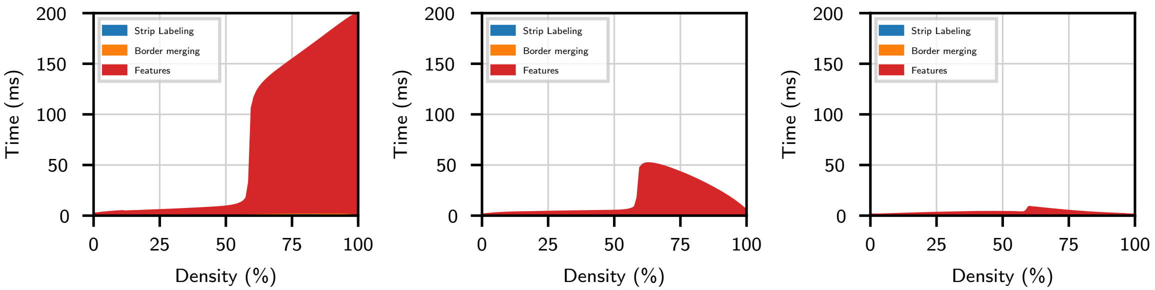
Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm

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

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

GTC 2021

Teaser



(a) Naive processing **(b)** GTC 2019: State-of-the-Art **(c)** GTC 2021: This session

Processing time of Connected Component Analysis on 8192×8192 random images

- Almost all the time is spent in feature computation (core of the algorithm) (third step of algorithm)
- Naive and State-of-the-Art are slow after 60%

What are Connected Component **L**abeling and **A**nalysis ?

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 algorithms are based on Union-Find structure

Algorithm 1: 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 2: Find(e, T)

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

Algorithm 3: 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 4: Transitive Closure

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

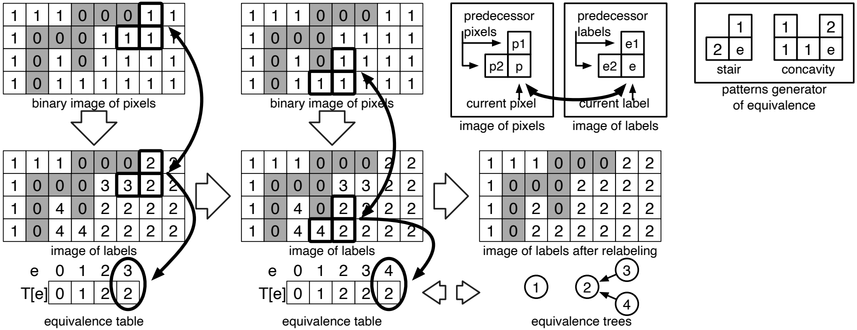
Parallel algorithms **have to do**:

- sparse addressing ⇒ scatter/gather SIMD instructions (AVX512/SVE)
- concurrent min computation ⇒ lock-free union (CUDA)

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



Parallel State-of-the-art on CPU

- Parallel Light Speed Labeling(LSL)[1](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[2](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.

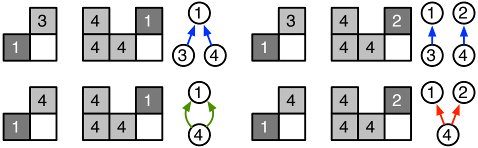
- HA32/64[4](A. Hennequin, Q. L. Meunier, L. Lacassagne, L. Cabaret) (2018)
  - *direct* CC**L** and CC**A** algorithm for GPU (2D 4-connexe)
  - use warp level intrinsics and sub-segment data structure to save memory accesses.

