A new efficient Split and Merge algorithm for embedded systems

Nathan Maurice, Julien Sopena, Lionel Lacassagne

A NEW EFFICIENT SPLIT & MERGE ALGORITHM FOR EMBEDDED SYSTEMS

Nathan Maurice, Julien Sopena, Lionel Lacassagne

LIP6
Sorbonne University
Paris

ABSTRACT

This article presents a new image segmentation algorithm based on a *Split & Merge* approach. By nature, the execution time of *Split & Merge* algorithms is data-dependent, as their halting conditions are tied to the homogeneity of each region. While previous algorithms made the *Split* step less sensitive to input data, the execution time of the more complex *Merge* step remains highly sensitive to image content.

This paper tackles the sensitivity and performance problems from a system and architecture perspective. Memory reallocations due to array fusions are eliminated with the introduction of a TTA (*Three Table Array*) structure in the *Merge* step. As iterating over entries in this structure causes a loss of memory locality, we propose two new mechanisms that implement a software cache to mitigate this.

An experimental study on an embedded system (Nvidia Jetson Xavier NX) has shown our *Merge* algorithm to be 10.6 times faster than the state-of-the-art *Split & Merge* algorithm for 960 $\times$ 720 images. Moreover, the execution time of our algorithm is also more resistant to image characteristics.

Index Terms— Image segmentation, Merging, Embedded systems, Data structures

1. INTRODUCTION

Image segmentation is a classical operation in image processing. It divides an image into several regions that match its perceived structure. It is one of the first steps in the resolution of complex problems: scene interpretation, object recognition, tracking, etc. A large amount of literature has been produced on this subject, notably thanks to the progress of machine learning. However, the execution time and the energy consumption of these algorithms make their use incompatible with embedded systems. Notably, *Panoptic* [1], one of the best algorithms with regard to quality, has an execution time of 100ms with a 200W GPU (Nvidia V100) to segment and label 1024 $\times$ 2048 images. It is therefore far from reaching a real-time execution time of 40ms, (which is the rate of image acquisition) or an energy consumption of roughly 10 Watts, which is what is available in heavily constrained embedded systems.

This issue can be solved by using a more simple algorithm and by making a compromise between speed/quality/power consumption. Such a lightweight algorithm can also be used to pre-process and over-segment the image before its analysis by a neural network-based algorithm, for instance, one that uses superpixels [2]. This can decrease the data size in addition to filtering unwanted noise. These lightweight algorithms can be found throughout the classical image segmentation literature: some approaches are built upon a series of splits and merges [3, 4, 5], others use watershed lines [6, 7, 8, 9], or even trees of shapes (*MaxTree*) [10, 11, 12]. However, all are still too slow and too power-hungry for our target. Moreover, the execution time of these algorithms is highly sensitive to the nature of the images rather than just their size.

In this article, we redesign a *Split & Merge* algorithm [5] by taking into consideration architectural and systems constraints on embedded systems. The proposed solution uses two new strategies that not only remove memory allocations, using a TTA data structure, but also improve memory locality by using a software cache-based approach.

A single-threaded evaluation on a Jetson Xavier NX CPU, configured to consume 15W of power highlights:

- An execution time of the *Merge* step that is *almost* independent of the merge criteria, independent of image characteristics, and almost proportional to the number of produced regions by the *Split* step.
- An acceleration by a factor that ranges from $\times$ 4 to $\times$ 10 when compared to state-of-the-art *Split & Merge* algorithms.
- An equivalent segmentation quality.

Section 2 presents the general principles of *Split & Merge* approaches. Section 3 describes the new proposed algorithms for the *Merge* step, and Section 4 evaluates our new algorithms to the *Optimal Split* & *Merge* algorithms.

