

A new Direct Connected Component Labeling and Analysis Algorithms for GPUs

Arthur Hennequin, Lionel Lacassagne, Laurent Cabaret, Quentin Meunier

▶ **To cite this version:**

Arthur Hennequin, Lionel Lacassagne, Laurent Cabaret, Quentin Meunier. A new Direct Connected Component Labeling and Analysis Algorithms for GPUs. 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP), Oct 2018, Porto, Portugal. 10.1109/dasip.2018.8596835 . hal-01923784

HAL Id: hal-01923784
https://hal.science/hal-01923784

Submitted on 15 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A new Direct Connected Component Labeling and Analysis Algorithms for GPUs

Arthur Hennequin[1,2], Lionel Lacassagne[1], Laurent Cabaret[3], Quentin Meunier[1]
[1]LIP6, Sorbonne University, CNRS, France [2]LPNHE, Sorbonne University, CNRS, France [3]MICS, CentraleSupélec, France
email: arthur.hennequin@lip6.fr, lionel.lacassagne@lip6.fr

***Abstract*—Until recent years, labeling algorithms for GPUs have been iterative. This was a major problem because the computation time depended on the content of the image. The number of iterations to reach the stability of labels propagation could be very high. In the last years, new direct labeling algorithms have been proposed. They add some extra tests to avoid memory accesses and serialization due to atomic instructions.**

This article presents two new algorithms, one for labeling (CCL) and one for analysis (CCA). These algorithms use a new data structure combined with low-level intrinsics to leverage the architecture. The connected component analysis algorithm can efficiently compute features like bounding rectangles or statistical moments. A benchmark on a Jetson TX2 shows that the labeling algorithm is from 1.8 up to 2.7 times faster than the State-of-the-Art and can reach a processing rate of 200 fps for a resolution of 2048×2048.

I. Introduction

Connected Component Labeling (CCL) was born with computer vision [20] [21] [7]. It is a central algorithm between low-level image processing (filtering) and high-level image processing (recognition, decision). CCL consists in providing a unique number to each connected components of a binary image. There are several applications in computer vision (Optical Character Recognition, motion detection, tracking) but also in High Energy Physics (tracking particles by labeling hits on detectors) or in simulated annealing.

From the beginning, CCL needed to be accelerated to run in real-time and has been ported on a wide set of parallel machines [1] [15]. After an era on single-core processors, where many sequential algorithms were developed [9] and few codes were released [2], new parallel algorithms were developed on multi-core processors [18] [6], SIMD processors [22] [13] and GPUs [12] [4] [19].

Considering a processing chain, CCL algorithms are usually followed by algorithms which compute some features like the bounding box of a component or its first moments, to compute the center of gravity. Full labeling is required for human visualization but the features computation is enough for the image analysis algorithms. This evolution of CCL algorithm is called *Connected Component Analysis* (CCA).

This paper deals with the evolution of CCL algorithm for GPU and introduces new algorithms for CCL and CCA. Section II presents the CCL algorithms for GPU, compares them, and explain the CCA implementation issue; Section III provides the CUDA terminology, Section IV introduces a new CCL and a new CCA algorithm; Section V presents the benchmarks and analyzes the results. Finally, Section VI concludes.

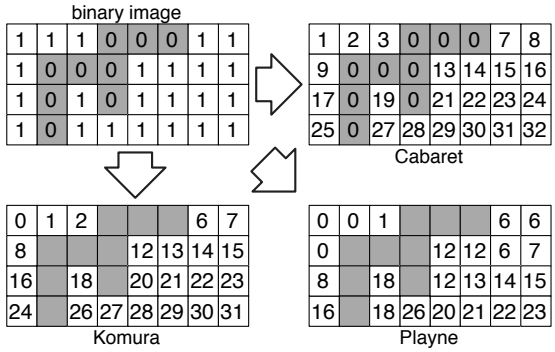


Fig. 1. Initialization on GPU for Cabaret, Komura and Playne algorithms

II. Connected Component Labeling for GPUs

The 4-connectivity Playne CCL algorithm [19] is currently the State-of-the-Art of GPU-based algorithm. For the sake of comparison, this paper uses the same notations.

