Un nouvel algorithme efficace de Split & Merge pour systèmes embarqués

Nathan Maurice, Julien Sopena, Lionel Lacassagne

Compas'2021 : Parallélisme/ Architecture/ Système / Temps Réel
MILC - Lyon, France, 5-9 juillet 2021

Un nouvel algorithme efficace de *Split & Merge* pour systèmes embarqués

N. Maurice, J. Sopena, L. Lacassagne

LIP6, Sorbonne Université, CNRS
nom.prénom@lip6.fr

Résumé
Cet article présente un nouvel algorithme de segmentation d'images en régions basé sur l'approche *Split & Merge*. Les algorithmes de *Split & Merge* ont par nature, un temps d'exécution dépendant des données car le critère d'arrêt de ces deux étapes est lié à l'homogénéité de chaque région. Si les précédents algorithmes ont permis de rendre l'étape de *Split* bien moins sensible aux données, l'étape de *Merge*, bien plus complexe, reste toujours très sensible au contenu de l'image.
Une structure TTA (*Three Table Array*) est introduite dans l'étape de *Merge* pour éliminer les réallocations mémoire dues aux fusions de tableaux. Comme cette structure entraîne une perte de localité lors du parcours des tableaux, nous proposons d'ajouter deux mécanismes implémentant un cache logiciel afin d'en limiter l'impact. Une étude expérimentale menée sur un système embarqué (carte Nvidia Jetson NX) a montré que notre algorithme de *Merge* est 10.6 fois plus rapide que l'état de l'art pour des images 960×720 tout en ayant un temps d'exécution bien moins sensible aux images segmentées.

1. Introduction

La segmentation d'images est une opération classique dans le domaine du traitement d'images. Cela consiste à diviser une image en plusieurs régions correspondant à la structure perçue de l'image. Elle est une première étape dans la résolution de problèmes complexes : interprétation de scène, reconnaissance d'objets, *tracking*, etc. De nombreux travaux ont été faits dans le domaine, en particulier grâce aux progrès de l'intelligence artificielle. Malheureusement, leur temps d'exécution et leur consommation énergétique les rendent incompatibles pour une utilisation dans l'embarqué. Par exemple *Panoptic*[9] un des meilleurs algorithmes actuellement en termes de qualité, nécessite 100ms avec un GPU Nvidia V100 de plus de 200W pour segmenter et étiqueter une image 1024×2048. On est donc loin des 40 ms nécessaires à un traitement temps réel (cadence d'acquisition vidéo) et de la dizaine de Watts disponibles dans l'embarqué fortement contraint.

Pour résoudre ce problème, on peut passer par un algorithme plus léger, offrant un compromis vitesse/qualité/consommation, qui servira de pré-traitement à un algorithme de classification par réseau de neurones, en lui fournissant une image sur-segmentée. De nombreux travaux ont été menés pour développer de tels algorithmes, certains se basent sur une succession de découpages (*Split*) et de fusions (*Merge*) [8, 12, 1], d'autres utilisent les lignes de partages des eaux (*Watershed*) [3, 14, 2, 4], ou encore les arbres des formes (*Max-Tree*) [15, 10, 13]. Mais tous restent encore aujourd'hui trop lents et trop énergivores pour la cible visée. De plus, le temps de traitement de ces algorithmes dépend beaucoup de la nature des images et pas uniquement de leur taille.

Dans cet article, nous proposons de repenser l'algorithme de *Split & Merge* [1] en fonction des contraintes architecturales et systèmes pour l'embarqué. La solution proposée repose ainsi sur deux nouvelles stratégies permettant d'à la fois optimiser la localité spatiale grâce à une structure de données compacte et de réduire l'intensité des traitements réalisés par le CPU via une structure de cache logiciel.

Une évaluation mono-threadée sur le CPU d'une carte Jetson NX configurée pour consommer 15W montre :

— Un temps d'exécution de la partie *Merge quasi* indépendant du critère de fusion et indépendant des caractéristiques de l'image, pratiquement proportionnel au nombre de régions produites par

l'étape de *Split*.
— Un facteur d'accélération de $\times 4$ à $\times 10$ par rapport à l'état de l'art.
— Une segmentation de qualité équivalente.

La Section 2 présente le principe général du *Split & Merge*. La section 3 présente les nouveaux algorithmes proposés pour la phase de *Merge*. Et la section 4 présente un benchmark comparant l'*Optimal Split* à notre nouvel algorithme.

2. Segmentation par *Split & Merge*

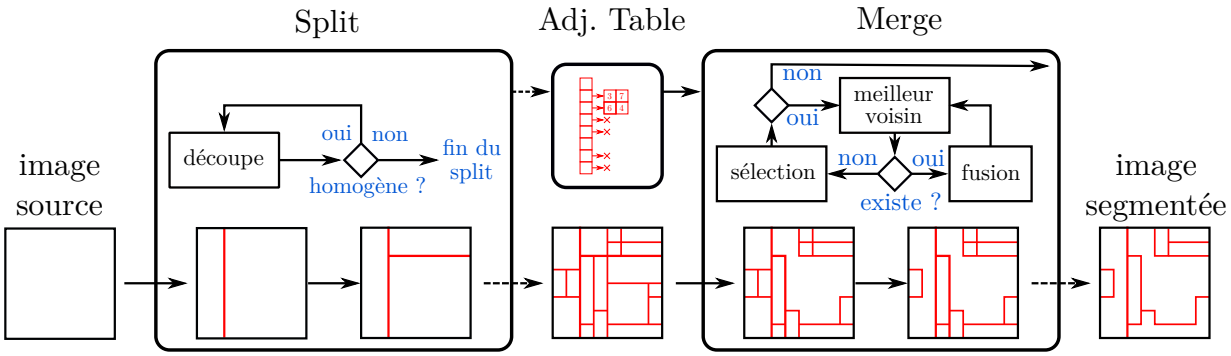


FIGURE 1 – Fonctionnement du *Split & Merge*

La segmentation par *Split & Merge* se compose de deux étapes : une étape de *Split*, puis une étape de *Merge*. La première phase découpe récursivement l'image, horizontalement ou verticalement, jusqu'à ce qu'un critère d'homogénéité soit atteint, par exemple, une variance maximum dans chacune des zones. Malheureusement, cette première étape sur-segmente l'image afin d'épouser ces formes avec des zones rectangulaires. On effectue alors une deuxième étape consistant à fusionner des zones voisines semblables. La segmentation finale permet d'approcher les différents objets de l'image par des zones rectilinéaires.