2. SPLIT & MERGE SEGMENTATION

Split & Merge algorithms segment the image into multiple homogeneous regions that match the perceived structure of

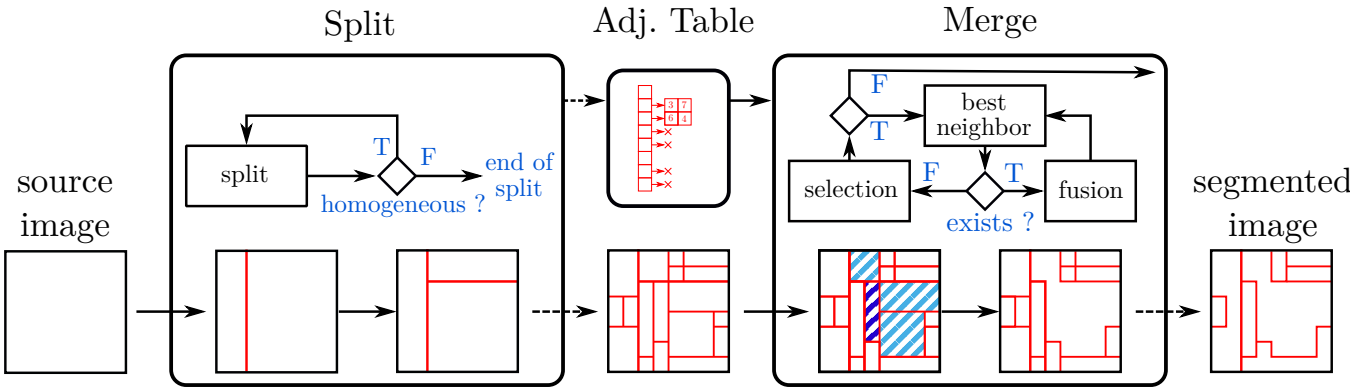


Fig. 1: Processing steps of the *Split & Merge* approach

the image. It achieves this in two major steps, as illustrated on Figure 1: a *Split* step, and a *Merge* step. The first phase recursively slices the image, either horizontally or vertically, until a homogeneity criterion is reached (*e.g.* a maximum variance in each area). Matching complex structures with rectilinear shapes results in an over-segmented image and a second step is therefore required to merge similar neighboring regions.

While the execution time of the *Split* step used to be significant, the *Optimal Split* algorithm was introduced by Merigot to both reduce this execution time and to improve the image partitioning scheme [4]. Unlike classical algorithms, which cut the current region in its center, *Optimal Split* finds an optical cut that maximizes region homogeneity (whose criterion will be noted as V_S) and reduces the number of produced regions.

While *Optimal Split* produces a better image partition, a *Merge* step is necessary to fit complex shapes (*e.g.* concave objects on the image), which combines neighbors iteratively. In order to make these iterations more efficient, Aneja et al proposed to generate an adjacency table using the partition that has been produced by the *Split* step [5]. Using this table, the *Merge* step successively selects regions of interest. For each selected region R_S, the best neighbor is searched by iterating over adjacent regions. This best neighbor of R_S is the one with the lowest combined variance with R_S, if lower than a given threshold. If no such neighbor exists (*i.e.* combined variance too high), then R_S is marked as invalid and another region is selected. Otherwise, it is merged with R_S, and the best neighbor search restarts. The *Merge* step ends when all regions of interest have been processed. Its intrinsically sequential nature causes important variations in the final partitioning. Its execution time is closely tied to image characteristics and the fusion order, and is therefore data-dependent. Moreover, this execution time is still too high for a real-time execution on embedded devices.

3. A NEW MERGE ALGORITHM

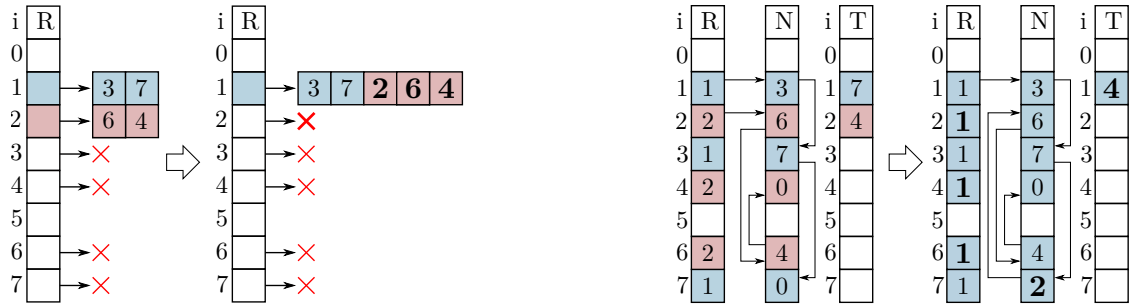
As mentioned in the previous section, the execution time of Aneja's *Split & Merge* algorithm is inadequate for real-time segmentation on smaller devices, especially because of the high cost of the *Merge* phase. This performance limitation stems from memory reallocations when concatenating lists of neighbors, which happen when merging regions.