A. Algorithms for CPUs

On CPUs, direct algorithms – like Rosenfeld [20] – process the binary image pixel by pixel with a neighborhood mask and an equivalence table that holds a graph structure to represent the labels connections. All direct algorithms share the same steps: 1) an initialization and a first labeling that builds the equivalence table. 2) the equivalences solving that computes the transitive closure of the graph. 3) the final labeling that updates the labels value with the equivalence table [9]. Conversely, iterative algorithms based on Haralick [7] do not use an additional equivalence table to manage the connection information. They propagate the labels step by step across the image until stabilization. The number of iterations can be very high as it is equal to the longest geodesic distance (distance in a constrained geometry) of two pixels within the shape.

B. Algorithms on GPUs

On GPUs, the first implementations were very close to Haralick algorithm [23] [5] [10]. If those algorithms are going faster and faster – due to the still-increasing number of elementary-processors with a GPU (up to 5120 CUDA cores on a TitanV) – they are still iterative and cannot match the

performance of direct algorithms on CPUs.

Designing a direct algorithm for GPU is complex due to the irregularity of Union-Find, since it contains an unbounded while-loop (within the findRoot function) and massive contention and concurrency issues (requiring atomic instructions within the Union function).

The first innovation that made direct algorithm possible on GPUs was the creation of the *label-equivalence* formula. For each pixel of coordinates (i, j), a unique label is set with the initial value $e = 1 + i \times width + j$ for Cabaret [4], or $e = i \times width + j$ for Komura [12] and Playne as they use the binary image to enable label processing. This value corresponds to the linear address in the image. Like this, the equivalence table *is* the image of labels itself: $L(i, j)$ and $T(e)$ refer to the same location (Fig. 1). Concerning Playne, the label value that is written into memory is the min over the 2×2-neighborhood (it can be viewed as an on-the-fly homogenization between variables before the memory store).

The second innovation was the introduction of the Union-Find functions which update the roots thanks to a *recursive* update based on *atomic* functions.

Direct algorithms for GPUs are very close to direct parallel algorithms for CPUs and are composed of three steps: 1) each tile (or strip) is initialized and labeled in parallel (the equivalences between labels are built during this step), 2) the borders are merged and 3) the equivalences are solved (by computing the transitive closure of the equivalence table) and the labels are relabeled with the minimum value. These steps are detailed in [4] [19] and in section IV.

Even if GPUs have a huge bandwidth, it takes time to perform many memory accesses. Same problem for *atomic* instructions: if many threads want to access the same memory address, thread serialization cannot be avoided. By performing additional tests, Playne does only the mandatory memory accesses to update the image and the equivalence table. Doing so, Playne is faster than Cabaret and Komura.

C. From labeling to analysis: features computation problem

Features computation (FC) consists in calculating for each component, some geometrical (like the bounding rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$) or statistical descriptors (like the first raw moments: S the number of pixels, S_x the sum of the x-value of labels and S_y the sum of the y-value of labels).

The straightforward implementation is to do it *after* the relabeling. Each thread performs an atomic instruction (`atomicAdd` for the moments, `atomicMin` and `atomicMax` for the bounding rectangle) on the *same* memory address, causing a catastrophic serialization leading to poor performance.

The only efficient way is to do it *during* the transitive closure to aggregate the features (belonging to the same equivalence set) into the roots [3], [11], [16]. An efficient implementation is presented in section IV.

III. CUDA terminology

Let us introduce some CUDA terms:

- kernel: The program executed by each GPU core,
- thread: A single instance of the program running on one GPU core,
- warp: A group of 32 threads executing together,
- block: A group of threads executing the same kernel,
- grid: Abstract representation of the set of blocks.

Blocks are dispatched by the GPU scheduler following a launch configuration provided by the user, consisting in a size and number of blocks, in 1D, 2D or 3D. Because we process 2D images, we choose to also express the block's size and number in 2D. Inside the kernel, CUDA provides special registers containing the thread and block coordinates. We name the following variables:

- `BLOCK_W`, `BLOCK_H`: The dimensions of a block
- bx, by: The block indexes in the grid
- tx, ty: The thread indexes in a block
- x, y: The thread indexes in the image

Each thread has its own set of registers and can access two types of memory:

- Global memory: can be accessed by all the threads and be used to communicate with the host CPU.
- Shared memory: is shared between the threads of a same block. Access latency is shorter than for global memory.