Bien que les algorithmes classiques placent les lignes de découpage au centre de la région courante, l'*Optimal Split* [12] trouve une ligne de découpage optimale pour mieux correspondre au partitionnement perçu par la vision humaine sur l'image initiale, tout en réduisant le nombre de régions requis pour atteindre ce critère. Au cours de la présente contribution, l'algorithme de *Split* utilisé est l'*Optimal Split*. Le critère d'homogénéité utilisé sera noté V_S.

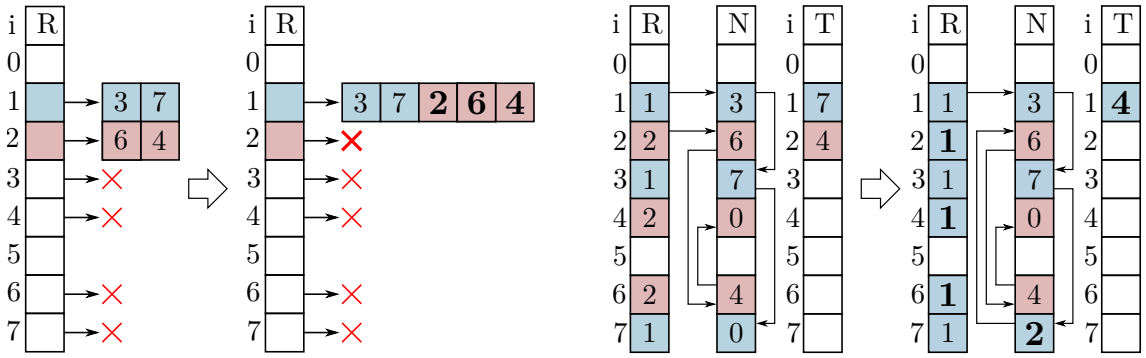
Une amélioration du *Merge* a été proposée par Aneja et al [1] permettant une réduction du temps de traitement. Le traitement du *Split & Merge* de Aneja est décrit sur la Figure 1. L'étape de *Split* lit l'image source et la découpe. Une table d'adjacence est ensuite construite. L'étape de *Merge* sélectionne successivement les régions d'intérêt. Pour chaque région sélectionnée R_S, le voisinage est parcouru pour trouver le meilleur voisin, c'est-à-dire celui dont la variance combinée avec R_S est la plus faible et inférieure à un seuil donné, noté V_M. S'il n'y a pas de voisin éligible, alors R_S est marquée invalide et une autre région est sélectionnée. Si un tel voisin existe alors il y a fusion et la recherche de meilleur voisin recommence. L'étape de *Merge* se termine lorsque toutes les régions d'intérêt ont étés traitées. La nature intrinsèquement séquentielle du *Merge* entraîne donc des variations importantes sur le partitionnement final. Son temps d'exécution est ainsi irrégulier, car fortement lié aux caractéristiques de l'image ainsi qu'à cet ordre de fusion.

3. Un nouvel algorithme de Merge

Afin d'améliorer les performances de l'algorithme de *Merge* tout en le rendant moins sensible aux caractéristiques de l'image, nous revisitons en premier lieu la fusion de régions en utilisant une structure de données (TTA) qui permet d'éviter les réallocations mémoire. Ceci se faisant au prix d'une perte de localité mémoire, nous ajoutons donc dans l'algorithme de recherche de meilleur voisin deux mécanismes permettant d'en limiter l'impact sur les performances à l'aide d'un cache.

3.1. Atténuation du coût de fusion de régions

L'algorithme de *Merge* proposé par Aneja [1] maintient une table d'adjacence qui contient une entrée pour chaque région initialement crée par le Split (comme le montre la Figure 2a). La fusion de deux régions



(a) Table d'adjacence avec tableau de tableau (b) Avec TTA

FIGURE 2 – Fusion des listes [1,3,7] et [2,6,4]

se traduit algorithmiquement par la concaténation de deux tableaux, ce qui est couteux, car cela génère des réallocations de taille croissante, et conduit à une fragmentation de la mémoire et donc à des défauts de page en raison de l'allocation paresseuse du `mmap`.

Pour réaliser ces fusions sans aucune réallocation mémoire, nous proposons d'utiliser une structure spécifique nommée TTA (*Three Table Array*). Cette dernière est une alternative à la structure Union-Find des algorithmes d'Etiquetage en Composantes Connexes [7]. Elle est composée de trois tableaux d'entiers R (*Root*), N (*Next*) et T (*Tail*) alloués définitivement au début de l'algorithme avec une taille correspondant au nombre de régions initiales. Les entiers contenus dans ces trois tableaux représentent l'ensemble des listes correspondant aux régions fusionnées (`[1, 3, 7]` et `[2, 6, 4]` dans l'exemple de la Figure 2b). Ainsi pour chaque région initiale : R contient l'identifiant de la racine de la liste auquelle elle appartient (e.g., 1 pour la région 3 car elle appartient à la liste `[1, 3, 7]`), N l'indice de l'élément qui suit la région dans la liste (e.g., 7 pour la région 3) et T contient, si la région est racine d'une liste, l'indice du dernier élément de cette liste (e.g., 7 dans la case 1 puisqu'elle est racine de `[1, 3, 7]`).

L'avantage de cette structure est de simplifier la fusion de listes comme le montre l'exemple de la figure 2b qui présente le résultat la fusion des listes `[1, 3, 7]` et `[2, 6, 4]` . Plusieurs modifications ont lieu dans les 3 tableaux : les cases 2, 6 et 4 de R prennent la valeur 1 qui devient ainsi la racine de la région fusionnée, la case 7 de N prend la valeur 2 pour réaliser la liaison entre les deux listes, la case 1 de T prend la valeur 7 qui devient la fin de la nouvelle liste. Sur cet exemple on voit bien qu'aucune réallocation n'est nécessaire, toutes les modifications se font dans les tableaux déjà alloués.