We therefore propose a novel strategy to improve the execution time of this *Merge* step and make it more robust to image characteristics. First, we propose the use of a *Three Table Array* (TTA) data structure to avoid memory reallocations when merging lists of regions. However, iterating on TTA does not take full advantage of memory locality. To mitigate this performance issue, we propose two more mechanisms which are based upon a software cache.

3.1. Suppression of allocations in region fusions

An adjacency table is maintained by the *Merge* algorithm, as proposed by Aneja et al [5]. It contains one entry for each rectangular region initially produced by the *Split* step (as shown in Figure 2a). Algorithmically, merging two regions means concatenating two arrays. This is expensive as it generates allocations of increasing size, which leads to memory fragmentation, and therefore page faults due to `mmap`'s lazy allocation strategy.

In order to make lists fusions reallocation-free, we propose the use of a specific structure called TTA (*Three Table Array*). It is an alternative to the Union-Find structure in Connected Component Labelling algorithms [13]. It is made out of three arrays of integers R (*Root*), N (*Next*) and T (*Tail*). All three are allocated only once, at the start of the *Merge* algorithm, with their size being the number of initial rectangular regions. Lists that correspond to merged regions are represented using these integers (*e.g.* `[1, 3, 7]` and `[2, 6, 4]` in Figure 2b). As such, for each initial region: R contains the identifier of the root of the list the region belongs to (*e.g.* 1 for region 3 as it belongs to `[1, 3, 7]`), N contains the index of the



(a) With array of arrays

(b) With TTA

Fig. 2: Fusion of [1,3,7] and [2,6,4] lists

next element (*e.g.* 7 for region 3) and, if the region is the root, then T contains the index of the last element (*e.g.* 7 for region 1 because 1 is the root of `[1, 3, 7]`).

This structure simplifies list concatenations as shown by the example of Figure 2b, with the fusion of `[1, 3, 7]` and `[2, 6, 4]`. Several modifications have been made in the 3 arrays: cells 2, 6, and 4 of R now contain 1, as it is the root of the merged region. Cell 7 of N now contains 2 and links both lists. Cell 1 of T is now 4 and contains the index of the end of the new list. No memory reallocation is needed here, all modifications are done in place.

Gains provided by this new structure have been confirmed through a comparison with an array-based implementation (Figure 2a) by randomly merging lists. For different array sizes, the dotted line of Figure 3 shows the acceleration factor provided by TTA, over an array-based implementation using the `realloc` function from the `C` standard library to perform fusions. In this figure, we also show an implementation that uses the `std::vector` from `C++`. This other allocation method tries to reduce systematic reallocations by over-allocating memory (`capacity` parameter). Although this third approach sensibly improves performances, it remains less efficient than using TTA. While we notice a decrease in the efficiency of TTA when the volume of data increases due to the loss of memory locality, we do note, however, that TTA is 2.7 × faster than `realloc` whereas `std::vector` is 0.85 × for 10^6 elements, which is the targeted order of magnitude of the initial number of regions.

3.2. Addition of a cache to improve best neighbor search

As seen previously, switching to TTA creates a memory locality problem. Indeed, the best neighbor is searched consecutively using a variance criterion. This implies iterating multiple times on the neighbor list. While this was done on a contiguous array in the original *Split & Merge* algorithm, our approach can generate multiple jumps in the `N` array when using a TTA. In order to reduce the impact, we introduce a software cache can better re-use work done during the previous iteration, and a mechanism that will take a set of