Coherency in shared and global memory can be achieved by using atomic operations provided by CUDA or by synchronizing threads of the same block with the `__syncthread` intrinsic. Threads of the same warp can also communicate without memory by directly exchanging registers with the warp-level primitive intrinsics [14].

IV. New CCL & CCA Algorithms : HA4

This section presents HA4, a new Hardware Accelerated 4-connected CCL / CCA algorithm. It is based on an hybrid pixel / segment (*run length*) approach and relies on CUDA low-level intrinsics functions to be efficient. The image is not split into tiles but into horizontal strips, each strip being processed by a unique warp. Each image segment is itself split into segments of max length the size of a warp. The low-level intrinsics let each thread efficiently tests if it is the start of a segment: only the start of segment performs memory access to manage equivalences or the features computations: the longer the segment the more it saves accesses.

The algorithm can be divided into three successive kernels that are presented in the following sections:

- Strip labeling: we independently label horizontal strips of the image.
- Border merging: we check for labels equivalences at the borders between strips.
- CCL / CCA: we perform a transitive closure of each pixel or compute some features for each label.

A. Strip labeling

The first step of the algorithm is to produce a partially labeled image. The input image I is divided into horizontal strips and each one is attributed to a block. In order to support any image width without having to increase the block size, we use the grid-stride loop design pattern [8]. Instead of assuming the block is large enough to process the entire strip, the kernel loops over the data one block size at a time. Because the same kernel processes the pixels of one strip, we can reuse past information about the continuity of the pixels, removing the need for the vertical border merging kernel. The loop also helps to amortize the threads creation and destruction by reusing them. Here, we set the block width to the number of threads in a warp, which is 32 on current hardware, and the block height to 4, as we found out that this block size provides high occupancy and good performance.

Because each warp of the block processes consecutive pixels that are on the same line, we can use some warp-level primitives to optimize computations and memory accesses. We define a segment as a consecutive set of non-zero pixels. By construction, a warp can contain up to 16 different segments. We define the start and end of the segment as its leftmost and rightmost pixels. We associate each thread of the warp to one pixel of the image. Each thread can share the value of its corresponding pixel to all the threads in the warp by using a `__ballot_sync` instruction. This instruction builds a 32-bit bitmask where the i^{th} bit is set if some predicate for the i^{th} thread of the warp is true. Here, our predicate is simply the boolean value of the thread's pixel.

Once the bitmask is known by all the threads, each thread can retrieve some information about its segment. We define two distance operators: `start_distance` and `end_distance` described in algorithm 1. These operators have two properties: for the start of the segment `start_distance` is always equal to zero, and `end_distance` is always equal to the number of pixels in the segment. For each thread, `start_distance` gives the distance to the start of the segment. Figure 2 shows an example of both operators. The `__clz` (*Count Leading Zeros*) intrinsic returns the number of consecutive zeros starting from the most significant bit and going down inside a 32-bit register. The `__ffs` (*Find First Set*) intrinsic returns the position of the first bit set to one, starting from the least significant bit and going up inside a 32-bit register.

Since CUDA 9, all warp level primitives take a mask parameter that determines which threads are participating to the operation. This allows the threads to diverge and only synchronize if it is needed. We assume the image width is a multiple of the warp size, and set the mask to `ALL` = `0xFFFFFFFF`.

For each block, the threads load their corresponding pixel from global memory, then build the bitmask and perform a segment start detection. The labels of the start pixels are initialized to their linear address $L[k_{y,x}] = k_{y,x}$. The other pixels are not initialized to reduce the amount of memory stores. For each line, we keep track of the distance to the last

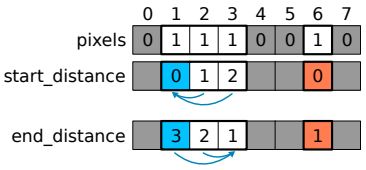


Fig. 2. Distance operators on a 8-bit bitmask. Only set pixels are considered.

