A new Direct Connected Component Labeling and Analysis Algorithm for GPUs

Arthur Hennequin$^{1,2}$, Lionel Lacassagne$^1$

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

GTC 2019 March 21$^{st}$
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.

**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 that has a global view of the image but,
- **ill-posed problem**: the computer has only a local view around a pixel (neighborhood)
- important in computer vision for pattern recognition, motion detection ...
Two classes of CCL algorithms

- **multi-pass iterative** algorithms
  - compute the local *positive* min over a $3 \times 3$ neighborhood
  - until stabilization: the number of iterations depends on the data
  - not predictable, nor suited for embedded systems

- **two-pass direct** algorithms
  - first pass = *temporary* label creation and equivalence building
  - need an equivalence table to memorize the connectivity between labels
  - then transitive closure of the tree associated to the equivalence table
  - second pass = label relabeling

- on CPU, scalar algorithms are all *direct* and can be parallelized
- on SIMD CPU, until 2019, all SIMD algorithms are *iterative*, except 1
- on GPU, until 2018, all algorithms are *iterative*, except 3

Why so few direct algorithms on GPU and SIMD ?
⇒ because *extremely complex to design* (not suited for SIMD nor GPU)
Direct algorithms are based on Union-Find structure

Algorithm 1: Rosenfeld labeling algorithm

for $i = 0 : h - 1$ do
  for $j = 0 : w - 1$ do
    if $I[i][j] \neq 0$ then
      $e_1 \leftarrow E[i - 1][j]$
      $e_2 \leftarrow E[i][j - 1]$
      if ($e_1 = e_2 = 0$) then
        $ne \leftarrow ne + 1$
        $e_x \leftarrow ne$
      else
        $r_1 \leftarrow \text{Find}(e_1, T)$
        $r_2 \leftarrow \text{Find}(e_2, T)$
        $e_x \leftarrow \text{min}^+(r_1, r_2)$
        if ($r_1 \neq 0$ and $r_1 \neq e_x$) then $T[r_1] \leftarrow e_x$
        if ($r_2 \neq 0$ and $r_2 \neq e_x$) then $T[r_2] \leftarrow e_x$
    end if
  end for
end for

$E[i][j] \leftarrow e_x$

Algorithm 2: Find($e, T$)

while $T[e] \neq e$ do
  $e \leftarrow T[e]$
return $e$  // 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
  $T[r_2] \leftarrow r_1$
else
  $T[r_1] \leftarrow r_2$

Algorithm 4: Transitive Closure

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

Parallel algorithms do:

- sparse addressing ⇒ scatter/gather SIMD instructions (AVX512/SVE)
- concurrent min computation ⇒ recursive atomic min instruction (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 $T$
- for example, there is an equivalence between 2 and 3 (stair pattern) and between 4 and 2 (concavity pattern)
- stair and concavity are the only two patterns generator of equivalence
- here, background in gray and foreground in white
Parallel State-of-the-art

- **Parallel Light Speed Labeling**[1](L. Cabaret, L. Lacassagne, D. Etiemble) (2018)
  - parallel algorithm for CPU
  - based on RLE (Run Length Encoding) to speed up processing and saves memory accesses
  - current fastest CCA algorithm on CPU

  - *direct* CCL algorithm for 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.
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):** non-problematic concurrency, \(T[4] \leftarrow 1, T[4] \leftarrow 1\)
- **4th example (bottom-right):** concurrency issue, \(T[4] \leftarrow 1, T[4] \leftarrow 2\)
  
  - 4 can’t be equal to 1 and 2
  - \(4 \Rightarrow 4\) has to point to 1 *and* 2 has to point to 1 too...
The merge function, introduced by Playne and Hawick, solves the concurrency issues by iteratively merging labels using atomic operations.