- BKE[5](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

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 5: 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:

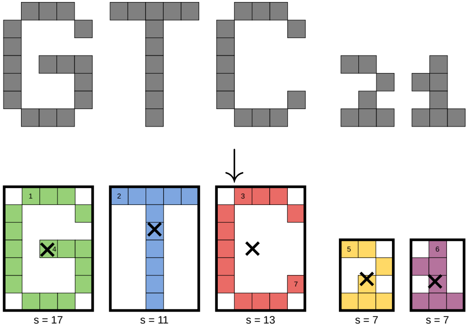
- if $e = e_2$: no concurrent write, update of T is successful, terminates the loop
- if $e < e_2$: concurrent write, T was updated by another thread, need to merge e and $e1$

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

Connected Component Analysis: data structure

- 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	6	7
T	1	2	3	1	5	6	3
F: S	17	11	13		7	7	
F: Sx	33	88	176		140	167	
F: Sy	52	21	39		33	34	
F: Xmin	0	6	12		19	23	
F: Ymin	0	0	0		3	3	
F: Xmax	4	10	16		21	25	
F: Ymax	6	6	6		6	6	

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

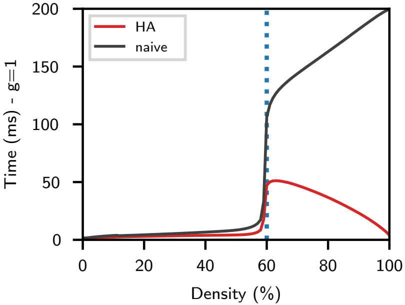
Naive Feature Computation

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

Algorithm 6: Naive Feature Computation

```
for y = 0 : h − 1 do ▷ parallel
    for x = 0 : w − 1 do ▷ parallel
        if I[y · width + x] ≠ 0 then
            e ← E[y · width + x]
            atomicAdd(&S[e], 1)
```

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



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

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

State-of-the-Art: **H**ardware **A**ccelerated (HA)

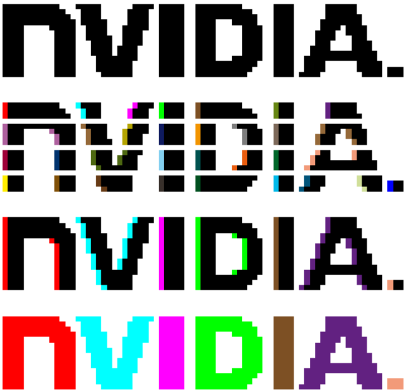
The algorithm is divided into 3 kernels:

- 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)

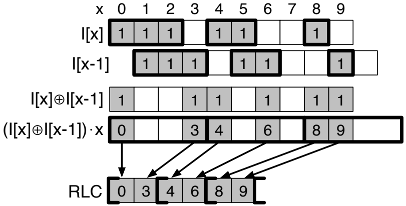


Full runs: FLSL (Faster LSL)

Based on the CPU algorithm with the same name[2] and expands the use of runs from HA.

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

- 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 7: Kernel for FLSL segment detection

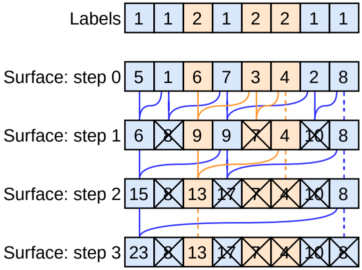
$n \leftarrow 0$ ▷ Number of runs on the line y
$m_p \leftarrow 0$ ▷ Previous pixel mask
▷ Detect runs
for $x \leftarrow$ `laneid()` **to** *width* **by** *warp_size* **do**
  $p \leftarrow I[y \cdot width + x]$
  $m_c \leftarrow$ `__ballot_sync(ALL,` p`)`
  ▷ Detect edges
  $m_e \leftarrow m_c$ ^ `__funnelshift_l(`$m_p, m_c, 1$`)`
  $m_p \leftarrow m_c$
  ▷ 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** $N[y] \leftarrow n$