Pour confirmer les gains apportés par cette nouvelle structure, nous l'avons comparée à une implémentation basée sur un tableau de tableaux (figure 2a) en fusionnant aléatoirement des listes. Ainsi la courbe pointillée de la figure 3 présente, pour différentes tailles de tableau, le facteur d'accélération d'une implémentation utilisant un TTA par rapport à une implémentation reposant sur un tableau de tableaux et utilisant la fonction `realloc` du C pour les fusions. Sur cette figure on trouve aussi une comparaison avec une implémentation utilisant la classe `std:vector` pour réaliser les fusions. En effet, cette classe permet, grâce à une surallocation (paramètre `capacity`), d'éviter une réallocation systématique. Mais, si l'utilisation de cette classe améliore sensiblement les performances, elle reste moins efficace que l'approche TTA. On remarque toute de même une perte d'efficacité lorsque le volume de données traitées augmente. Cette réduction de l'efficacité s'explique par une perte de localité. Remarquons tout de même qu'à l'ordre de grandeur visé, 10^6, le gain du TTA est de 2.7 tandis que `std::vector` est à 0.85.

3.2. Ajout d'un cache pour la recherche du meilleur voisin

Comme nous venons de le voir, le passage au TTA pose un problème de localité, en effet, à chaque itération l'algorithme recherche le meilleur voisin sur le critère de variance, ce qui implique un parcours. Si cette recherche se fait en parcourant un tableau dans l'implémentation originale, elle peut générer de multiples sauts dans le tableau N dans le cas du TTA. Afin d'en réduire l'impact, nous proposons d'introduire d'une part un cache logiciel qui exploitera au mieux le travail fait à l'étape précédente, et d'autre part un traitement par ensemble de candidats très proches en terme de variance.

Ainsi, l'algorithme 1 parcourt le voisinage deux fois : le premier parcours trouve le meilleur voisin, le second agrège les voisins suffisamment proches dans le cache, *i.e.* les éléments dont la variance est à

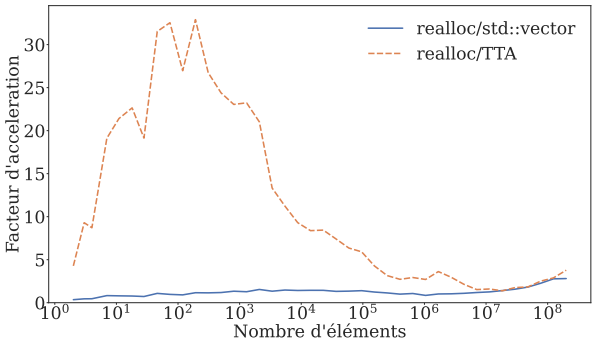


FIGURE 3 – accélération de fusions de listes pour TTA et `std::vector` par rapport à `realloc`

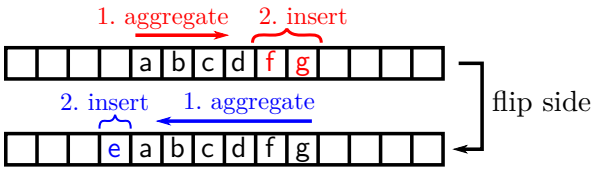


FIGURE 4 – Inversion de la direction des agrégations par flipflop

moins de ϵ du voisin retenu pour la fusion.

À la suite cette agrégation, l'algorithme 1 réalise les fusions successives sur le cache et les retire. Lors de la fusion d'une région R_k dans R_i, les voisins de R_k sont ajoutés dans le voisinage. Ils pourront ainsi être choisis par l'agrégation suivante. Les fusions continuent jusqu'à ce qu'il n'y ait plus de voisin fusionnable.

Cette approche permet de diminuer le coût de la fusion en limitant le nombre de voisins considérés, ce qui a un impact tant sur le temps de calcul, que sur la localité des données et donc l'efficacité des caches matériels.

3.3. Flip flop

Notre approche requiert tout de même deux parcours du voisinage lors de la phase d'agrégation. Nous proposons donc une amélioration permettant de se limiter à une unique passe : lors du parcours, si l'on découvre un voisin avec une variance plus proche que les précédents, on l'ajoute immédiatement dans le cache puis on l'utilise comme nouveau point de référence. Les voisins non retenus sont eux ajoutés dans le cache s'il sont suffisamment proches (à moins de ϵ) du point de référence actuel. Contrairement à la version à deux passes, certains voisins enregistrés dans le cache peuvent avoir un écart supérieur à ϵ du voisin finalement retenu pour la fusion.

Si chaque itération considérait les voisins du cache dans un ordre aléatoire, le surcoût ne serait théoriquement pas très grand. Cependant pour utiliser la localité spatiale il est préférable de parcourir le cache séquentiellement ; or notre algorithme tend à insérer les voisins par ordre décroissant, ce qui est le pire cas pour l'itération suivante.

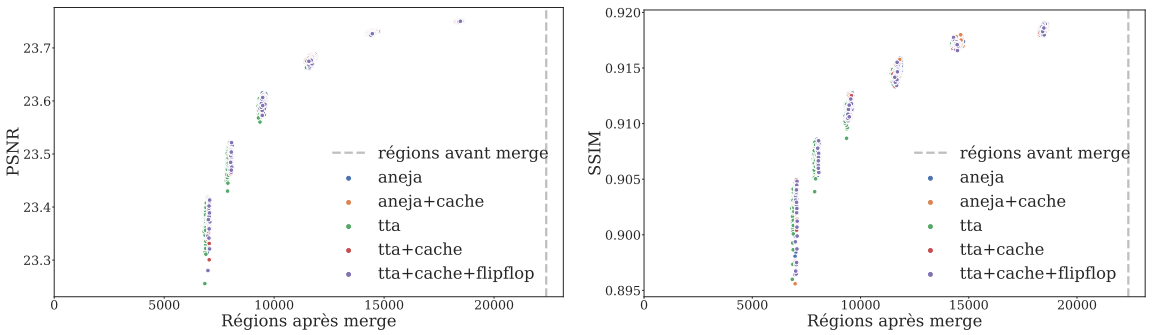
Pour bénéficier de l'accélération matérielle tout en évitant ce cas pathologique, nous avons mis au point une approche flipflop : Figure 4, dans laquelle le parcours reste séquentiel mais dont le sens change à chaque itération. On renverse alors la situation : ce qui était un problème devient un avantage puisque dans un ordre pseudo-croissant le voisin retenu a de très grandes chances d'être découvert très rapidement. La justesse d'insertion dans le cache, devient alors équivalente à celle de l'algorithme à deux passes (où l'insertion se fait en connaissant la variance de référence).

