Étiquetage et analyse en composantes connexes sur GPUs

Arthur Hennequin, Lionel Lacassagne, Ian Masliah

Étiquetage et analyse en composantes connexes sur GPUs

Arthur Hennequin[1,2], Lionel Lacassagne[1], Ian Masliah[1]
[1]LIP6, Sorbonne University, CNRS, France [2]Expérience LHCb, CERN, Suisse
email: arthur.hennequin@lip6.fr, lionel.lacassagne@lip6.fr

***Abstract*—Jusqu'à récemment, les algorithmes d'étiquetage pour GPU étaient itératifs. Ce qui était un problème majeur car le temps de calcul dépendait du contenu de l'image. Le nombre d'itérations pour atteindre la stabilité de propagation des étiquettes pouvait être très élevé. Durant ces dernières années, de nouveaux algorithmes d'étiquetage directs ont été proposés. Ils ajoutent des tests additionnels pour éviter les accès mémoires et la sérialisation dû aux instructions atomiques.**

Cet article présente deux nouveaux algorithmes, un pour l'étiquetage (ECC) et un pour l'analyse (ACC) en composantes connexes. Ces algorithmes utilisent une nouvelle structure de données combinée avec des instructions de bas niveau pour tirer parti de l'architecture. Les algorithmes d'analyse en composantes connexes peuvent calculer efficacement les caractéristiques telles que les rectangles englobants ou les moments statistiques. Un benchmark sur une carte Jetson TX2 montre que l'algorithme d'étiquetage est 1.8 à 2.7 fois plus rapide que l'état de l'art et peut atteindre une vitesse de traitement de 200 ips pour une résolution de 2048×2048.

I. Introduction

L'étiquetage en composantes connexes (ECC) est né avec la vision par ordinateur [20] [21] [7]. C'est un algorithme central, situé entre le traitement d'image de bas niveau (filtrage) et le traitement d'image de haut niveau (reconnaissance, décision). L'ECC consiste à associer un nombre unique à toutes les composantes connexes d'une image binaire. Il y a plusieurs applications en vision par ordinateur (reconnaissance optique de caractères, détection de mouvement, suivi d'objets) mais aussi en physique des hautes énergies (suivi de particules en étiquetant les coups sur les détecteurs) ou bien en recuit simulé.

Depuis le début, l'ECC a besoin d'être accéléré pour se faire en temps réel et a été porté sur un large panel de machines parallèles [1] [15]. Après une ère des processeurs mono-coeur, sur lesquels beaucoup d'algorithmes séquentiels ont été développés [9] et des codes publiés [2], de nouveaux algorithmes parallèles ont été développés sur des processeurs multi-coeurs [18] [6], des processeurs SIMD [22] [13] et sur des GPU [12] [4] [19].

Si l'on regarde une chaîne de traitement, les algorithmes d'ECC sont habituellement suivis par une passe de calcul de caractéristiques comme le rectangle englobant d'une composante ou ses premiers moments statistiques, pour calculer son centre de gravité. Un étiquetage complet est requis pour une visualisation humaine mais le calcul des caractéristiques est suffisant pour les algorithmes d'analyse d'image. Cette évolution des algorithmes d'ECC est appelée *Analyse en Composantes Connexes* (ACC).

Cet article rend compte de l'évolution des algorithmes d'ECC pour GPU et introduit deux nouveaux algorithmes d'ECC et d'ACC. La section II présente les algorithme d'ECC pour GPU, les compare, et explique les problèmes des implémentions d'ACC; la section III fournit la terminologie CUDA, la section IV introduit un nouvel algorithme d'ECC et un d'ACC; la section V présente les benchmarks et analyse les résultats. Pour finir, la section VI conclut.

II. Étiquetage en composantes connexes pour GPU

L'algorithme d'ECC Playne [19] en connectivité 4 est actuellement l'état de l'art des algorithmes GPU. Pour pouvoir faire des comparaisons cet article utilisera les mêmes notations.

