run-length is limited to tile width (64)

HA is the only State-of-the-Art algorithm that reduces the number of atomic accesses in order to reduce conflicts (GTC 2019)

On-the-fly Feature update: sequential algorithm

Algorithm 10: Sequential on-the-fly feature update

operator otf_merge(e_1, e_2)
  $e_1 \leftarrow \mathrm{Find}(e_1)$
  $e_2 \leftarrow \mathrm{Find}(e_2)$
  if $e_1 \neq e_2$ **then**
    if $e_2 < e_1$ **then** swap e_1, e_2
    $T[e_2] \leftarrow e_1$
    $s \leftarrow S[e_2]$ ▷ extract feature
    $S[e_2] \leftarrow 0$ ▷ reset feature
    $S[e_1] \leftarrow S[e_1] + s$ ▷ merge feature

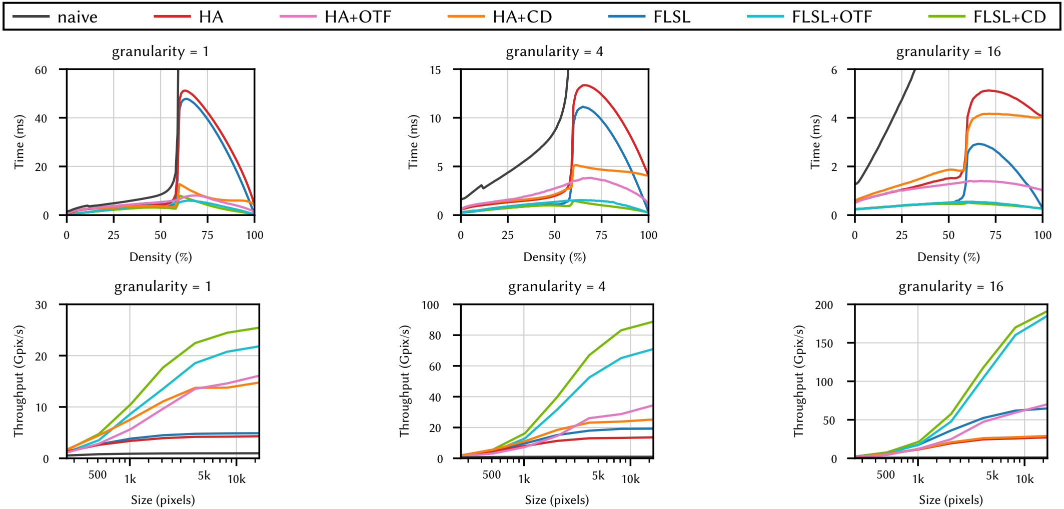
- Compute features for temporary labels and move features along the way when label unions are recorded
- Tree based reduction that follows the trees from Union-Find
- Updates are spread on all the temporary labels of a component instead being concentrated only in the final root
- More work is required as features need to be first computed for each temporary labels, and extracted

Emulation of __match_any_sync

Algorithm 11: Emulation of `__match_any_sync`

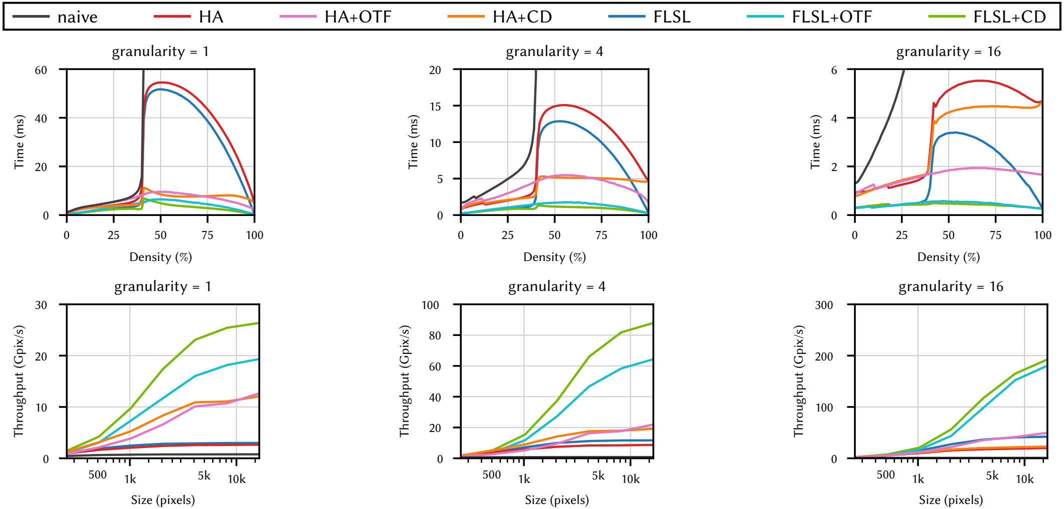
```
operator match_any_sync(mask, v)
    ▷ Thread must be in mask
    if not (mask & lanemask_eq()) then return 0
    ballot ← 0
    do ▷ One iteration per distinct value
        ▷ Remove all threads from previously find group
        mask ← mask & ~ballot
        ▷ Find the first thread among the remaining ones
        leader ← __ffs(mask) − 1
        ▷ Broadcast the value of the leader
        ref ← __shfl_sync(mask, v, leader)
        ▷ Mask of all threads having the same value as the leader
        ballot ← __ballot_sync(mask, v = ref)
    while not (ballot & lanemask_eq())
    return ballot
```

A100 performance 4-connex



Processing time (ms/img) for 8192×8192 and Throughput (Gpix/s) on A100 (4-connex)

A100 performance 8-connex



Processing time (ms/img) for 8192×8192 and Throughput (Gpix/s) on A100 (8-connex)