4. Benchmarks qualitatif et quantitatifs

La mise en place du cache et du cache avec stratégie de flipflop ne produisent pas la même segmentation que l'algorithme de Aneja. Pour valider ces algorithmes, vérifier la qualité de la segmentation finale est donc nécessaire. L'objectif est une phase de *Merge* efficace, produisant une segmentation proche de celle de Aneja.

La vérification se fait avec deux métriques de similarité : le *Peak Signal to Noise Ratio* (PSNR) et le *Structural Similarity Index Measure* (SSIM) [16]. Elles permettent de quantifier la distance entre deux images : deux images identiques ont un PNSR «infini» et un SSIM de 1. En revanche, plus les images diffèrent, plus le PSNR et le SSIM se rapprocheront de 0.

Dans notre cas, ces métriques permettent d'évaluer la qualité de la segmentation. On cherche ainsi à obtenir une segmentation pour laquelle les valeurs de SSIM et PSNR avec l'image source sont proches des valeurs avec Aneja, et ce, pour un même nombre de régions finales. Nous avons donc exécuté les différents algorithmes sur plusieurs images en faisant varier le critère de fusion V_M. Comme les sélections



(a) PNSR (b) SSIM

FIGURE 5 – PSNR et SSIM de la segmentation des algorithmes pour la 1ère image de CAMVID, $V_S = 15$

successives des régions sont aléatoires, le nombre de régions finales varie entre deux exécutions. Chaque point correspond ainsi à une segmentation produite par un algorithme, son abscisse et son ordonnée correspondent respectivement à son nombre de régions finales et son PNSR ou SSIM.

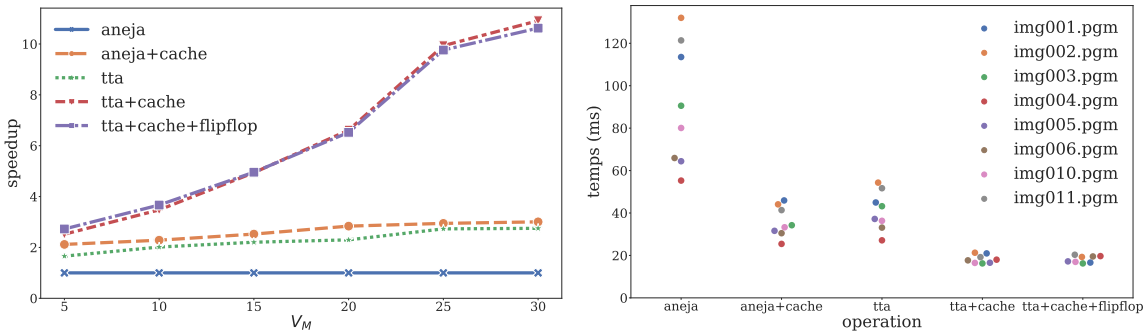
D'après la Figure 5, chaque graphique présente 6 groupes de points distincts. Chaque groupe de points correspond à une configuration et chaque point correspond à une exécution. Les valeurs de V_M sont choisies sur l'intervalle $[5; 30]$, avec un pas de 5. Le groupe le plus à gauche correspond aux exécutions pour lesquelles le plus de fusions ont étés réalisées, c'est-à-dire V_M élevé, et inversement pour la droite. Les exécutions des algorithmes testés au sein d'un même groupe sont proches les unes des autres et l'amplitude des mesures est également faible. Elle augmente lorsqu'il y a moins de régions : cela est cohérent, car plus V_M est élevé, plus le nombre de connections possibles suite aux fusions plus nombreuses sera important et donc plus la segmentation variera. Les segmentations semblent donc bien produire le résultat espéré, ce qui est confirmé par une vérification visuelle.

Comme les algorithmes de *Merge* proposés produisent des résultats de qualité proche, il n'est possible de comparer leur temps d'exécution qu'à qualités égales. Afin de mesurer les performances entre les algorithmes, le CPU de la carte Jetson NX a été utilisé, car à la fois très performant (Cortex ARM custom proche du A76) et très *customizable* (plusieurs profils types avec 2, 4 ou 6 coeurs et à différentes fréquences). Un profil avec 2 coeurs actifs pour une enveloppe thermique de 15 Watt a été utilisé. L'ensemble des algorithmes sont mono-threadés.
Les exécutions sont fixées sur un coeur et sont précédées d'un tour de *warm up* pour réduire les biais au début de l'exécution. Les nombres aléatoires sont générés avec Mersenne Twister 19337 [11].

Les temps d'exécution ont été mesurés sur différentes images, pour des valeurs de paramètres V_S et V_M allant de 5 à 30. Cet intervalle permet des segmentations fines. Les segmentations produites sur l'image *cameraman* avec cet intervalle sont disponibles en Appendix 9. Pour ces benchmarks, nous utilisons une base de 11 images urbaines de 960×720 tirées de la séquence vidéo CAMVID [5, 6]. Sur chaque image, 5 algorithmes différents sont testés : l'algorithme de Aneja (*aneja*), la version avec TTA (*tta*) seul, une version TTA avec cache logiciel (*tta+cache*), le TTA avec flipflop (*tta+cache+flipflop*) et Aneja avec cache (*aneja+cache*). Pour chaque configuration, 50 exécutions sont réalisées. La taille du cache logiciel choisie est 32 et $\epsilon = 500$.