A. Algorithmes sur CPU

Sur CPU, les algorithmes directs (tels que Rosenfeld [20]) traitent l'image binaire pixel par pixel avec un masque de voisinage et une table d'équivalences qui contient une structure de graphe pour représenter les connections entre les étiquettes. Tous les algorithmes directs partagent les mêmes étapes : 1) une initialisation et un premier étiquetage qui construit la table d'équivalences. 2) la résolution des équivalences qui calcule la fermeture transitive du graphe. 3) l'étiquetage final qui met à jour les valeurs des étiquettes avec la table d'équivalences [9]. Inversement, les algorithmes itératifs basés sur Haralick [7] n'utilisent pas une table d'équivalences additionnelle pour gérer l'information des connexions. Ils propagent les étiquettes pixel par pixel à travers l'image jusqu'à stabilisation. Le nombre d'itérations peut être très élevé car il est égal à la plus longue distance géodésique (distance dans une géométrie contrainte) dans l'image.

B. Algorithmes sur GPU

Sur GPU, les premières implémentations étaient très proches de l'algorithme Haralick [23] [5] [10]. Si ces algorithmes vont de plus en plus vite, grâce au nombre toujours grandissant de processeurs élémenaires dans un GPU (jusqu'à 5120 coeurs CUDA sur une Titan V), ils sont toujours itératifs et ne peuvent

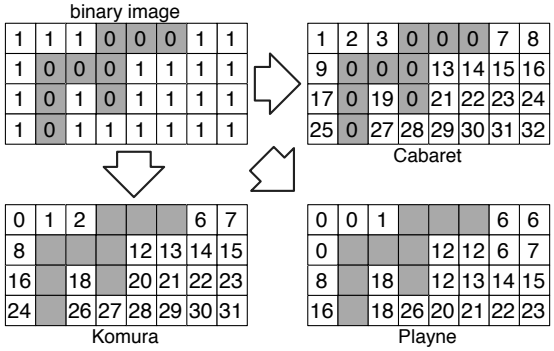


Fig. 1. Initialisation sur GPU pour les algorithmes Cabaret, Komura and Playne

pas rivaliser avec la performance d'algorithmes directs sur CPU.

Créer un algorithme direct pour GPU est complexe à cause de l'irrégularité de la structure *Union-Find*, puisqu'elle contient une boucle while sans limites connues (à l'intérieur de la fonction *findRoot*) et des problèmes de concurrence (qui demandent des instructions atomiques à l'intérieur de la fonction *Union*).

La première innovation qui a rendu possible les algorithmes directs sur GPU a été la création de la formule *label-equivalence*. Pour chaque pixel de coordonnées (i, j), une étiquette unique est mise à la valeur initiale $e = 1 + i \times width + j$ pour Cabaret [4], ou $e = i \times width + j$ pour Komura [12] et Playne car ils utilisent l'image binaire pour différencier l'arrière-plan des objets. Cette valeur correspond à l'adresse linéaire dans l'image. Ainsi, la table d'équivalences *est* l'image de labels elle même : $L(i, j)$ et $T(e)$ font référence à la même position (Fig. 1). En ce qui concerne Playne, la valeur de l'étiquette écrite en mémoire est le min sur le voisinage-2×2 (ce qui peut être vu comme une homogénéisation à la volée entre les variables avant le stockage en mémoire).

La deuxième innovation a été l'introduction des fonctions *Union-Find* qui mettent à jour les racines grâce à une mise à jour *récursive* basée sur les fonctions *atomiques*.

Les algorithmes directs sur GPU sont très proches des algorithmes parallèles sur CPU et se construisent en trois étapes : 1) chaque tuile (ou bande) est initialisée et étiquetée en parallèle (les équivalences entre les étiquettes sont construites pendant cette étape), 2) les bords sont fusionnés et 3) les équivalences sont résolues (en calculant la fermeture transitive de la table d'équivalences) et les étiquettes sont réétiquetées avec la valeur minimum. Ces étapes sont détaillées dans [4] [19] et dans la section IV.