Algorithm 1: Distance operators for 32-bit bitmasks

```
1 operator start_distance (pixels, tx)
2     return __clz(~(pixels << (32−tx)))
1 operator end_distance (pixels, tx)
2     return __ffs(~(pixels >> (tx+1)))
```

segment start. If the first thread of the warp has a set pixel, we check if it belongs to a longer segment and initialize it to its start address. After this first line labeling, we synchronize the threads of the block and get the bitmask of pixels from the warp above. This allows us to merge the lines within the strip. Each thread checks if its corresponding pixel in the current line or the line above is a segment start and, if it is, performs a union-find merge as described in algorithm 2. This merge function was first described by Playne and Hawick in [19] and is based on Komura's reduce function [12]. It works by finding the root of the two equivalence trees the labels are belonging to and writing the minimum root index to the root with the maximum index.

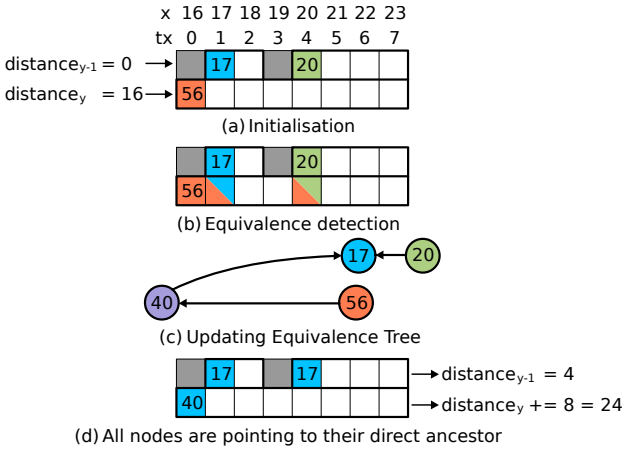


Fig. 3. Example of a block labeling (width = 40, `BLOCK_W` = 8). **(a)** shows the initialization of the start pixels to their linear address. In **(b)** each thread detects the equivalences between segments of the two lines. The equivalence of node 56 to node 40 is detected because $distance_y \neq 0$ and $56-16=40$. **(c)** shows the updated equivalence tree after the call of the merge function. Finally, **(d)** shows the final values of the start pixels and the updated values for the distances.

The strip labeling is done in global memory. Because of the few memory stores performed, going to shared memory first for the label image L, like in previous works [4] [19], would be inefficient. Instead, shared memory is used to exchange bitmasks between warps. As shared memory is organized in 32 banks, two threads willing to access different memory cells inside the same bank would result in a bank conflict, causing

Algorithm 2: merge(L, $label_1$, $label_2$)

```
while label1 ≠ label2 and label1 ≠ L[label1] do
    label1 ← L[label1]
while label1 ≠ label2 and label2 ≠ L[label2] do
    label2 ← L[label2]
while label1 ≠ label2 do
    if label1 < label2 then swap(label1, label2)
    label3 ← atomicMin(L[label1], label2)
    if label1 = label3 then label1 ← label2
    else label1 ← label3
```

access serialization. We propose to exchange the bitmask instead of the pixels. This way, only the first thread of each warp would do a memory store, in a different bank for each line, and in the next step, all threads from the same line would load from the same cell inside the same bank, resulting in a broadcast of the data. The entire strip-labeling kernel is described in algorithm 3 and an example is provided with figure 3.