La structure TTA permet une accélération des performances allant de 1.7 à 2.8 (Fig. 6a). L'ajout d'un cache sur Aneja permet également une amélioration, de 2.1 à 3. Cependant, la combinaison des deux stratégies permet une accélération considérable, de 2.5 à 10.9 pour tta+cache et de 2.7 à 10.6 pour tta+cache+flipflop. Nos nouveaux algorithmes passent ainsi mieux à l'échelle. Le temps d'exécution de l'approche par flipflop reste proche de tta+cache. La gestion du mécanisme de flipflop a vraisemblablement un surcoût important par rapport au gain de temps sur le double parcours du tableau de voisinage.

La Figure 6b montre une autre propriété intéressante : non seulement les temps d'exécution des algorithmes tta+cache et tta+cache+flipflop sur l'ensemble des images du benchmark sont plus courts, mais ils présentent aussi une très faible variation entre les différentes images, ce qui n'est pas le cas de l'original qui présente une variation allant du simple au double. On retrouve cette régularité sur la Figure 7 qui étudie les écarts types relatifs des algorithmes et montre comment l'ajout du mécanisme flipflop



(a) Facteur d'accélération par rapport à Aneja sur CAMVID, V_S=15

(b) Temps d'exécution normalisé de l'étape de *Merge* pour CAMVID, avec V_S=15, V_M=15

FIGURE 6 – Temps d'exécution pour CAMVID, sur Jetson NX

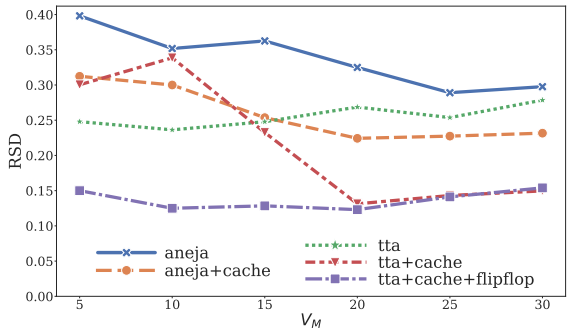


FIGURE 7 – Écarts types relatifs des algorithmes sur CAMVID, $V_S = 15$

permet d'obtenir cette bonne propriété quel que soit le paramètre de variance. On note également que celui de tta+cache+flipflop reste stable, à 0.15 alors que celui de tta+cache varie entre 0.35 et 0.15. Cela signifie donc que tta+cache+flipflop est davantage *data-independent* que le reste, le contenu de l'image influant moins sur le temps d'exécution.

La Figure 10 montre que l'étape de *Merge* a un temps très stable autour de 20 ms pour les images CAMVID et autour de 5 ms pour les images classiques 512 × 512. Il est donc possible de pipeliner de manière équilibrée l'algorithme de *Split & Merge* sur deux coeurs : l'étape de *Split* sur un premier coeur et l'ensemble des autres étapes (labeling, calcul des adjacences et *Merge* sur le second et de respecter la cadence vidéo "temps réel" de 25 images par seconde (soit 40 ms par image).

5. Conclusion

Dans cet article, nous avons proposé un nouvel algorithme de *Merge* qui améliore significativement les performances de la segmentation d'image par *Split & Merge*.

La fusion de listes qui est le cœur des algorithmes de *Merge* se fait maintenant sans réallocation ni mouvement mémoire grâce à l'utilisation d'une structure de données TTA. Nous avons également présenté une stratégie de cache logiciel à laquelle s'ajoute un parcours flipflop pour rendre les recherches itératives de meilleur voisin les plus rapides possible en compensant la perte de localité.

Outre le gain de performance, les deux mécanismes proposés rendent le traitement de l'image très peu sensible aux caractéristiques intrinsèques de l'image ; ce qui les rend très adaptés à une utilisation dans des systèmes embarqués où la régularité des temps de traitement est un point important pour respecter une cadence de traitement vidéo (typiquement 25 images / seconde soit 40 ms).

Enfin, l'amélioration de performance sur l'étape de *Merge* la rapproche de la durée de l'étape de *Split*. Il est donc maintenant possible de pipeliner de manière équilibrée l'algorithme complet sur deux cœurs, ce qui n'était pas faisable auparavant. Ainsi les algorithmes de *Split & Merge* pourraient s'exécuter en temps réel (cadence vidéo) sur des systèmes embarqués et pour des images de grandes tailles.