Même si les GPU ont une large bande passante, cela prend du temps d'effectuer de nombreux accès mémoire. Même problème pour les instructions *atomiques* : si de nombreux *threads* veulent accéder à la même adresse mémoire, la sérialisation des *threads* peut être évitée. En effectuant des tests additionnels, Playne ne fait que les accès mémoire obligatoires pour mettre à jour l'image et la table d'équivalences. Ainsi, Playne est plus rapide que Cabaret ou Komura.

C. De l'étiquetage à l'analyse : problème de calcul des caractéristiques

Le calcul de caractéristiques consiste à calculer pour chaque composante, des descripteurs géométriques (comme le rectangle englobant $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$) ou statistiques (comme les premiers moments bruts : S le nombre de pixels, S_x la somme des valeurs-x des labels et S_y la somme des valeurs-y des étiquettes).

L'implémentation directe le fait *après* le réétiquetage. Chaque *thread* effectue une instruction atomique (`atomicAdd` pour les moments, `atomicMin` et `atomicMax` pour le rectangle englobant) sur la *même* adresse mémoire, causant une sérialisation catastrophique menant à une performance médiocre.

La seule manière efficace est de le faire *pendant* la fermeture transitive pour regrouper les caractéristiques (appartenant au même set d'équivalence) dans les racines [3], [11], [16]. Une implémentation efficace est présentée dans la section IV.

III. Terminologie CUDA

Introduisons quelques termes CUDA :

- *kernel*: Le programme exécuté par chaque coeur de GPU,
- *thread*: Une instance unique du programme s'exécutant sur un coeur de GPU,
- *warp*:Un groupe de 32 *threads* s'exécutant ensemble,
- *bloc*: Un groupe de *threads* exécutant le même *kernel*,
- *grille*: Représentation abstraite du set de blocs.

Les blocs sont répartis par le *scheduler* du GPU en suivant une configuration de lancement fournie par l'utilisateur, consistant en une taille et un nombre de blocs, en 1D, 2D ou 3D. Comme on traite des images 2D, nous choisissons d'exprimer aussi la taille et le nombre de blocs en 2D. Dans le *kernel*, CUDA fournit des registres spéciaux contenant le *thread* et les coordonnées du bloc. On nomme les variables suivantes :

- `BLOCK_W`, `BLOCK_H`: Les dimensions d'un bloc
- bx, by: Les indices du bloc dans la grille
- tx, ty: Les indices du *thread* dans un bloc
- x, y: Les indices du *thread* dans l'image

Chaque *thread* a sont propre set de registres et peut accéder à deux types de mémoire:

- La mémoire globale : tous les *threads* peuvent y accéder et elle peut être utilisé pour communiquer avec le CPU hôte.
- La mémoire partagée : est partagée entre les *threads* d'un même bloc. La latence d'accès est plus courte que pour la mémoire globale.

La cohérence dans les mémoires partagée et globale peut être atteinte en utilisant des opérations atomique fournies par CUDA ou en synchronisant les *threads* d'un même bloc avec l'instruction `__syncthread`. Les *threads* d'un même

warp peuvent aussi communiquer sans mémoire en échangeant directement des registres avec les instructions primitives au niveau du *warp* [14].

IV. Nouveaux algorithmes d'ECC & ACC : HA4

Cette section présente HA4, un nouvel algorithme d'ECC / ACC *Hardware Accelerated* en connectivité 4. Il est basé sur une approche hybride pixel / segment (*run length*) et se repose sur des fonctions CUDA *intrinsics* de bas niveau pour être efficace. L'image n'est pas divisée en tuile mais en bandes horizontales, chaque bande étant traitée par un *warp* unique. Chaque segment d'image est lui-même séparé en segments de taille maximale la taille d'un *warp*. Des instructions bas niveau laissent chaque *thread* tester efficacement s'il est au début d'un segment : seul le début d'un segment effectue un accès mémoire pour gérer les équivalences ou les calculs de caractéristiques : plus le segment est long plus il économise les accès.

