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

Florian Lemaitre¹, Arthur Hennequin¹,², Lionel Lacassagne¹

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

ICASSP 2021
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 Labeling** (CCL) consists in assigning a unique number (label) to each connected component of a binary image to cluster pixels.

**Connected Components Analysis** (CCA) consists in computing some features associated to each connected component like the bounding box \([x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]\), the sum of pixels \(S\), the sums of \(x\) and \(y\) coordinates \(S_x, S_y\).

- 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):

- \( \text{find}(e, T) \) search for the root of \( e \)
- \( \text{union}(e_1, e_2, T) \) join the trees containing \( e_1 \) and \( e_2 \)
- \( \text{flatten}(T) \) 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: $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$
  - 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

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 $I[y \cdot \text{width} + x] \neq 0$ then
      $e \leftarrow E[y \cdot \text{width} + x]$
      \text{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

- 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

State-of-the-Art Feature Computation on 8192$\times$8192 random images on an A100
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.

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
I[x] & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
I[x-1] & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
(l[x]⊙l[x-1]) \cdot x & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9 \\
RLC & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9 & 0 & 3 & 4 & 6 & 8 & 9
\end{array}
\]

Algorithm 2: Kernel for FLSL segment detection

1. \(n \leftarrow 0\) ▷ Number of runs on the line \(y\)
2. \(m_p \leftarrow 0\) ▷ Previous pixel mask
3. ▷ Detect runs
4. for \(x \leftarrow \text{laneid}()\) to \(\text{width by warp}_\text{size}\) do
5. \(p \leftarrow [y \cdot \text{width} + x]\)
6. \(m_c \leftarrow \_\text{ballot.sync}(\text{ALL}, p)\)
7. ▷ Detect edges
8. \(m_e \leftarrow m_c \^ \_\text{funnelshift.l}(m_p, m_c, 1)\)
9. \(m_p \leftarrow m_e\)
10. ▷ Count edges before current index
11. \(er \leftarrow n + \_\text{popc}(m_e \& \text{lannemask.l0}())\)
12. \(ER[y \cdot \text{width} + x] \leftarrow er\)
13. ▷ “Compress store”
14. if \(m_e \& m_l\) then \(RLC[y \cdot \text{width} + er - 1] \leftarrow x\)
15. \(n \leftarrow n + \text{count_edges}(m_e)\) ▷ same \(n\) for the whole warp
16. if \(n\) is odd then
17. if \(tx = 0\) then \(RLC[y \cdot \text{width} + n] \leftarrow w\)
18. \(n \leftarrow n + 1\)
19. 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`
- 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

```c
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)
```

Labels

| 1 | 1 | 2 | 2 | 2 | 1 | 1 |

Surface: step 0

| 5 | 1 | 6 | 7 | 3 | 4 | 2 | 8 |

Surface: step 1

| 6 | 9 | 9 | 4 | 8 |

Surface: step 2

| 15 | 13 | 8 | 8 |

Surface: step 3

| 23 | 13 | 8 | 8 |

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>#updates</th>
<th>pixels generating updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>229</td>
<td><img src="image" alt="naive" /></td>
</tr>
<tr>
<td>HA</td>
<td>119</td>
<td><img src="image" alt="HA" /></td>
</tr>
<tr>
<td>FLSL</td>
<td>101</td>
<td><img src="image" alt="FLSL" /></td>
</tr>
<tr>
<td>HA+CD</td>
<td>80</td>
<td><img src="image" alt="HA+CD" /></td>
</tr>
<tr>
<td>FLSL+CD</td>
<td>48</td>
<td><img src="image" alt="FLSL+CD" /></td>
</tr>
<tr>
<td>lower-bound</td>
<td>10</td>
<td><img src="image" alt="lower-bound" /></td>
</tr>
</tbody>
</table>

F. Lemaitre – A. Hennequin – L. Lacassagne
Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm
ICASSP 2021 9/15
Algorithm 4: Concurrent on-the-fly feature update

```c
operator otf_merge(e₁, e₂)
1. e₁ ← Find(e₁)
2. e₂ ← Find(e₂)
3. __threadfence()  
while e₁ ≠ e₂ do
4. if e₂ < e₁ then swap e₁, e₂
5. e ← atomicMin(&T[e₂], e₁)  // label merge
6. __threadfence()  
7. s ← atomicExch(&S[e₂], 0)  // feature extraction
8. atomicAdd(&S[e₁], s)  // feature merge in current root
9. __threadfence()  
10. if e = e₂ then break
11. e₂ ← e  

▷ Ensure the features have reached an actual root
12. a ← Find(e₁)
13. __threadfence()  
while a ≠ e₁ do
14. s ← atomicExch(&S[e₁], 0)
15. atomicAdd(&S[a], s)  // feature merge in current root
16. __threadfence()  
17. e₁ ← a
18. a ← Find(e₁)
19. __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. 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%
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.

⇒ **FLSL+CD is the most effective combination**
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

$$\text{Table: Average CCA throughput (Gpix/s) for 8192} \times 8192 \text{ on an Nvidia A100}$$

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

* : our contributions

F. Lemaitre – A. Hennequin – L. Lacassagne
Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm
ICASSP 2021 14/15
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 CCA 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 CCL algorithm on CPU
Parallel State-of-the-art on GPU

- **Playne-Equivalence** [3](D. P. Playne, K.A. Hawick) (2018)
  - *direct CCL* 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 CCL* and *CCA* 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 CCL* for GPU (8-connexe)
  - use $2 \times 2$ blocks

- only HA tackles CCA implementation


Direct algorithms are based on Union-Find structure

**Algorithm 5: Rosenfeld labeling algorithm**