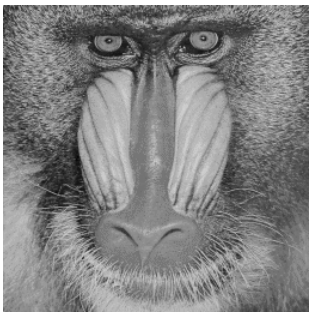
Bibliographie

1. Aneja (K.), Laguzet (F.), Lacassagne (L.) et Merigot (A.). – Video-rate image segmentation by means of region splitting and merging. – In *2009 IEEE International Conference on Signal and Image Processing Applications*, pp. 437–442, Kuala Lumpur, Malaysia, 2009. IEEE.
2. Audigier (R.) et Lotufo (R.). – Uniquely-Determined Thinning of the Tie-Zone Watershed Based on Label Frequency. *Journal of Mathematical Imaging and Vision*, vol. 27, n2, février 2007, pp. 157–173.
3. Beucher (S.) et Meyer (F.). – *The Morphological Approach to Segmentation : The Watershed Transformation*, pp. 433–481. – CRC Press, 1993, first édition.
4. Braham (Y.), Elloumi (Y.), Akil (M.) et Bedoui (M. H.). – Parallel computation of Watershed Transform in weighted graphs on shared memory machines. *Journal of Real-Time Image Processing*, vol. 17, n3, 2018, pp. 527–542.
5. Brostow (G. J.), Fauqueur (J.) et Cipolla (R.). – Semantic object classes in video : A high-definition ground truth database. *Pattern Recognition Letters*, vol. 30, n2, janvier 2009, pp. 88–97.
6. Brostow (G. J.), Shotton (J.), Fauqueur (J.) et Cipolla (R.). – Segmentation and Recognition Using Structure from Motion Point Clouds. *In : Computer Vision – ECCV 2008*, éd. par Forsyth (D.), Torr (P.) et Zisserman (A.), pp. 44–57. – Berlin, Heidelberg, Springer Berlin Heidelberg, 2008. Series Title : Lecture Notes in Computer Science.
7. He (L.), Chao (Y.) et Suzuki (K.). – A Linear-Time Two-Scan Labeling Algorithm. – In *2007 IEEE International Conference on Image Processing*, pp. V – 241–V – 244, San Antonio, TX, USA, 2007. IEEE.
8. Horowitz (S. L.) et Pavlidis (T.). – Picture Segmentation by a Tree Traversal Algorithm. *Journal of the ACM (JACM)*, vol. 23, n2, avril 1976, pp. 368–388.
9. Hou (R.), Li (J.), Bhargava (A.), Raventos (A.), Guizilini (V.), Fang (C.), Lynch (J.) et Gaidon (A.). – Real-Time Panoptic Segmentation from Dense Detections. – In *Conference on Computer Vision and Pattern Recognition*, avril 2020.
10. Jones (R.). – Connected Filtering and Segmentation Using Component Trees. *Computer Vision and Image Understanding*, vol. 75, n3, septembre 1999, pp. 215–228.
11. Matsumoto (M.) et Nishimura (T.). – Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, vol. 8, n1, janvier 1998, p. 26.
12. Merigot (A.). – Revisiting image splitting. – In *12th International Conference on Image Analysis and Processing, 2003.Proceedings.*, pp. 314–319, Mantova, Italy, 2003. IEEE Comput. Soc.
13. Passat (N.), Naegel (B.), Rousseau (F.), Koob (M.) et Dietemann (J.-L.). – Interactive segmentation based on component-trees. *Pattern Recognition*, vol. 44, n10-11, octobre 2011, pp. 2539–2554.
14. Roerdink (J. B.) et Meijster (A.). – The Watershed Transform : Definitions, Algorithms and Parallelization Strategies. *Fundamenta Informaticae*, vol. 41, n1,2, 2000, pp. 187–228.
15. Salembier (P.), Oliveras (A.) et Garrido (L.). – Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, vol. 7, n4, avril 1998, pp. 555–570.
16. Wang (Z.), Bovik (A.), Sheikh (H.) et Simoncelli (E.). – Image Quality Assessment : From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, vol. 13, n4, avril 2004, pp. 600–612.