L'algorithme peut être divisé en trois *kernels* successifs qui sont présentés dans les sections suivantes :

- L'étiquetage de bande : on étiquette indépendemment les bandes horizontales de l'image.
- La fusion des bords : on cherche des équivalences entre les étiquettes au bord des bandes
- ECC / ACC: on effectue une fermeture transitive pour chaque pixel ou on calcule des caractéristiques pour chaque étiquette.

A. Étiquetage de bandes

La première étape de l'algorithme est de fournir une image partiellement étiquetée. L'image d'entrée I est divisée en bandes horizontales et chacune est attribuée à un bloc. Pour supporter toute taille d'image sans avoir à augmenter la taille du bloc, on utilise le *design pattern grid-stride loop* [8]. Au lieu de partir du principe que le bloc est assez large pour traiter la bande entière, le *kernel* boucle sur les données une taille de bloc à la fois. Comme le même *kernel* traite les pixels d'une bande, on peut réutiliser l'information passée sur la continuité des pixels, retirant la nécessité d'un *kernel* pour la fusion verticale des bords. La boucle aide aussi à amortir la créations de threads et leur destruction en les réutilisant. Ici, on met la largeur d'un bloc au nombre de *threads* dans un *warp*, 32 pour les architectures actuelles, et la hauteur du bloc à 4, car nous avons trouvé que cette taille de bloc fournit une occupation élevée et une bonne performance.

Comme chaque *warp* du bloc traite des pixels consécutifs qui sont sur la même ligne, on peut utiliser des primitives du *warp* pour optimiser les calculs et les accès mémoire. On définit un segment comme un set de pixels non nuls consécutifs. Par construction, un *warp* peut contenir jusqu'à 16 segments différents. On définit le début et la fin du segment comme le pixel le plus à gauche et le pixel le plus à droite. On associe chaque *thread* du *warp* à un pixel de l'image. Chaque *thread* peut partager la valeur de son pixel correspondant à tous les *threads* dans le *warp* en utilisant une instruction `__ballot_sync`. Cette instruction construit un masque 32 bits où le bit i est "mis à 1" si un prédicat pour le *thread* i du *warp* est vrai. Ici, notre prédicat est simplement la valeur booléenne du pixel du *thread*.

Une fois que le *bitmask* est connu de tous les *threads*, chaque *thread* peut récupérer de l'information sur son segment. On définit deux opérateurs de distance : `start_distance` et `end_distance` décrits dans l'algorithme 1. Ces opérateurs ont deux propriétés : pour le début du segment, `start_distance` est toujours égal à zéro, et `end_distance` est toujours égal au nombre de pixels dans le segment. La figure 2 montre un exemple des deux opérateurs. L'*instrinsic* `__clz` (*Count Leading Zeros*) retourne le nombre de zéros consécutifs en commençant par le bit de poids fort et descendant dans un registre 32 bits. L'*instrinsic* `__ffs` (*Find First Set*) retourne la position du premier bit mis à 1, en commençant par le bit de poids le plus faible et en remontant dans un registre 32 bits.

Depuis CUDA 9, toutes les primitives de niveau *warp* prennent un paramètre de masque qui détermine quel *threads* participent à l'opération. Cela permet aux *threads* de diverger et de se synchroniser seulement si nécessaire. On suppose que la largeur de l'image est un multiple de la taille du *warp*, et on met le masque à `ALL = 0xFFFFFFFF`.

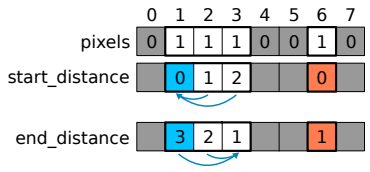


Fig. 2. Opérateurs de distance sur un masque 8 bits. Seuls les pixels mis à 1 sont pris en compte.