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`
  - We provide an emulation of `__match_any_sync` for pre-Volta architectures
- 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 8: Function for feature update with conflict detection

operator feature_update_cd($mask, e, s$)
- $peers \leftarrow$ `__match_any_sync`($mask, e$)
- $rank \leftarrow$ `__popc`($peers$ & `lanemask_lt`())
- $leader \leftarrow rank = 0$
- $peers \leftarrow peers$ & `lanemask_gt`()
- ▷ Reduce features among peers
- **while** `__any_sync`($mask, peers$) **do**
  - $next \leftarrow$ `__ffs`($peers$)
  - $s' \leftarrow$ `__shuffle_sync`($mask, s, next$) ▷ Reduction step
  - **if** $next \neq 0$ **then** $s \leftarrow s + s'$
  - $peers \leftarrow peers$ & `__ballot_sync`($mask$, $rank$ **is** even)
  - $rank \leftarrow rank >> 1$
- ▷ Only the leader updates the features
- **if** $leader$ **then** `atomicAdd`($\&S[e], s$)

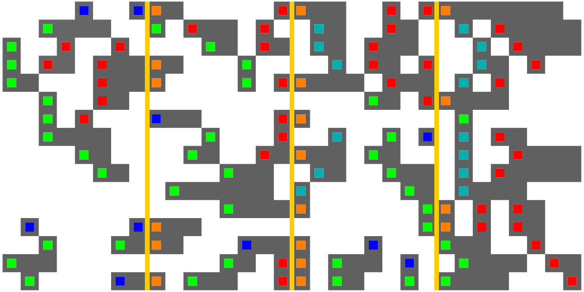


Parallel masked tree-based reduction for conflict detection during surface computation.

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: sequential algorithm

Algorithm 9: Sequential on-the-fly feature update

operator otf_merge(e_1, e_2)

- $e_1 \leftarrow \text{Find}(e_1)$
- $e_2 \leftarrow \text{Find}(e_2)$
- **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

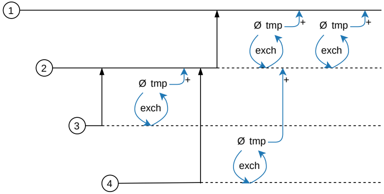
- Compute features for temporary labels and move features along the way when label unions are recorded
- Tree based reduction that follows the trees from Union-Find
- Updates are spread on all the temporary labels of a component instead being concentrated only in the final root
- More work is required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