Algorithm 3: HA4_Strip_Labeling(I, L, width)

```
declare shared array shared_pixels of size BLOCK_H
line_base ← y × width + tx
distance_y ← 0, distance_{y-1} ← 0
for i ← 0 to width by warp_size do
    k_{y,x} ← line_base + i
    p_{y,x} ← I[k_{y,x}]
    pixels_y ← __ballot_sync(ALL, p_{y,x})
    s_dist_y ← start_distance(pixels_y, tx)
    if p_{y,x} and s_dist_y = 0 then
        L[k_{y,x}] ← k_{y,x} (− distance_y if tx = 0)
    if tx = 0 then shared_pixels[ty] ← pixels_y
    __syncthreads()
    pixels_{y-1} ← shared_pixels[ty-1] if ty > 0 else 0
    p_{y-1,x} ← get bit tx of pixels_{y-1}
    s_dist_{y-1} ← start_distance(pixels_{y-1}, tx)
    if tx = 0 then
        s_dist_y ← distance_y
        s_dist_{y-1} ← distance_{y-1}
    if p_{y,x} and p_{y-1,x} and (s_dist_y = 0 or s_dist_{y-1} = 0) then
        label1 ← k_{y,x} − s_dist_y
        label2 ← k_{y,x} − width − s_dist_{y-1}
        merge(L, label1, label2)
    d ← start_distance(pixels_{y-1}, 32)
    distance_{y-1} ← d (+ distance_{y-1} if d = 32)
    d ← start_distance(pixels_y, 32)
    distance_y ← d (+ distance_y if d = 32)
```

B. Border Merging

Previous works suffered from the non-coalesced access of the vertical border merging. Because of the strip division, we only have to merge the horizontal border. As in the strip labeling, we perform merge operations only on the starts of the segments, limiting the number of expensive global memory accesses and atomic operations.

The border merging described in algorithm 4 produces an equivalence forest of all the segments starts inside the L array. From this forest, we can decide to finalize the labeling as described in subsection IV-C or to compute some features as described in subsection IV-D.

Algorithm 4: HA4_Strip_Merge(I, L, width)

```
if y > 0 then
    k_{y,x} ← y × width + x
    k_{y-1,x} ← k_{y,x} − width
    p_{y,x} ← I[k_{y,x}]
    p_{y-1,x} ← I[k_{y-1,x}]
    pixels_y ← __ballot_sync(ALL, p_{y,x})
    pixels_{y-1} ← __ballot_sync(ALL, p_{y-1,x})
    if p_{y,x} and p_{y-1,x} then
        s_dist_y ← start_distance(pixels_y, tx)
        s_dist_{y-1} ← start_distance(pixels_{y-1}, tx)
        if s_dist_y = 0 or s_dist_{y-1} = 0 then
            merge(L, k_{y,x} − s_dist_y, k_{y-1,x} − s_dist_{y-1})
```

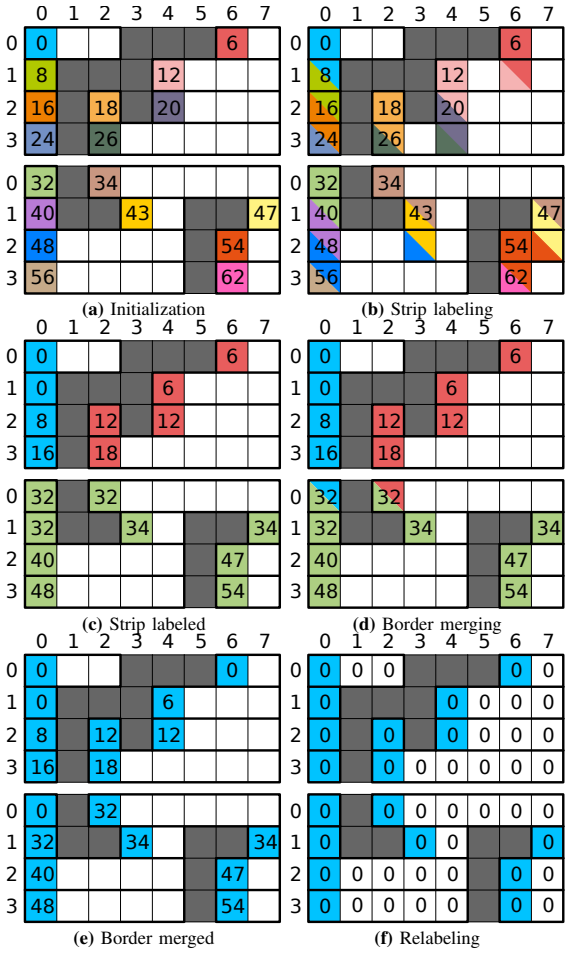


Fig. 4. Example of the HA4 algorithm on a 8×8 image divided into two strips of height 4. In **(a)**, each segment start is initialized with its linear address. In **(b)**, local equivalences are resolved for each strip. In **(d)**, we merge the equivalence trees of the two strips. Finally, in **(f)**, each segment start finds the root of its tree and shares it with the other threads of the segment for relabeling.