Algorithm 1: Opérateurs de distance pour un masque 32 bits

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

Pour chaque bloc, les *threads* chargent leur pixel correspondant depuis la mémoire globale, puis construisent les *bitmasks* et effectuent une détection de début de segment. Les étiquettes des pixels de départ sont initialisées à leur adresse linéaire $L[k_{y,x}] = k_{y,x}$. Les autres pixels ne sont pas initialisés pour réduire la quantité de stockage en mémoire. Pour chaque ligne, on garde la distance au début du dernier segment dans un registre. Si le premier *thread* du *warp* a un pixel mis à 1, on vérifie s'il appartient à un segment plus long et on l'initialise à son adresse de départ. Après ce premier étiquetage de ligne, on synchronise les *threads* du bloc et on prend le *bitmask* de pixels du *warp* du dessus. Cela nous permet de fusionner les lignes au sein de la bande. Chaque *thread* vérifie si son pixel correspondant dans la ligne courante ou la ligne du dessus est un début de segment et, si c'est le cas, effectue

une fusion *union-find* comme décrit dans l'algorithme 2. Cette fonction de fusion a été décrite pour la première fois par Playne et Hawick dans [19] et est basée sur la fonction de réduction de Komura [12]. Elle trouve les racines de deux arbres d'équivalence auxquels les étiquettes appartiennent et écrit l'index de la plus petite racine dans la plus grande.

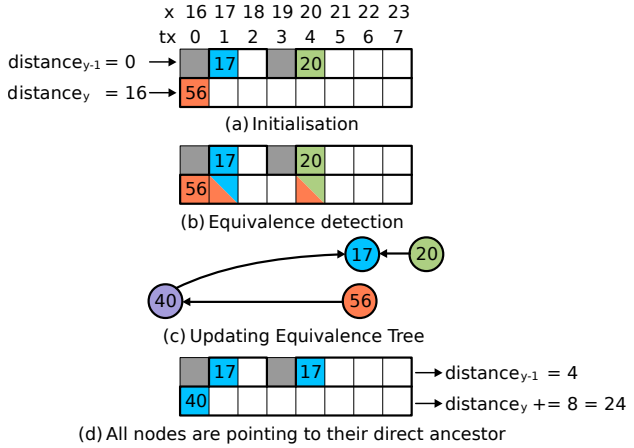


Fig. 3. Exemple d'étiquetage d'un bloc (largeur = 40, `BLOCK_W` = 8). **(a)** montre l'initialisation des pixels de départ à leur adresse linéaire. Dans **(b)** chaque thread détecte les équivalences entre les segments des deux lignes. L'équivalence du noeud 56 au noeud 40 est détectée car distance$_y \neq 0$ et 56−16=40. **(c)** montre l'arbre d'équivalences mis à jour après l'appel de la fonction de fusion. Finalement, **(d)** montre les valeurs finales des pixels de départ et les valeurs mises à jour des distances.

Algorithm 2: merge(L, label$_1$, label$_2$)

```
while label_1 ≠ label_2 and label_1 ≠ L[label_1] do
    label_1 ← L[label_1]
while label_1 ≠ label_2 and label_2 ≠ L[label_2] do
    label_2 ← L[label_2]
while label_1 ≠ label_2 do
    if label_1 < label_2 then swap(label_1, label_2)
    label_3 ← atomicMin(L[label_1], label_2)
    if label_1 = label_3 then label_1 ← label_2
    else label_1 ← label_3
```

L'étiquetage des bandes est fait en mémoire globale. Comme peu d'écritures en mémoire sont effectuées, aller à la mémoire partagée en premier pour l'image L d'étiquettes, comme dans les travaux précédents [4] [19], serait inefficace. A la place, la mémoire partagée est utilisée pour échanger des *bitmasks* entre les *warps*. Comme la mémoire partagée est organisée en 32 banques, deux *threads* voulant accéder à des cellules mémoire différentes dans la même banque résulterait en un conflit de banque, causant une sérialisation d'accès. Nous proposons d'échanger le *bitmask* à la place des pixels. De cette manière, seul le premier *thread* de chaque *warp* fait une écriture mémoire, dans des banques différentes pour chaque ligne, et dans l'étape suivante, tous les *threads* de la même ligne chargent depuis la même cellule dans la même banque, résultant en une diffusion des données. Le *kernel* entier d'étiquetage de bandes est décrit dans l'algorithme 3 et un exemple est fourni avec la figure3.