On-the-fly Feature update: concurrent algorithm

Algorithm 10: Concurrent on-the-fly feature update

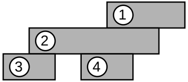
```
operator otf_merge(e1, e2)
    e1 ← Find(e1)
    e2 ← Find(e2)
    __threadfence()
    while e1 ≠ e2 do
        if e2 < e1 then swap e1, e2
        e ← atomicMin(&T[e2], e1)        ▷ label merge
        __threadfence()
        s ← atomicExch(&S[e2], 0)        ▷ feature extraction
        atomicAdd(&S[e1],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

- Enhancement of Komura/Playne equivalence to support feature moves
- Same lock-free guarantee as Playne equivalence
- 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. 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.

On-the-fly Feature update: example



Equivalences to process:

- ③ ≡ ②
- ④ ≡ ②
- ② ≡ ①

Equivalences are processed in parallel:

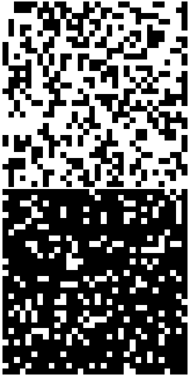
- order is non-deterministic
- example shows one possible order

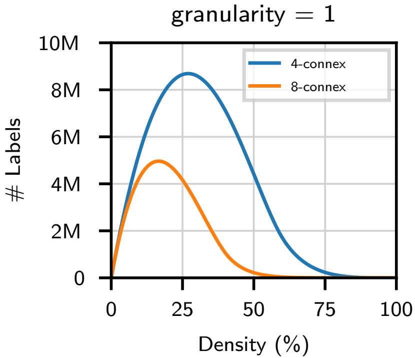
	T 1	T 2	T 3	T 4	S 1	S 2	S 3	S 4	on-going updates	
t = 0	1	2	3	4	a	b	c	d		
t = 1	1	2	2	4	a	b	c	d	6: ③≡②	
t = 2	1	2	2	4	a	b	0	d	8: ③≡② ←c	
t = 3	1	2	2	4	a	bc	0	d	9: ③≡② +c	
t = 4	1	2	2	2	a	bc	0	d	6: ④≡②	
t = 5	1	1	2	2	a	bc	0	d	6-8: ④≡②	6: ②≡①
t = 6	1	1	2	2	a	bc	0	0	8: ④≡② ←d	6-8: ②≡①
t = 7	1	1	2	2	a	0	0	0	8-9: ④≡② d	8: ②≡① ←bc
t = 8	1	1	2	2	a	d	0	0	9: ④≡② +d	8-9: ②≡① bc
t = 9	1	1	2	2	abc	d	0	0	15: ②≡① retry	9: ②≡① +bc
t = 10	1	1	2	2	abc	0	0	0	16: ②≡① ←d	
t = 11	1	1	2	2	abcd	0	0	0	17: ②≡① +d	
t = 12	1	1	2	2	abcd	0	0	0		

Only atomic write steps are shown. ←x: extract feature. +x: add feature.

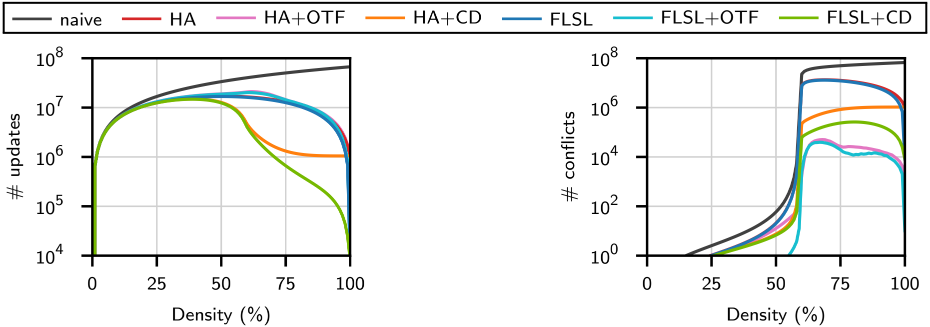
Benchmark of CCL and CCA algorithms

- 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%



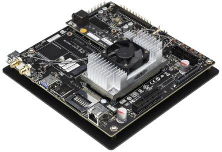


Number of conflicts: theoretical analysis



- 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

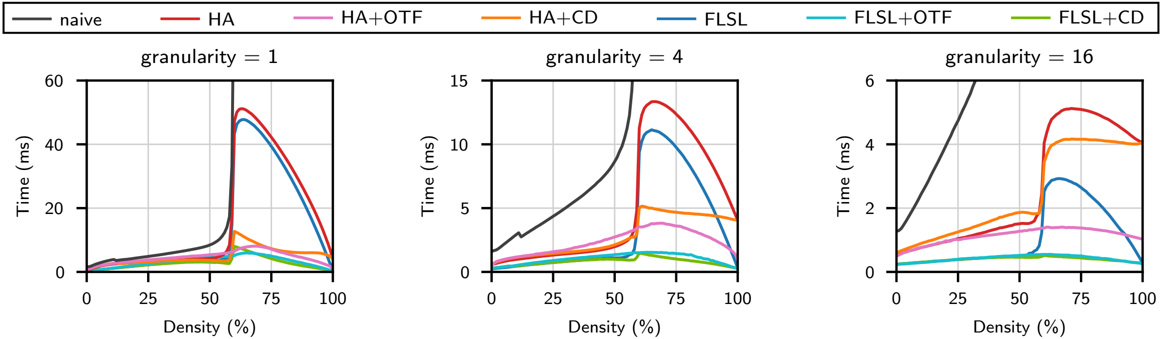
Tested machines (HPC & embedded)



A100	Tesla V100	Tegra AGX	Tegra X2	Jetson Nano
2020	2017	2018	2017	2019
Ampere (2020)	Volta (2017)	Volta (2017)	Pascal (2016)	Maxwell (2014)
6912 cores	5120 cores	512 cores	256 cores	128 cores