**Algorithm 5: merge(L, e₁, e₂)**

```
while e₁ ≠ e₂ and e₁ ≠ L[e₁] do  
    e₁ ← L[e₁]  // root of e₁

while e₁ ≠ e₂ and e₂ ≠ L[e₂] do  
    e₂ ← L[e₂]  // root of e₂

while e₁ ≠ e₂ do
    if e₁ < e₂ then swap(e₁, e₂)
    e₃ ← atomicMin(L[e₁], e₂)  // recursive min
    if e₃ = e₁ then e₁ ← e₂
    else e₁ ← e₃
```

By definition, e₃ ≤ L[e₁], so:
- if e₃ = e₁: no concurrent write, update of L is successful, terminates the loop
- if e₃ < e₁: concurrent write, L was updated by another thread, need to merge e₃ and e₂
Hardware Accelerated algorithm : HA4

Analysis of state-of-the-art weaknesses:
- vertical borders (non-coalescent memory accesses)
- expensive atomic operations

Analysis of state-of-the-art strengths:
- equivalence table embedded in the image (Cabaret, Playne)
- merge function (Komura [4] + Playne)
- segments labeling (Light Speed Labeling)
- necessary condition to merge two equivalence trees (Playne)

Figure 1: All possible 4 pixels configurations. Only (f) need to merge labels. (Playne)
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 segment is labeled.

- **border merging**: to merge the labels on the horizontal borders between strips.

- **relabeling / features computation**: to propagate the label of each segment to the pixels or to compute the features associated to the connected components.
The $8 \times 8$ image is divided into 2 strips of $8 \times 4$ pixels, warp size = 8

Initial strip labeling:

- only the head of each segment (start node) is labeled with an unique label
- equal to its linear address: $L[k] = k$
  with $k \Delta = y \times \text{width} + x$
- warning: label numbering starts at 0, not 1
After initialization:

- detection of merging nodes using necessary conditions in each thread
- update of start nodes only

Strips’ segments are now labeled

Here, a CC spanning over several strips is represented by 3 disjoint trees of labels
Example – Border merging (Step #2)

Same merging operations on border nodes only. All the segments are correctly labeled. A CC spanning to several strips is represented by 1 tree.

(d) Border merging

(e) Border merged
In the final step *only*, each start node (blue) **flattens** its equivalence tree

- **Label** the image: broadcast the label to the whole segment
- **Analyse** the image: accumulate features into global memory using *atomics*

Example of features associated to segment \([x_0, x_1]\) at line \(y\):

\[
S = x_1 - x_0, \quad S_y = S \times y_0, \quad S_x = \frac{1}{2} [x_1(x_1 - 1) - (x_0(x_0 - 1))]
\]
Implementation details: Grid-stride loop

- first weakness of previous GPU algorithms is the vertical border merging: the non-coalescent memory accesses are slower
- we used the grid-stride loop \[5\] design pattern to divide the image in strips instead of tiles

```
kernel Classic(width)
   x ← blockDim.x × blockIdx.x + threadIdx.x
   if x < width then
      // do stuff..

kernel Grid_stride_loop(width)
   for x ← threadIdx.x to width by blockDim.x do
      // do stuff..
```

Benefits:

- **thread reuse**: less thread creation. Helps to amortize the cost of thread creation/destruction
- **thread context is preserved**: the loop ensures that pixels are processed in a specific order and allows to reuse previously computed values
Implementation details: horizontal data exchange

All threads working on the same row are from the same warp, CUDA Warp-Level Primitives [6] can be used to directly exchange data from threads registers:

- `__ballot_sync` primitive returns a 32-bit bitmask based on the value of a boolean within each thread (1 bit per thread)

- `__shfl_sync` primitive exchanges a 32-bit value between any pair of threads in a warp. Each thread specifies a thread ID to read and a value to share
Implementation details: segments

- each thread needs to find its distance to the segment’s start node
- distance to the end is also needed for features computation
- bitwise operations can accelerate the computation of these distances (tx = thread number)

```
operator start_distance(pixels, tx)
    return _clz(~(pixels << (32-tx))) // clz = Count Leading Zeros

operator end_distance(pixels, tx)
    return _ffs(~(pixels >> (tx+1))) // ffs = Find First Set
```
Implementation details: vertical data exchange

- classic way of optimizing memory accesses: copying data from global to shared memory
- shared memory is divided in 32 banks: same bank memory accesses at different addresses get serialized [7]
Implementation details: vertical data exchange

- for each row, we store the bitmasks of the 32 neighbor pixels in different banks
- store: no serialization, load: broadcast
One final optimization...