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
        label_1 ← k_{y,x} − s_dist_y
        label_2 ← k_{y,x} − width − s_dist_{y-1}
        merge(L, label_1, label_2)
    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. Fusion des bords

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})
```

Les travaux précédents souffraient de l'accès non coalescent de la fusion verticale des bords. Grâce à la division en bandes, on a seulement besoin de fusionner les bords horizontaux. Comme dans l'étiquetage des bandes, on effectue les opérations de fusion seulement sur le début des segments, limitant le nombre d'accès coûteux en mémoire globale et les opérations atomiques.

La fusion des bords décrite dans l'algorithme 4 produit une forêt d'équivalences de tous les débuts de segments dans le tableau L. A partir de cette forêt, on peut décider de finaliser l'étiquetage comme décrit dans la sous-section IV-C ou bien calculer des caractéristiques comme décrit dans la sous-section IV-D.

C. ECC - Étiquetage final

On implémente un *kernel* de réétiquetage pour comparer la version HA4 d'ECC avec les travaux précédents [4] [19]. Pour éviter les accès mémoire inutiles, chaque segment délègue la

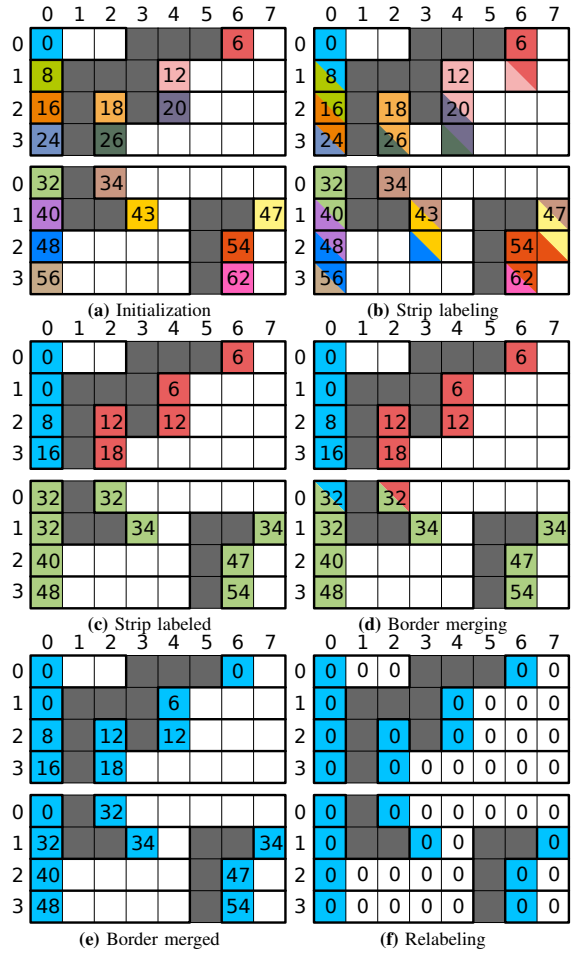


Fig. 4. Exemple d'algorithme HA4 sur une image 8×8 divisée en deux bandes de hauteur 4. Dans **(a)**, chauqe début de segment est initialisé avec son adresse linéaire. Dans **(b)**, les équivalences locales sont résolues pour chaque bande. Dans **(d)**,on fusionne les arbres d'équivalence des deux bandes. Finalement, dans **(f)**, chaque début de segment trouve la racine de son arbre et la partage avec les autres threads du segment pour le réétiquetage.

tâche de trouver la racine de l'arbre d'équivalence au *thread* correspondant à son début. Une fois que le *thread* de début a trouvé sa vraie étiquette, il la propage aux autres *threads* du segment en utilisant une instruction `__shfl_sync`. Après avoir reçu l'étiquette, chaque *thread* met à jour l'image d'étiquettes L. L'algorithme 5 décrit ce *kernel*. Comme dans les *kernels* précédents, on lance les blocs de largeur égale à la taille du *warp*. La figure 4 montre l'exécution complète de l'algorithme sur une petite 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. ACC et calcul de caractéristiques