1.41 GHz	1.38 GHz	1.37 GHz	1.30 GHz	0.922 GHz

We focus our analysis on A100 results as it is the biggest and most recent GPU, and vote conflicts are the most problematic

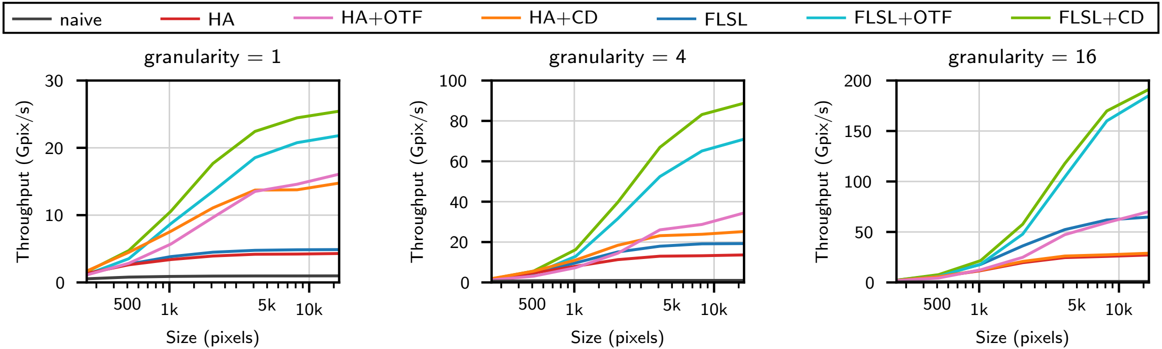
A100 Density performance



- 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.

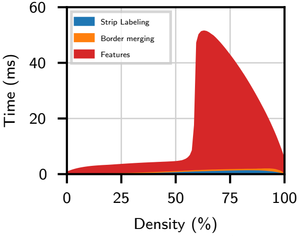
$\Rightarrow$ **FLSL+CD is the most effective combination**

A100 size performance

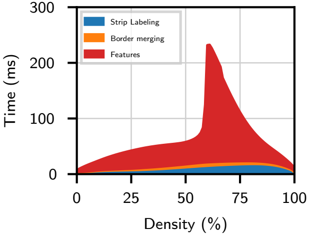


- FLSL+CD is always the best version, no matter the granularity or the size
- The ranking between versions does not depend on the image size, except for HA+OTF
  - on larger images, OTF reduces even more the number of conflicts as the effective merge tree is larger
- Small images suffer from parallelism lost
  - Not enough pixels to process in order to feed all the cores
  - Relevant only on such a big GPU (6912 CUDA cores)

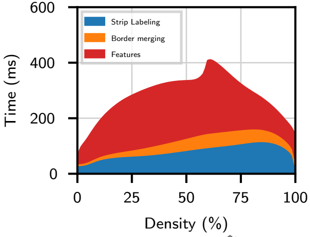
Multiple machines (HPC & embedded)



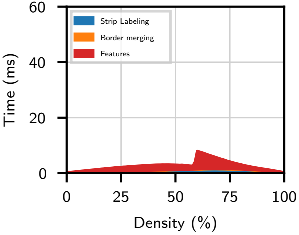
(a) HA64 (A100) 



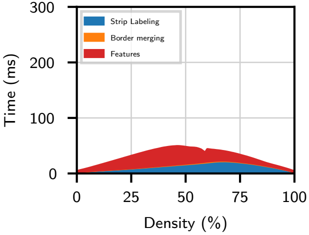
(c) HA64 (AGX) 



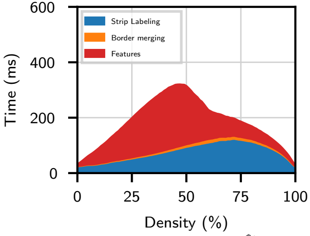
(e) HA64 (Nano) 



(b) FLSL+CD (A100) 



(d) FLSL+CD (AGX) 



(f) FLSL+CD (Nano) 

Summary

$g = 4$ ($\sim$ highy structured natural images)

Algo	A100 		V100 		AGX 		TX2 		Nano 	
naive	**1.00**	($\times$0.07)	**0.955**	($\times$0.08)	**0.438**	($\times$0.13)	**0.248**	($\times$0.23)	**0.140**	($\times$0.30)
HA	13.6	($\times$1)	12.3	($\times$1)	3.39	($\times$1)	1.08	($\times$1)	0.463	($\times$1)
HA+OTF	34.4	($\times$2.5)	22.1	($\times$1.8)	2.61	($\times$0.78)	0.914	($\times$0.85)	0.385	($\times$0.83)
HA+CD	25.2	($\times$1.9)	21.7	($\times$1.8)	3.47	($\times$1.0)	1.02	($\times$0.95)	0.405	($\times$0.87)
FLSL	19.2	($\times$1.4)	17.0	($\times$1.4)	4.95	($\times$1.5)	2.38	($\times$2.2)	1.04	($\times$2.24)
FLSL+OTF	71.0	($\times$5.2)	42.8	($\times$3.5)	5.14	($\times$1.5)	1.84	($\times$1.7)	0.871	($\times$1.88)
FLSL+CD	**88.8**	($\times$6.5)	**61.0**	($\times$5.0)	**7.14**	($\times$2.1)	**2.90**	($\times$2.7)	**1.13**	($\times$2.44)

Table 1: Average throughput (Gpix/s) for 8192$\times$8192 at $g = 4$