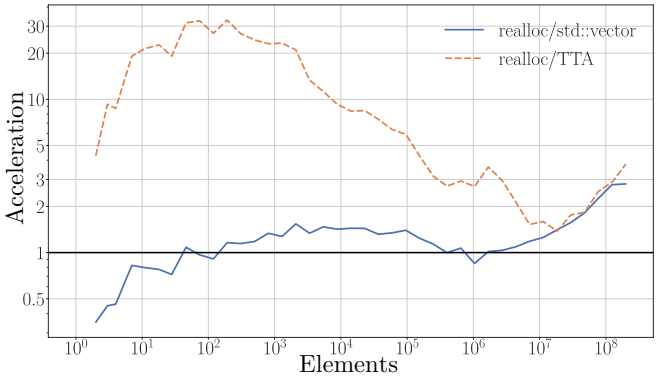


Fig. 3: Acceleration of list fusion for TTA and `std::vector` compared to `realloc`

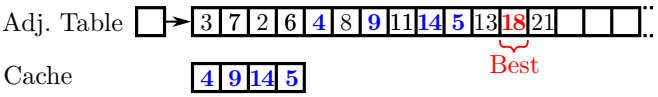


Fig. 4: Insertion of $[4, 9, 14, 5]$ neighbors in software cache. The best neighbor (18) is immediately merged whereas *good* neighbors $(4, 8, 14, 5)$ are inserted in the cache.

candidates of similar variances.

Our proposed Algorithm 1 iterates over the full neighborhood twice: a first time to find the best neighbor (`L8`), and a second time to aggregate close enough neighbors into the cache, *i.e.* elements for which the variance from the best neighbor is less than ϵ (`L11-17`). Figure 4 presents an example of an aggregation and insertion into a cache of *good* neighbors: $(4, 9, 14, 5)$ are close enough to the best neighbor (18) and are thus inserted into the cache.

Following this aggregation, successive fusions are done on the neighbors in the cache (`L20-27`), which are then removed from it (`L23`). During the fusion of a region R_k into R_i, neighbors of R_k are added to the neighborhood list (`L24`). They will therefore be able to be picked by the next aggregation. Fusions continue until there is a shortage of mergeable neighbors.

The number of neighbors that have to be checked each time

is therefore reduced. Moreover, these searches are done on a continuous array. This does not only have an impact on computation time but also data locality and therefore uses hardware caches more efficiently.

```
1  Function MergeCache(nodeTree, adjTable[][])
2      validRegions ← CountValidRegions(nodeTree)
3      tta ← create tta of size n_nodes
4      while can still select valid region do
5          N_i ← selectRegion(nodeTree)
6          canStillMerge ← true
7          while canStillMerge do
8              // Aggregate best neighbors
                 best ← best neighbor of N_j
9              v_min ← CoVariance(N_i, N_j)
10             if can not merge with best then
11                 break
12             // Iterate over full neighborhood
13             foreach neighbor N_k of N_j do
14                 if CoVariance(N_i, N_k) − v_min < ϵ then
15                     CacheAdd(cache, N_k)
16                 if CacheFull(cache) then
17                     break
18             // Try to merge neighbors from cache
19             cacheOK ← true
20             while cacheOK do
21                 N_k ← best neighbor of N_i from cache
22                 if can merge N_k with N then
23                     remove N_k from cache
24                     MergeTTA(tta, N_i, N_k)
25                     update N_i
26                 else
27                     cacheOK ← false
28         SetInvalid(N_i) // Do not select N_i again
```

Algorithm 1: Find and merge best neighbors using software cache

3.3. A Flip flop caching strategy

While our cache-based approach improved memory locality, neighbor aggregation still requires a double iteration: first to find the best neighbor and then to insert *good enough* neighbors into the cache.

We therefore propose a single pass strategy: when iterating on the neighborhood, if a neighbor is found to have a better variance than all its predecessors, then it is immediately added to the cache and used as a reference point. Neighbors that are still close enough to the current best neighbor are also added to the cache. Unlike the two-pass version, some neighbors may be inserted into the cache even if they are initially outside the tolerance range ($> \epsilon$ away from best neighbor).

With this idea, new neighbors are added at the end of the neighbor list, whereas old and unlikely-to-be-merged neighbors stay at the beginning. This can become a worst-case as aggregations start from the beginning of the list, and would have to iterate over "bad" neighbors first.