L'algorithme d'ACC présenté dans cette section utilise la même idée de *warps* et de segments que dans les *kernels* précédents. La performance maximale est atteinte quand il est utilisé avec les *kernels* d'étiquetage de bandes et de fusion des bords, présentés dans les sous-sections précédentes, mais ce *kernel* peut être utilisé après n'importe quel algorithme qui produit une image d'équivalences d'étiquettes. Comme précédemment, l'idée centrale est que seuls les débuts des segments cherchent les racines de leurs arbres d'équivalences et mettent à jour les caractéristiques avec des opérations atomiques. Avec les opérateurs de distance que nous avons définit dans la sous-section IV-A, le début peut calculer toutes les caractéristiques pour le segment depuis le *bitmask* de pixels seulement. Dans l'algorithme 6, on montre comment calculer les caractéristiques les plus fréquemment utilisées : le nombre de pixels S, la somme S_x des coordonnées x, la somme S_y des coordonnées y et le rectangle englobant MIN_x, MIN_y, MAX_x et MAX_y. Pour un segment donnée commençant à x_0 et finissant à x_1, $S = x_1 - x_0 + 1$, $S_x = \phi(x_1) - \phi(x_0 - 1)$, et $S_y = y \times S$, avec ϕ la somme des n premiers entiers : $\phi(n) = n(n+1)/2$. Cet algorithme est modulaire car on peut ne garder que les caractéristiques souhaitées. On peut aussi noter que la caractéristique MIN_y est déjà encodée dans l'étiquette et peut être récupérée comme $min_y = \lfloor label/width \rfloor$.

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. Traiter deux pixels par thread

Jusque-là nous avons réduit avec succès le travail effectué par chaque *threads*. En fait, pour le pire scénario dans lequel on a une alternance de pixels noirs et blancs, seule la moitié des *threads* travaillent. Cela signifie que dans toute situation, il ne pourrait pas y avoir deux *threads* consécutifs dans le même *warp* effectuant du travail utile en même temps. Ainsi, on peut modifier nos *kernels* pour traiter deux pixels par *thread*.

Dans cette nouvelle version, chaque *warp* de 32 *threads* traite 64 pixels, on a donc besoin de mettre à jour l'indice horizontal du thread $tx \leftarrow tx \times 2$ et `BLOCK_W` ← `BLOCK_W` × 2 dans les *kernels*. On utilise le type `uint64_t` pour stocker les *bitmasks* et presque toutes les primitives que nous utilisons

pour des *bitmasks* 32 bits ont un équivalent 64 bits. Chaque *thread* charge les pixels $p_{y,x}$ et $p_{y,x+32}$. Comme l'instruction `__ballot_sync` peut seulement créer des *bitmasks* 32 bits, on doit recombiner les deux *bitmasks* en un seul *bitmask* 64 bits après le transfert.

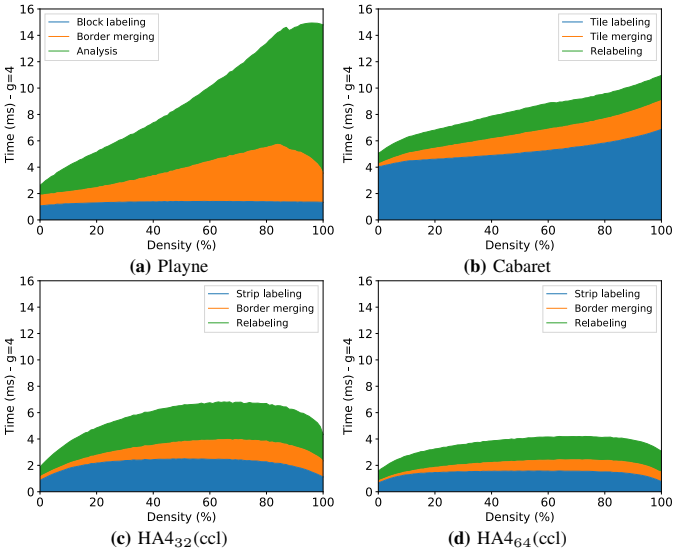
On doit aussi changer légèrement les opérateurs de distance et le calcul de caractéristiques pour prendre en compte lequel des deux pixels traités par le *thread* courant est la racine réelle du segment. L'algorithme 7 décrit les opérateurs modifiés pour des *bitmaks* 64 bits.

Algorithm 7: Opérateurs de distance pour de bitmaks 64 bits