- two pixels directly next to each other either belong to the same segment or have a different color
- we can assign a thread two pixels instead of one.
- 32-bit $\rightarrow$ 64-bit bitmask: modified distance operators.
- new version: $\text{HA4}_{64}$

```
operator start_distance64(pixels, tx)
    b ← get bit tx of ∼pixels
    txb ← tx + b
    return __clzll(~(pixels << (64−txb)))

operator end_distance64(pixels, tx)
    b ← get bit tx of ∼pixels
    txb ← tx + b
    return __ffsll(~(pixels >> (txb+1)))
```
Benchmark of CCL and CCA algorithms

- random 2048x2048 (2k) 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 = 45%
  - 4C: density = 64%
Comparison of CCL algorithms on Jetson TX2

- comparison with 2 state-of-the-art algorithms [Playne, Cabaret]
- Cabaret and Playne lose time updating all the temporary labels
- thanks to the use of segments, HA4’s processing time decreases after the percolation threshold \( d=64\% \)
- HA4\(_{64}\) is \( 2 \times \) faster in average than Playne and Cabaret
- CCL throughput: 1.2 Gpx/s (HA4\(_{64}\), 2k, \( g=4 \))
• **HA4\textsubscript{64} CCA**: labeling kernel is replaced by \textit{on-the-fly} analysis kernel
• other algorithms: features computation kernel \textit{after} relabeling kernel
• 7 features: S, Sx, Sy, x\textsubscript{min}, y\textsubscript{min}, x\textsubscript{max}, y\textsubscript{max} \rightarrow 1.1 \text{ Gpx/s} (HA4\textsubscript{64}, 2k, g=4)
Performance of CCL on Jetson AGX & V100

Latest results on Volta architecture:

- AGX: 4.6 Gpx/s (HA4\textsubscript{64}, 2k, g=4)
- V100: 27.0 Gpx/s (HA4\textsubscript{64}, 2k, g=4)
Performance of CCA on Jetson AGX & V100

Latest results on Volta architecture:

- **AGX**: 3.4 Gpx/s (HA4\textsubscript{64}, 2k, (S, Sx, Sy, x\textsubscript{min}, y\textsubscript{min}, x\textsubscript{max}, y\textsubscript{max}), g=4)
- **V100**: 14.9 Gpx/s (HA4\textsubscript{64}, 2k, (S, Sx, Sy, Sx\textsuperscript{2}, Sy\textsuperscript{2}), g=4)
Observations for Jetson AGX & V100

- **strong** scalability for CCL
- **weak** scalability for CCA (concurrent accesses in atomic operations)
- some features are faster to compute than others: the first statistical moments, computed with atomic addition, are faster than the bounding boxes computed with atomic min and max

(a) HA4_{64}(cca) V100 \((S, Sx, Sy, x_{min}, y_{min}, x_{max}, y_{max})\)

(b) HA4_{64}(cca) V100 \((S, Sx, Sy, Sx^2, Sy^2)\)
Conclusion

• two new algorithms for 4-connectivity connected component processing on GPU:
  ▶ CCL $2 \times$ faster than State-of-the-Art
  ▶ CCA new on GPU

• introduced a new way to efficiently reduce the number of global memory accesses using segments, combined with low-level intrinsics

• HA4$_{64}$ ready for realtime embedded processing.
  ▶ CCL throughput: 4.6 Gpix/s on AGX (1920x1080: 2208 fps) or
  ▶ CCA throughput: 3.4 Gpix/s on AGX (1920x1080: 1615 fps)

• future works:
  ▶ Design 8-connectivity versions on GPUs
  ▶ Improve CCA by implementing different merging strategies

• Algorithm and benchmarks published at DASIP 2018 [8]
Thank you!