To remedy this while still benefiting from memory locality, we introduce a flipflop approach: the run remains sequential, but its direction flips each iteration. The situation is now flipped and what used to be a problem is now an advantage: Neighbors that will be cached will likely be discovered quickly. The insertion accuracy will then become equivalent to that of the two-pass algorithm (in which insertions are performed by knowing the reference variance)

4. QUALITATIVE AND QUANTITATIVE BENCHMARKS

To compare ourselves to Aneja fairly, we first have to ascertain the quality of the final segmentation. Indeed, due to the sequential nature of *Split & Merge* algorithms, small changes can lead to major differences in segmented images. Our goal is an efficient *Merge* step that produces a similar segmentation to Aneja.

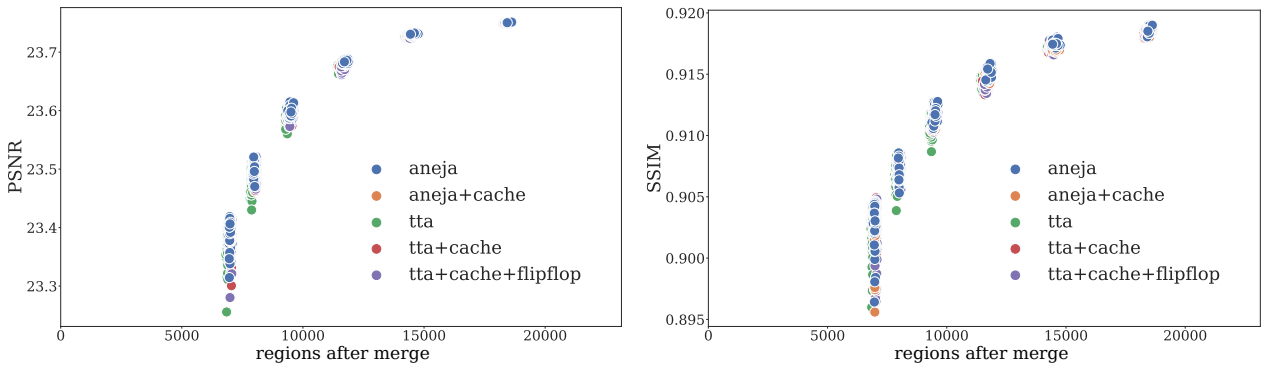
This verification is performed using two similarity metrics: the *Peak Signal to Noise ratio* (PSNR) and *Structural Similarity Index Measure* (SSIM) [14]. They quantify the distance between two images: two identical images will have an "infinite" PNSR and an SSIM of 1. On the other hand, the more the images differ, the more the PSNR and SSIM will move towards 0.

In our case, these metrics describe how similar the segmentations are with regard to the original image. If our algorithms are correct, then we should observe two things: **(1)** the number of regions must the same as Aneja, **(2)** the PSNR and SSIM of the segmented images with respect to the source images must be close to the ones reached by Aneja. To test this, the algorithms have been executed on 11 images from the CamVid urban image dataset [15, 16] (960×720 image sizes), with several values for the fusion criterion V_M.

The evaluation of both quality metrics is shown in the scatter plot of Figure 5, with a homogeneity criterion V_M varying from 5 (few fusions, therefore many regions after *Merge*) to 30 (many fusions, therefore few regions after *Merge*), with a step of 5 (*i.e.* 6 different values of V_M). For each algorithm, 50 runs are performed per value of V_M, which gives us 50 different segmentations per algorithm and value of V_M, due to the randomness of *Merge*. Each point matches 1 run of an algorithm at a given V_M: the number of regions after *Merge* is presented on the x-axis, and the quality metric (PSNR or SSIM) on the y-axis.