```
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. Évaluation expérimentale

Les algorithmes de l'états de l'art Playne [19] et Cabaret [4] ont été implémentés à partir de leurs articles respectifs et ont été comparé à HA4 ECC / ACC sur une carte embarquée Jetson TX2. Le GPU a 256 coeurs CUDA Pascal mis à 1.3 GHz utilisant le réglage de performance MAX_N. Tous les codes sont compilés avec CUDA 9.0. Pour des résultats reproductibles, MT19937 [17] a été utilisé pour générer des images de densité ($d \in [0\% - 100\%]$) et de granularité ($g \in \{1 - 16\}$) variables comme dans [4].



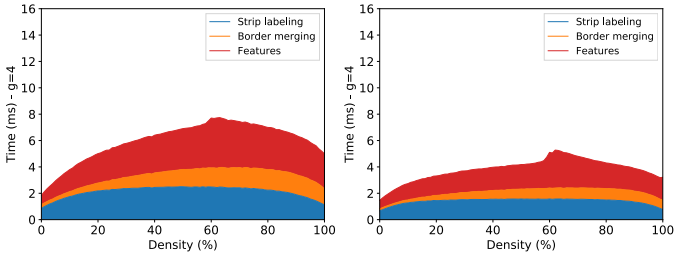
(a) Playne **(b)** Cabaret

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

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

La figure 5 montre le temps d'exécution des trois étapes de Playne, Cabaret et (des deux versions de) HA4. La légende de chaque étape reflète celle des articles originaux. Les étapes de même couleur effectuent une fonction similaire.

Grâce à la version 64 bits de HA4, chacune de ces trois étapes est plus rapide que celles des autres algorithmes. HA4 est (en moyenne) 2.4 fois plus rapide que Playne ou Cabaret pour $g = 4$. Quand la granularité varie de $g = 1$ (pire cas pour le traitement par segment) à $g = 16$, les ratios d'accélération varient de 1.8 à 2.7.



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

Fig. 6. Temps d'exécution d'analyse pour des images 2048×2048, $g = 4$

La figure 6 montre le temps d'exécution d'algorithmes d'ACC. Les deux premières étapes sont identiques aux algorithmes d'ECC. La troisième étape (le réétiquetage) est remplacée par le *kernel* d'analyse qui effectue le calcul de caractéristiques (CC). Le temps moyen du CC est 1.75 ms, ce qui est 6.4 fois plus rapide qu'un *kernel* post-CC naïf (11.2 ms). On note que la bosse autour de $d = 64\%$ correspond au seuil de percolation en connectivité 4.

VI. Conclusion

Cet article a introduit deux nouveaux algorithmes directs pour GPU : un pour l'étiquetage en composantes connexes et un pour l'analyse en composantes connexes. Les deux sont basés sur un traitement *segment/run-length* et reposent sur des instructions CUDA de bas niveau pour accélérer chaque étapes. Grâce à ces nouveaux algorithmes et à ces optimisations architecturales, ces nouveaux algorithmes hybride et *hardware accelerated* sont de 1.8 à 2.7 fois plus rapides que l'état de l'art sur un GPU embarqué Jetson TX2.

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.