C. CCL - Final labeling

We implement a relabeling kernel to compare the CCL version of HA4 with previous works [4] [19].

To avoid unnecessary memory accesses, each segment delegates the task of finding the equivalence tree's root to the thread corresponding to its start. Once the start thread has found its true label, it propagates it to the others threads of the

segment using a __shfl_sync instruction. After receiving the label, each thread updates the label image L. Algorithm 5 describes this kernel. Like in previous kernels, we launch blocks of width equal to the warp size. Figure 4 shows the execution of the complete algorithm on a small image.

Algorithm 5: HA4_Relabeling(I, L, width)

```
1  k_{y,x} ← y × width + x
2  p_{y,x} ← I[k_{y,x}]
3  pixels ← __ballot_sync(ALL, p_{y,x})
4  s_dist ← start_distance(pixels, tx)
5  label ← 0
6  if p_{y,x} and s_dist = 0 then
7      label ← L[k_{y,x}]
8      while label ≠ L[label] do label ← L[label]
9  label ← __shfl_sync(ALL, label, tx − s_dist)
10 if p_{y,x} then L[k_{y,x}] ← label
```

D. CCA & Features Computation

The CCA algorithm presented in this section uses the same warp and segments idea as in previous kernels. Maximum performance is reached when it is used in combination with the strip labeling and border merging kernels presented in previous subsections, but this kernel can be used after any algorithm that produces a label equivalence image. As previously, the core idea is that only the starts of the segments search for the roots of their equivalence trees and update the features with atomic operations. With the distance operators we defined in subsection IV-A, the start can compute all the features for the segment from the pixels bitmask only. In algorithm 6, we show how to compute the most frequently used features: the number of pixels S, the sum of x coordinates S_x, the sum of y coordinates S_y and the bounding rectangle MIN_x, MIN_y, MAX_x and MAX_y. For a given segment starting at x_0 and ending at x_1, $S = x_1 - x_0 + 1$, $S_x = \phi(x_1) - \phi(x_0 - 1)$, and $S_y = y \times S$, with ϕ the sum of the first n integers: $\phi(n) = n(n+1)/2$. This algorithm is modular as we can remove the unwanted features. We can also notice that the MIN_y feature is already encoded in the label and can be retrieved as $min_y = \lfloor label/width \rfloor$.

Algorithm 6: HA4_Features(I, L, *features*, width)

```
1  k_{y,x} ← y × width + x
2  p_{y,x} ← I[k_{y,x}]
3  pixels ← __ballot_sync(ALL, p_{y,x})
4  s_dist ← start_distance(pixels, tx)
5  count ← end_distance(pixels, tx)
6  sum_x ← ((2 × x + count − 1) × count) / 2
7  sum_y ← y × count
8  max_x ← x + count − 1
9  if p_{y,x} and s_dist = 0 then
10     label ← L[k_{y,x}]
11     while label ≠ L[label] do label ← L[label]
12     atomicAdd(S[label], count)
13     atomicAdd(S_x[label], sum_x), atomicAdd(S_y[label], sum_y)
14     atomicMin(MIN_x[label], x), atomicMin(MIN_y[label], y)
15     atomicMax(MAX_x[label], max_x), atomicMax(MAX_y[label], y)
```

E. Processing two pixels per thread

At this point we successfully reduced the work done by the threads. In fact, for the worst case scenario when for every two pixel there is one white and one black pixel, only half of the threads are working. This means that in every situation, there could not be two consecutive threads in the same warp doing useful work at a time. Therefore, we can modify our kernels to process two pixels per thread.

In this new version, each warp of 32 threads is processing 64 pixels, so we need to update the horizontal thread index $tx \leftarrow tx \times 2$ and BLOCK_W $\leftarrow$ BLOCK_W $\times 2$ inside the kernels. We use the uint64_t type to store bitmasks and almost all the primitives we used for 32-bit bitmasks have a 64-bit equivalent. Each thread loads the $p_{y,x}$ and $p_{y,x+32}$ pixel. As the __ballot_sync instruction can only create 32-bit bitmasks, we have to recombine the two bitmasks into one 64-bit bitmask after the transfer.

