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

Florian Lemaitre\textsuperscript{1}, Arthur Hennequin\textsuperscript{1,2}, Lionel Lacassagne\textsuperscript{1}

LIP6, Sorbonne University, CNRS, France \textsuperscript{1}
LHCb experiment, CERN, Switzerland \textsuperscript{2}

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

Processing time of Connected Component Analysis on $8192 \times 8192$ 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 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 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 \( I[y][x] \neq 0 \) then
      \( e_1 \leftarrow E[y - 1][x] \)
      \( e_2 \leftarrow E[y][x - 1] \)
      if \((e_1 = e_2 = 0)\) then
        \( ne \leftarrow ne + 1 \)
        \( e \leftarrow ne \)
      else
        \( r_1 \leftarrow \text{Find}(e_1, T) \)
        \( r_2 \leftarrow \text{Find}(e_2, T) \)
        \( e \leftarrow \min^+(r_1, r_2) \)
        if \((r_1 \neq 0 \text{ and } r_1 \neq e)\) then \( T[r_1] \leftarrow e \)
        if \((r_2 \neq 0 \text{ and } r_2 \neq e)\) then \( T[r_2] \leftarrow e \)
    else
      \( e \leftarrow 0 \)
      \( E[y][x] \leftarrow e \)

**Algorithm 2: Find\((e, T)\)**

while \( T[e] \neq e \) do
  \( e \leftarrow T[e] \)
return \( e \triangleright \) the root of the tree

**Algorithm 3: Union\((e_1, e_2, T)\)**

\( r_1 \leftarrow \text{Find}(e_1, T) \)
\( r_2 \leftarrow \text{Find}(e_2, T) \)
if \((r_1 < r_2)\) then
  \( T[r_2] \leftarrow r_1 \)
elserelabel{else}
  \( T[r_1] \leftarrow r_2 \)

**Algorithm 4: Transitive Closure**

for \( i = 0 : ne \) do
  \( T[e] \leftarrow T[T[e]] \)

Parallel algorithms have to do:

- sparse addressing \(\Rightarrow\) scatter/gather SIMD instructions (AVX512/SVE)
- concurrent min computation \(\Rightarrow\) lock-free union (CUDA)
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 CCA 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 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**[4](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**[5](S. Allegretti, F. Bolelli, and C. Grana) (2019)
  - *direct* CCL for GPU (8-connexe)
  - use $2 \times 2$ blocks

❖ only HA tackles CCA implementation
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, $T[3] \leftarrow 1, T[4] \leftarrow 2$
- 3rd example (bottom-left): benign concurrency, $T[4] \leftarrow 1, T[4] \leftarrow 1$
  - 4 can’t be equal to 1 and 2
  - $\Rightarrow$ 4 has to point to 1 and 2 has to point to 1 too...
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
  
  $e_1 \leftarrow T[e_1]$ \triangleright root of $e_1$

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

\triangleright "Compare And Swap" loop

while $e_1 \neq e_2$ do

  if $e_2 < e_1$ then swap $e_1, e_2$
  
  $e \leftarrow \text{atomicMin}(& T[e_2], e_1)$ \triangleright Convergence is faster with atomicMin than atomicCAS

  if $e = e_2$ then $e_2 \leftarrow e_1$

  else $e_2 \leftarrow e$

By definition, $e \leq T[e_2]$, 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 $e_1$
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: $[x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}]$
  - 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 6: Naive Feature Computation**

```plaintext
for $y = 0 : h - 1$ do ▶ parallel
  for $x = 0 : w - 1$ do ▶ parallel
    if $I[y \cdot \text{width} + x] \neq 0$ then
      $e \leftarrow E[y \cdot \text{width} + x]$
      `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

- 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: **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**).
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

\[
\begin{align*}
    n &\leftarrow 0 \quad \triangleright \text{Number of runs on the line } y \\
    m_p &\leftarrow 0 \quad \triangleright \text{Previous pixel mask} \\
    \triangleright \text{Detect runs} \\
    \text{for } x &\leftarrow \text{laneid()} \text{ to width by warp_size do} \\
    \quad p &\leftarrow I[y \cdot \text{width} + x] \\
    \quad m_c &\leftarrow \text{ballot_sync(ALL, } p) \\
    \quad \triangleright \text{Detect edges} \\
    \quad m_e &\leftarrow m_c \oplus \text{funnelshift}_1(m_p, m_e, 1) \\
    \quad m_p &\leftarrow m_c \\
    \quad \triangleright \text{Count edges before current index} \\
    \quad er &\leftarrow n + \text{popc}(m_e \& \text{lanemask}_le()) \\
    \quad \text{ER}[y \cdot \text{width} + x] &\leftarrow er \\
    \quad \triangleright \text{"Compress store"} \\
    \quad \text{if } m_e \& m_l \text{ then } \text{RLC}[y \cdot \text{width} + er - 1] &\leftarrow x \\
    \quad n &\leftarrow n + \text{count_edges}(m_e) \quad \triangleright \text{same } n \text{ for the whole warp} \\
    \quad \text{if } n \text{ is odd then} \\
    \quad \quad \text{if } tx = 0 \text{ then } \text{RLC}[y \cdot \text{width} + n] &\leftarrow w \\
    \quad \quad n &\leftarrow n + 1 \\
    \quad \quad \text{if } tx = 0 \text{ then } N[y] &\leftarrow n
\end{align*}
\]
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