- For the naive version, HPC GPUs (A100 and V100) are only 2 times faster than embedded AGX
  - Naive version poorly uses the parallelism of high-end GPUs due to the extreme serialization of atomic memory accesses
- On embedded GPUs, HA+CD and HA+OTF are slower than HA
  - serialization is not as big an issue as for big GPUs
  - those variants have an overhead that are not compensated by the serialization reduction
  - this issue affects mainly HA and not FLSL because HA loses parallelism and makes them less effective
- FLSL+CD is always the most effective in average

Summary

Extreme case (for extreme low/high performance)

Algo	A100 	V100 	AGX 	TX2 	Nano 
naive	**0.337** (×0.02)	**0.325** (×0.02)	**0.310** (×0.05)	**0.151** (×0.11)	**0.098** (×0.18)
HA	16.6 (×1)	16.7 (×1)	5.92 (×1)	1.35 (×1)	0.551 (×1)
HA+OTF	78.8 (×4.7)	50.0 (×3.0)	6.69 (×1.1)	1.83 (×1.4)	0.469 (×0.85)
HA+CD	16.6 (×1.0)	16.7 (×1.0)	4.89 (×0.83)	1.23 (×0.91)	0.503 (×0.91)
FLSL	301 (×18)	191 (×12)	20.1 (×3.4)	**7.51** (×5.6)	**2.48** (×4.5)
FLSL+OTF	**320** (×19)	**197** (×12)	**21.3** (×3.6)	6.87 (×5.1)	2.45 (×4.4)
FLSL+CD	300 (×18)	192 (×12)	20.1 (×3.4)	**7.49** (×5.6)	**2.48** (×4.5)

Table 2: throughput (Gpix/s) for full images (all pixels set to 1)

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

- The naive version is as slow on A100 than on a AGX for full images
  - 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 ×900 speedup

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 on all Nvidia architectures
  - on embedded GPUs: from $\times 2$ up to $\times 5$ faster than State-of-the-Art
  - on high-end GPUs: from $\times 4$ up to $\times 20$ 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.

References I

- 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.
- 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.
- D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.
- 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.
- 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.
- E. Westphal, "`https://developer.nvidia.com/blog/voting-and-shuffling-optimize-atomic-operations/`," 2015.