We also have to slightly change the distance operators and the features computation to take into account which pixel of the two pixels processed by the current thread is the real root of the segment. Algorithm 7 describes the modified operators for 64-bit bitmasks.

Algorithm 7: Distance operators for 64-bit bitmasks

```
1 operator start_distance64 (pixels, tx)
2     b ← get bit tx of ∼pixels
3     txb ← tx + b
4     return __clzll(∼(pixels << (64−txb)))
1 operator end_distance64 (pixels, tx)
2     b ← get bit tx of ∼pixels
3     txb ← tx + b
4     return __ffsll(∼(pixels >> (txb+1)))
```

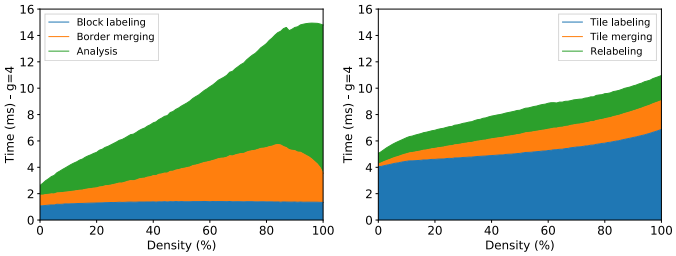
V. Experimental Evaluation

The State-of-the-Art Playne [19] and Cabaret [4] were implemented from their respective papers and compared to CCL / CCA HA4 on a embedded Jetson TX2 card. The GPU has 256 Pascal CUDA cores set to 1.3 GHz using the MAX_N performance setting. All codes are compiled with the CUDA 9.0. For reproducible results, MT19937 [17] was used to generate images of varying density ($d \in [0\% - 100\%]$) and granularity ($g \in \{1 - 16\}$) like in [4].

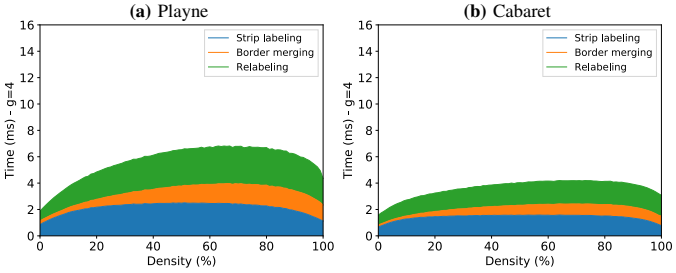
Figure 5 shows the execution time of the three steps of Playne, Cabaret and (the two versions of) HA4. We labeled each step as in the original articles. Steps with the same color perform a similar function.

Thanks to the 64-bit version of HA4, each of the three steps is faster than those of other algorithms. HA4 is - in average - 2.4× faster than Playne or Cabaret for $g = 4$. When the granularity varies from $g = 1$ (worst case for segment processing) up to $g = 16$, the speedup ratios varies from 1.8 up to 2.7.

Figure 6 shows execution time of CCA algorithms. The two first steps are identical to CCL algorithm. The third step

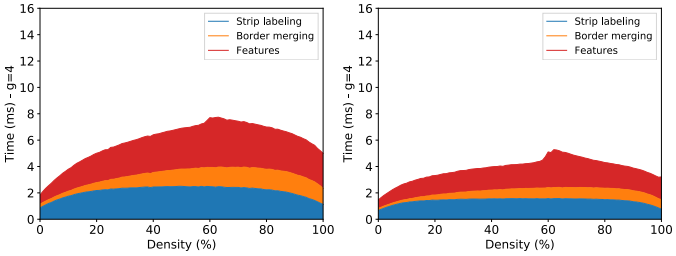


(a) Playne **(b)** Cabaret



(c) $HA4_{32}$(ccl) **(d)** $HA4_{64}$(ccl)

Fig. 5. Labeling execution time of 2048×2048 images, $g = 4$



(a) $HA4_{32}$ **(b)** $HA4_{64}$

Fig. 6. Analysis execution time for 2048×2048 images, $g = 4$

(relabeling) is replaced by the analysis kernel that performs features computation (FC). The average time of FC is 1.75 ms, which is 6.4 times faster than a naive post-FC kernel (11.2 ms). Note that the bump around $d = 64\%$ corresponds to the percolation threshold in 4-connectivity.