```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 1 2 2 1 1
```

Surface: step 0

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

Surface: step 1

```
6 x 9 9 x 4 x 8
```

Surface: step 2

```
15 x 13 x x x x 8
```

Surface: step 3

```
23 x 13 x x x x 1 x
```

Parallel masked tree-based reduction for conflict detection during surface computation.
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="null" alt="naive pixels" /></td>
</tr>
<tr>
<td>HA</td>
<td>119</td>
<td><img src="null" alt="HA pixels" /></td>
</tr>
<tr>
<td>FLSL</td>
<td>101</td>
<td><img src="null" alt="FLSL pixels" /></td>
</tr>
<tr>
<td>HA+CD</td>
<td>80</td>
<td><img src="null" alt="HA+CD pixels" /></td>
</tr>
<tr>
<td>FLSL+CD</td>
<td>48</td>
<td><img src="null" alt="FLSL+CD pixels" /></td>
</tr>
<tr>
<td>lower-bound</td>
<td>10</td>
<td><img src="null" alt="lower-bound pixels" /></td>
</tr>
</tbody>
</table>
On-the-fly Feature update: sequential algorithm

Algorithm 9: Sequential on-the-fly feature update

operator otf_merge(e₁, e₂)

\[
\begin{align*}
  e₁ &\leftarrow \text{Find}(e₁) \\
  e₂ &\leftarrow \text{Find}(e₂) \\
  \text{if } e₁ \neq e₂ \text{ then} & \\
  \quad \text{if } e₂ < e₁ \text{ then swap } e₁, e₂ \\
  \quad T[e₂] &\leftarrow e₁ \\
  s &\leftarrow S[e₂] \triangleright \text{extract feature} \\
  S[e₂] &\leftarrow 0 \triangleright \text{reset feature} \\
  S[e₁] &\leftarrow S[e₁] + s \triangleright \text{merge feature}
\end{align*}
\]

- 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(e_1, e_2)

1. e_1 ← Find(e_1)
2. e_2 ← Find(e_2)
3. __threadfence()
4. while e_1 ≠ e_2 do
5.   if e_2 < e_1 then swap e_1, e_2
6.   e ← atomicMin(&T[e_2], e_1) ▷ label merge
7.   __threadfence()
8.   s ← atomicExch(&S[e_2], 0) ▷ feature extraction
9.   atomicAdd(&S[e_1], s) ▷ feature merge in current root
10. __threadfence()
11. if e = e_2 then break
12. e_2 ← e

▷ Ensure the features have reached an actual root
13. a ← Find(e_1)
14. __threadfence()
15. while a ≠ e_1 do
16.   s ← atomicExch(&S[e_1], 0)
17.   atomicAdd(&S[a], s)
18.   __threadfence()
19.   e_1 ← a
20.   a ← Find(e_1)
21. __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:
• \( 3 \equiv 2 \)
• \( 4 \equiv 2 \)
• \( 2 \equiv 1 \)

Equivalences are processed in parallel:
• order is non-deterministic
• example shows one possible order

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**
  - 2020
  - Ampere (2020)
  - 6912 cores
  - 1.41 GHz

- **Tesla V100**
  - 2017
  - Volta (2017)
  - 5120 cores
  - 1.38 GHz

- **Tegra AGX**
  - 2018
  - Volta (2017)
  - 512 cores
  - 1.37 GHz

- **Tegra X2**
  - 2017
  - Pascal (2016)
  - 256 cores
  - 1.30 GHz

- **Jetson Nano**
  - 2019
  - Maxwell (2014)
  - 128 cores
  - 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.

⇒ 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)

(b) FLSL+CD (A100)

(c) HA64 (AGX)

(d) FLSL+CD (AGX)

(e) HA64 (Nano)

(f) FLSL+CD (Nano)
Summary

\(g = 4\) (\(\sim\) highly structured natural images)

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

Table 1: Average throughput (Gpix/s) for 8192×8192 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)

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

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.