```plaintext
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
            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
    E[y][x] ← e
```

**Algorithm 6: Find(e, T)**

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

**Algorithm 7: Union(e1, e2, T)**

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

**Algorithm 8: Transitive Closure**

```plaintext
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
The direct CCL algorithms rely on Union-Find to manage equivalences. A parallel merge operation can lead to concurrency issues:

- **1st example (top-left): no concurrency**, $T[3] \leftarrow 1$, $T[4] \leftarrow 1$
- **2nd example (top-right): no concurrency**, $T[3] \leftarrow 1$, $T[4] \leftarrow 2$
  - 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($T$, $e_1$, $e_2$)**

1. while $e_1 \neq e_2$ and $e_1 \neq T[e_1]$ do
2.     $e_1 \leftarrow T[e_1] \triangleright \text{root of } e_1$
3. while $e_1 \neq e_2$ and $e_2 \neq T[e_2]$ do
4.     $e_2 \leftarrow T[e_2] \triangleright \text{root of } e_2$
5.     "Compare And Swap" loop
6.     while $e_1 \neq e_2$ do
7.         if $e_2 < e_1$ then swap $e_1$, $e_2$
8.         $e \leftarrow \text{atomicMin}(&T[e_2], e_1) \triangleright \text{Convergence is faster with } \text{atomicMin} \text{ than } \text{atomicCAS}$
9.         if $e = e_2$ then $e_2 \leftarrow e_1$
10.        else $e_2 \leftarrow e$

By definition, $e \leq T[e_2]$, 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 $e_1$
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 $32 \times 4$ 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

1. **operator** otf_merge($e_1, e_2$)
2. $e_1 \leftarrow \text{Find}(e_1)$
3. $e_2 \leftarrow \text{Find}(e_2)$
4. **if** $e_1 \neq e_2$ **then**
   5. **if** $e_2 < e_1$ **then** swap $e_1, e_2$
   6. $T[e_2] \leftarrow e_1$
   7. $s \leftarrow S[e_2]$ ▷ extract feature
   8. $S[e_2] \leftarrow 0$ ▷ reset feature
   9. $S[e_1] \leftarrow S[e_1] + s$ ▷ 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 \texttt{\_match\_any\_sync}

\begin{algorithm}
\SetAlgoLined
\DontPrintSemicolon
\textbf{operator} \texttt{match\_any\_sync}(\texttt{mask}, \texttt{v}) \; \\
\Comment{Thread must be in \texttt{mask}}
\If{not (\texttt{mask} \& \texttt{lanemask\_eq}()) }{ \textbf{return} 0 }
\textit{ballot} \leftarrow 0 \\
\textbf{do } \Comment{One iteration per distinct value}
\Comment{Remove all threads from previously find group}
\textit{mask} \leftarrow \texttt{mask} \& \neg \textit{ballot} \\
\Comment{Find the first thread among the remaining ones}
\textit{leader} \leftarrow \texttt{\_ffs}(\texttt{mask}) - 1 \\
\Comment{Broadcast the value of the leader}
\textit{ref} \leftarrow \texttt{\_shfl\_sync}(\texttt{mask}, \texttt{v}, \texttt{leader}) \\
\Comment{Mask of all threads having the same value as the leader}
\textit{ballot} \leftarrow \texttt{\_ballot\_sync}(\texttt{mask}, \texttt{v} = \texttt{ref}) \\
\While{not (\textit{ballot} \& \texttt{lanemask\_eq}()) }{} \\
\textbf{return} \textit{ballot}
\end{algorithm}

F. Lemaitre – A. Hennequin – L. Lacassagne

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

ICASSP 2021 10/12
A100 performance 4-connex

Processing time (ms/img) for 8192×8192 and Throughput (Gpix/s) on A100 (4-connex)
Processing time (ms/img) for 8192×8192 and Throughput (Gpix/s) on A100 (8-connex)