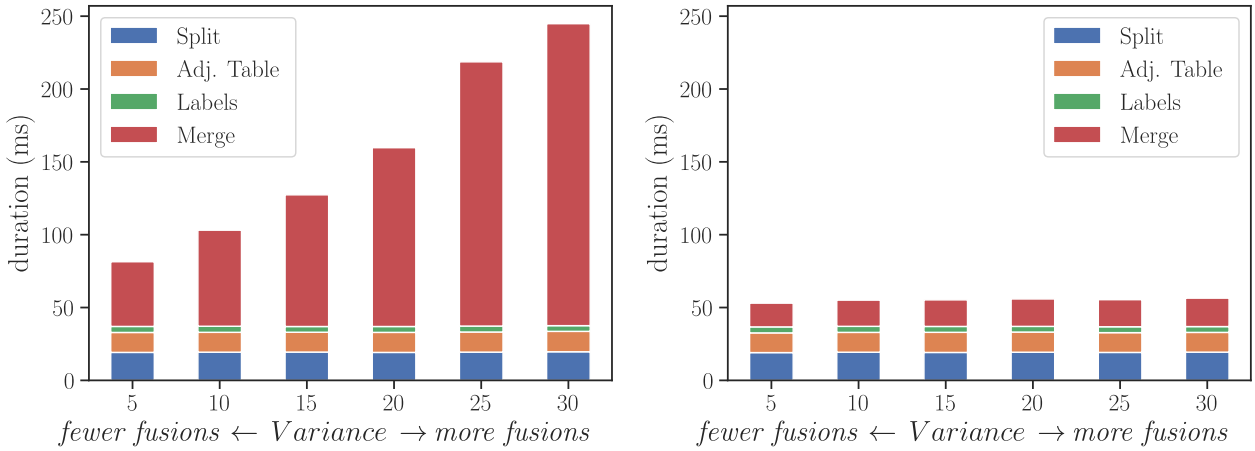
6 clusters of points can be observed in Figure 5: each cluster corresponds to a value of V_M, and contains 300 points (6 algorithms $\times$ 50 runs per algorithm).

Within each cluster (same *Merge* criterion V_M), the number of regions after *Merge* (x-axis) stays the same between runs of the same algorithm. The segmentations of the algorithms are therefore stable for the given criterion. Moreover, at a given V_M, our proposed algorithms keep the same number of regions as Aneja, which confirms **(1)**.



(a) PSNR (b) SSIM

Fig. 5: PSNR and SSIM of final segmentation for the first image of CAMVID, $V_S = 15$. (1 point = 1 run of an algorithm, for a given V_M)



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

Fig. 6: Execution time of individual steps of *Split & Merge* on CAMVID for V_S=15, on Jetson Xavier NX

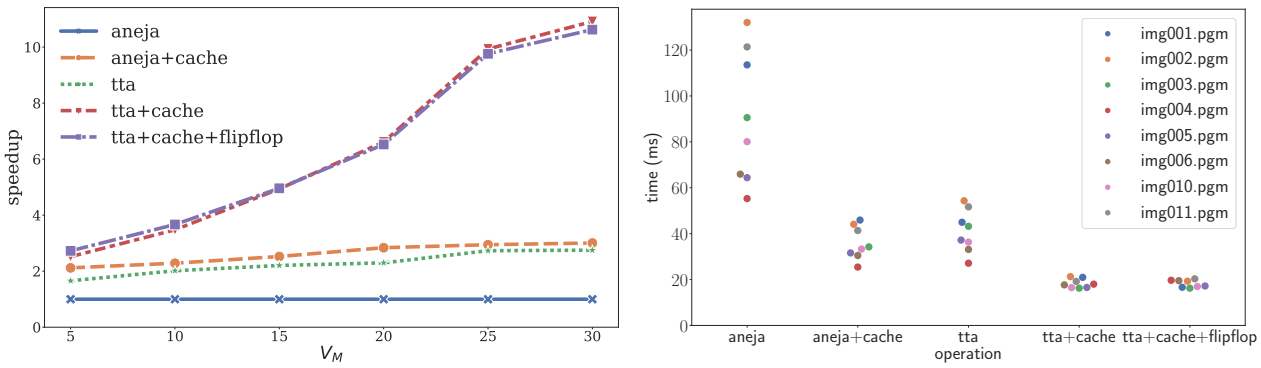
If we now look at segmentation quality: in the same cluster, runs of an algorithm have different PSNR and SSIM (y-axis). However, the magnitude of the variations is the same for all algorithms, which highlights that our algorithms have the same qualitative behavior as Aneja. Furthermore, the range of SSIM and PSNR values within a cluster is comparable between algorithms. The PSNR and SSIM also become stable when more regions are left ($V_M \leq 10$). These observations confirm that our algorithms maintain the same quality as Aneja (which proves **(2)** correct).