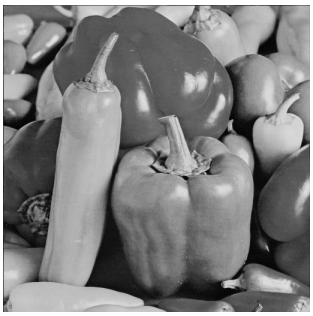
Annexes

A.1 - Images

(a) cameraman - 256 × 256



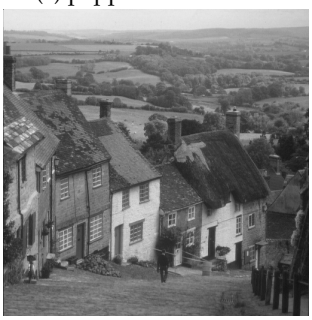
(b) mandrill - 256 × 256



(c) peppers - 256 × 256

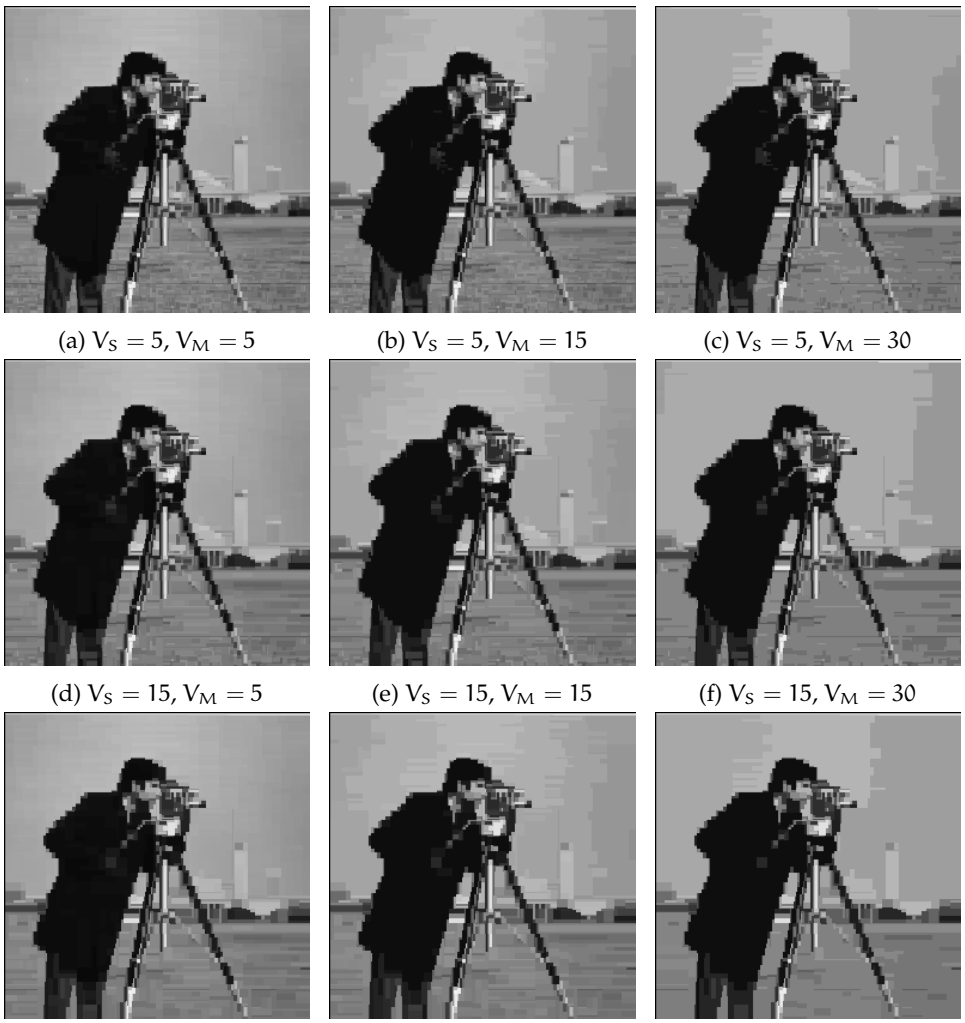
(d) barbara - 512 × 512

(e) boat - 512 × 512



(f) goldhill - 512 ×512

FIGURE 8 – Images classiques



(a) $V_S = 5$, $V_M = 5$ (b) $V_S = 5$, $V_M = 15$ (c) $V_S = 5$, $V_M = 30$

(d) $V_S = 15$, $V_M = 5$ (e) $V_S = 15$, $V_M = 15$ (f) $V_S = 15$, $V_M = 30$

(g) $V_S = 30$, $V_M = 5$ (h) $V_S = 30$, $V_M = 15$ (i) $V_S = 30$, $V_M = 30$

FIGURE 9 – *cameraman* segmenté avec différents critères de *Split & Merge*

A.2 - Pseudo-codes

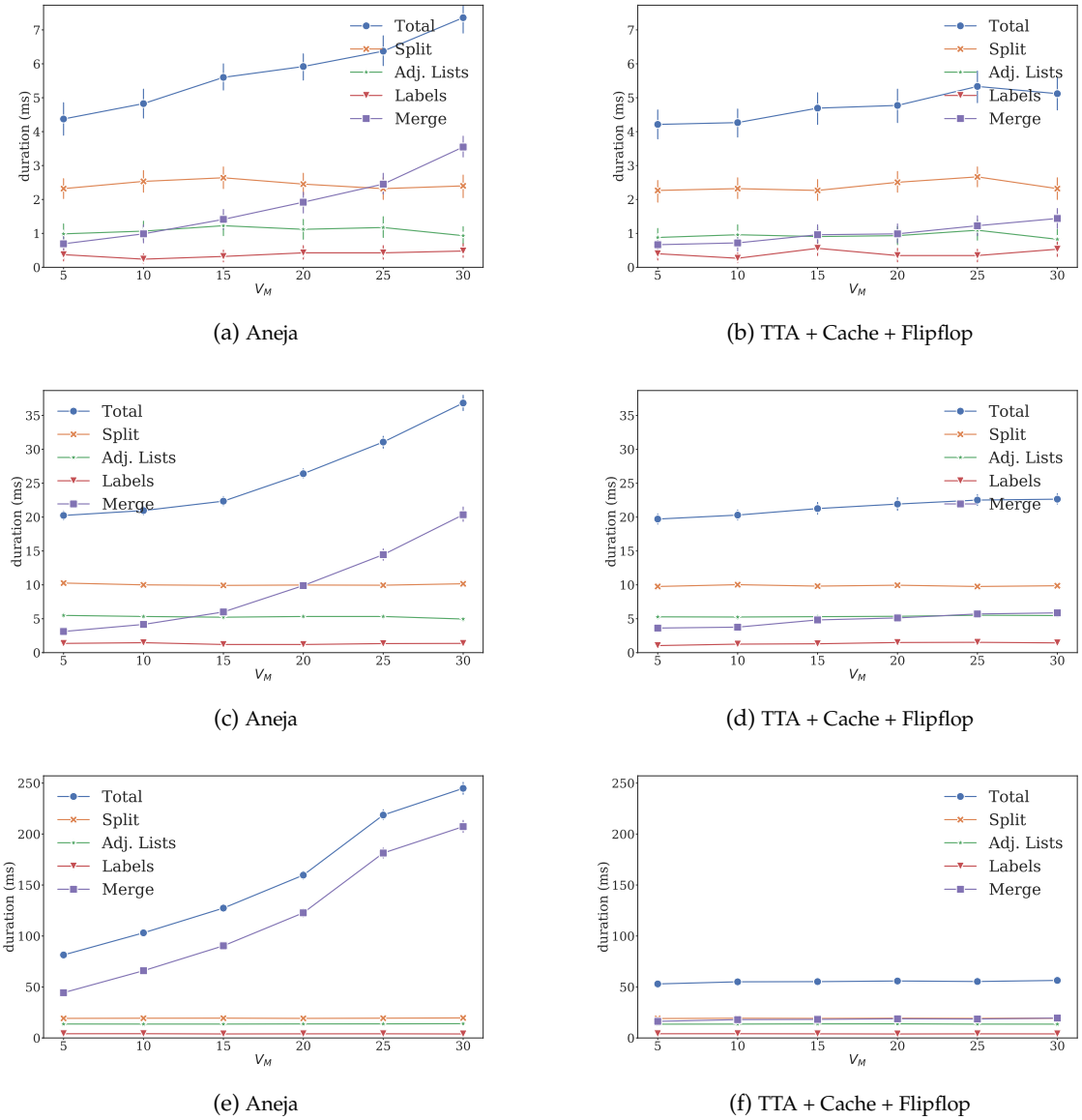
```
1  Function Merge(nodeTree, adjTable[][])
2      validRegions = CountValidRegions(nodeTree) ;
3      tta ← create tta of size n_nodes ;
4      bitmap = ← create bitmap of size n_nodes ;
5      while can still select valid region do
6          N_i ← selectRegion(nodeTree) ;
7          BitmapClear(bitmap, N_i) ;
8          canStillMerge ← true ;
9          while canStillMerge do
10             N_nb ← BestNeighborTTA(N_i, adjTable, tta) ;
11             cv ← CombinedVariance(N_i, N_nb) ;
12             if cv >= V_M^2 then
13                 SetInvalid(N_i) ;
14                 canStillMerge ← false ;
15             else if CanMerge(N_i, N_nb) then
16                 MergeRegions (N_i, N_nb, adjTable, tta) ;
17         SetInvalid(N_i) ;
```

```
1  Function MergeCache(nodeTree, adjTable[][])
2      validRegions = CountValidRegions(nodeTree) ;
3      tta ← create tta of size n_nodes ;
4      bitmap = ← create bitmap of size n_nodes ;
5      while can still select valid region do
6          N_i ← selectRegion(nodeTree) ;
7          BitmapClear(bitmap, N_i) ;
8          canStillMerge ← true ;
9          while canStillMerge do
10             N_nb ← AggregateGoodNeighours(N_i, cache) ;
11             ConsecutiveMerges(N_i, cache) ;
12         SetInvalid(N_i) ;
```

```
1  Function Aggregate(N_i, cache)
2      best ← best neighbour of N_j ;
3      var_min ← CombinedVariance(N_i, N_j) ;
4      foreach neighbour N_k of N_j do
5          if CombinedVariance(N_i, N_k) - var_min < ϵ)
           then
6              CacheAdd(cache, N_k) ;
7          if CacheFull(cache) then
8              Break ;
```

```
1  Function ConsecutiveMerges(N_i, cache)
2      canStillMerge ← true ;
3      while canStillMerge do
4          N_k ← best neighbour of N_i in cache ;
5          if can merge N_k with N_i then
6              remove N_k from cache ;
7              add neighbours of N_k in neigbours array ;
8              Merge(N_i, N_k) ;
9          else
10             canStillMerge ← false ;
```