VI. Conclusion

This article introduced two new direct algorithms for GPUs: one for connected components labeling and one for connected component analysis. Both are based on segment/run-length processing and rely on low-level CUDA intrinsics to accelerate all steps. Thanks to these new algorithmic and hardware optimizations, our new hybrid and hardware accelerated algorithms are from 1.8 up to 2.7 times faster than the State-of-the-Art on embedded Jetson TX2 GPU.

References

[1] D. A. Bader and J. Jaja. Parallel algorithms for image histogramming and connected components with an experimental study. *Parallel and Distributed Computing*, 35,2:173–190, 1995.

[2] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana. Toward reliable experiments on the performance of connected components labeling algorithms. *Journal of Real-Time Image Processing (JRTIP)*, pages 1–16, 2018.

[3] 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*, pages 1–24, 2016.

[4] L. Cabaret, L. Lacassagne, and D. Etiemble. Distanceless label propagation: an efficient direct connected component labeling algorithm for GPUs. In *IEEE International Conference on Image Processing Theory, Tools and Applications (IPTA)*, pages 1–8, 2017.

[5] M. Ceska. Computing strongly connected components in parallel on cuda. In Nvidia, editor, *GPU Technology Conference*, 2010.

[6] S. Gupta, D. Palsetia, M. A. Patwary, A. Agrawal, and A. Choudhary. A new parallel algorithm for two-pass connected component labeling. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1355–1362. IEEE, 2014.

[7] R. Haralick. Some neighborhood operations. In *Real-Time Parallel Computing Image Analysis*, pages 11–35. Plenum Press, 1981.

[8] M. Harris. https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2013.

[9] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao. The connected-component labeling problem: a review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[10] W. W. Hwu, editor. *GPU Computing Gems*, chapter 35: Connected Component Labeling in CUDA. Morgan Kaufman, 2001.

[11] M. Klaiber, D. Bailey, and S. Simon. A single cycle parallel multi-slice connected components analysis hardware architecture. *Journal of Real-Time Image Processing*, 2016.

[12] 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*, pages 54–58, 2015.

[13] L. Lacassagne, L. Cabaret, D. Etiemble, F. Hebache, and A. Petreto. A new SIMD iterative connected component labeling algorithm. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '16, pages 1:1–1:8, New York, NY, USA, 2016. ACM.

[14] Y. Lin and V. Grover. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/, 2018.

[15] A. Lindner, A. Bieniek, and H. Burkhardt. Pisa?parallel image segmentation algorithms. pages 1–10. Springer, 1999.

[16] N. Ma, D. Bailey, and C. Johnston. Optimised single pass connected component analysis. In *International Conference on Field Programmable Technology (FPT)*, pages 185–192. IEEE, 2008.

[17] M. Matsumoto and T. Nishimura. Mersenne twister web page: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.

[18] M. Niknam, P. Thulasiraman, and S. Camorlinga. A parallel algorithm for connected component labeling of gray-scale images on homogeneous multicore architectures. *Journal of Physics - High Performance Computing Symposium (HPCS)*, 2010.

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

[20] A. Rosenfeld and J. Platz. Sequential operator in digital pictures processing. *Journal of ACM*, 13,4:471–494, 1966.

[21] F. Veillon. One pass computation of morphological and geometrical properties of objects in digital pictures. *Signal Processing*, 1,3:175–179, 1979.

[22] F. Wende and T. Steinke. Swendsen-wang multi-cluster algorithm for the 2d/3d Ising Model on Xeon Phi and GPU. In ACM, editor, *International Conference on High Performance Computing (SuperComputing)*, pages 1–12, 2013.

[23] G. Ziegler and A. Rasmusson. Efficient volume segmentation on the GPU. In Nvidia, editor, *GPU Technology Conference*, pages 1–44, 2010.