A.3 - Résultats complémentaires



(a) Aneja

(b) TTA + Cache + Flipflop

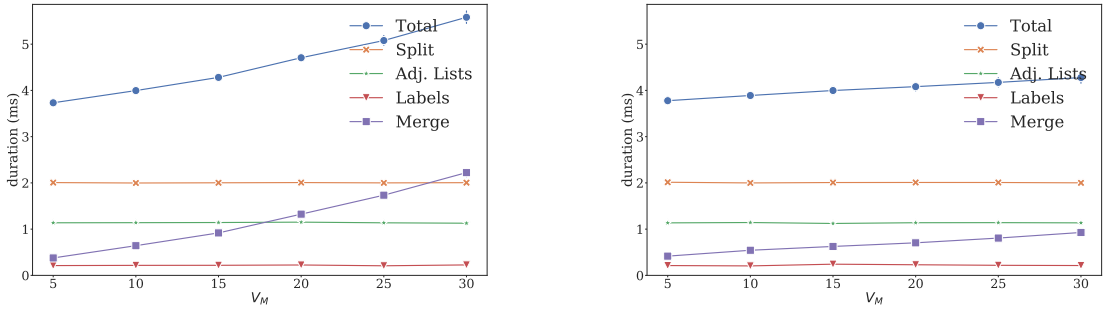
(c) Aneja

(d) TTA + Cache + Flipflop

(e) Aneja

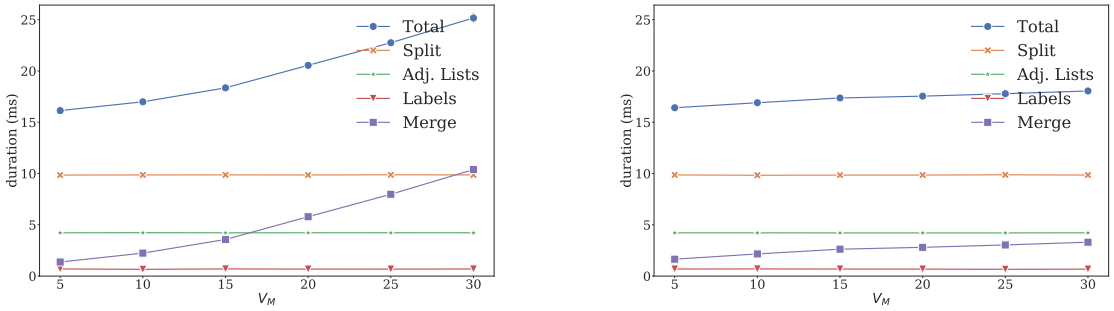
(f) TTA + Cache + Flipflop

FIGURE 10 – Temps d'exécution des étapes du *Split & Merge* pour images 256×256 (haut), 512×512 (milieu) et CAMVID (bas) pour V_S=15, sur Jetson NX



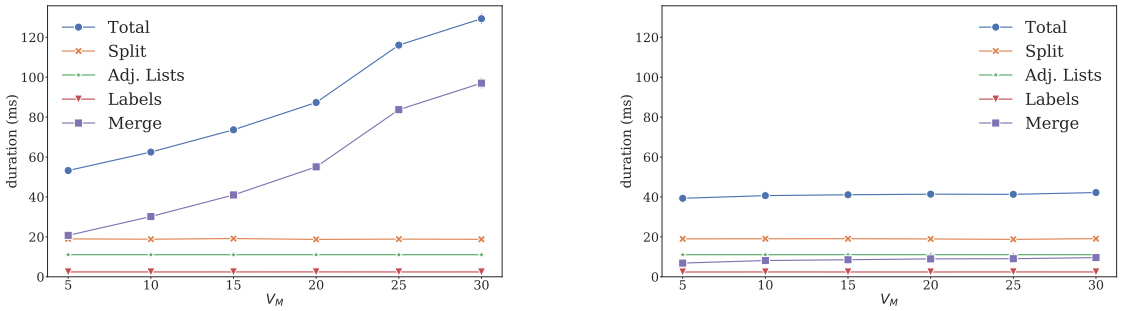
(a) Aneja (b) TTA + Cache + Flipflop

FIGURE 11 – Temps d'exécution des étapes du *Split & Merge* pour les images classiques de 256×256 pour $V_S = 15$



(a) Aneja (b) TTA + Cache + Flipflop

FIGURE 12 – Temps d'exécution des étapes du *Split & Merge* pour les images classiques de 512×512 pour $V_S = 15$



(a) Aneja (b) TTA + Cache + Flipflop

FIGURE 13 – Temps d'exécution des étapes du *Split & Merge* pour images 256×256 (haut), 512×512 (milieu) et CAMVID (bas) pour V_S=15, sur Intel Xeon montrant que les algorithmes restent efficaces sur d'autres architectures