Having numerically proved the correctness of our algorithms with respect to Aneja, we now check this visually. By doing so, we observe that the segmentations of our algorithms are indistinguishable from Aneja's.

After having shown that our algorithms give good results, we investigate which ones are the fastest for the same quality. We now measure the execution time of different *Split & Merge* algorithms. A Jetson Xavier NX CPU has been used as it is both highly efficient (custom Cortex ARM, close to A76) and highly customizable (several profiles with 2, 4, or 6 cores at varying clock frequencies). In our experiments, a power profile of 15 Watts has been used (with 2 active cores instead of 6). All *Split & Merge* implementations are single-threaded, with their executions tied to a single core, and they are preceded by a warm-up round. Random numbers have been generated using Mersenne Twister 19337 [17].

Execution times have been measured on the 11 CamVid images, for parameters values V_S and V_M going from 5 to 30. This interval allows for fine-grained segmentations. On each image, 5 different algorithms have been tested: Aneja, a version with TTA only, a version with TTA and software cache, TTA with flipflop, and Aneja with cache. For each image, 50 runs are performed per configuration V_M. The size of the software cache is 32 and $\epsilon = 500$.

The TTA structure provides a speedup of $\times 1.7$ to $\times 2.8$ when compared to Aneja (Figure 7a). The addition of a cache to Aneja also improves the execution time, with a speedup that ranges from $\times 2.1$ to $\times 3.0$. However, the combination



(a) Acceleration factor of our algorithms compared to Aneja on CamVid, with V_S=15

(b) Normalized execution time of *Merge* step for CamVid, with V_S=15, V_M=15

Fig. 7: Execution time for CAMVID, on Jetson Xavier NX

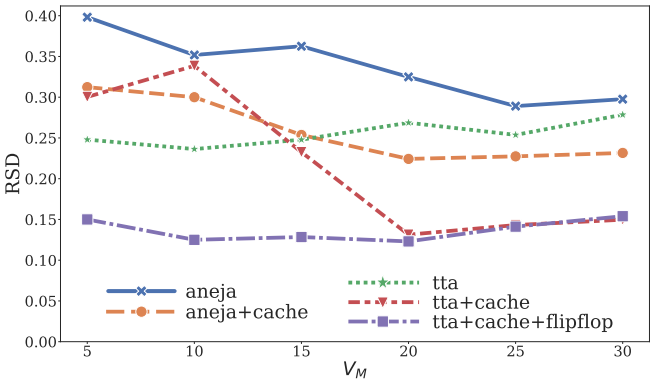


Fig. 8: Relative standard deviation on CamVid, $V_S = 15$

of both TTA and the cache gives a considerable acceleration, from 2.5 × to 10.9 × for *tta+cache* and 2.7 × to 10.6 × for *tta+cache+flipflop*. As execution time of the flipflop-based algorithm remains close to that of TTA+*cache*. The management of the flipflop mechanism likely has a non-negligible cost compared to the time spent on the double pass.

An interesting property can be observed in Figures 7b and 8: the execution time of *tta+cache* and *tta+cache+flipflop* on the image set is not only shorter, but also has a lower variation between images. This is not the case for Aneja, which has a variation from single to double. This pattern is also found when observing Figure 8, with the relative standard deviations of these studied algorithms. The addition of flipflop reduces the variations regardless of the variance parameter. We also note that *tta+cache+flipflop* remains stable, at 0.15, whereas *tta+cache* varies between 0.35 and 0.15. This means that *tta+cache+flipflop* is more robust to image data than the rest, as image structure has a lesser impact on the execution time.

Now that we have shown the efficiency of our proposed *Merge* algorithm, we study the full algorithms (*i.e.* with both the *Split* phase and our *Merge* mechanisms). Their execution times can be observed in Figure 6, which includes the duration of each step (*Split*, *Merge*, and intermediate steps such as the initialization of the adjacency table and the final labeling). While the execution time of Aneja (Figure 6a) increases significantly with the number of fusions, this is not the case for *TTA+cache+flipflop* (Figure 6b). Its *Merge* step has a stable execution time of roughly 20 ms for CamVid images and 5 ms for classical 512 × 512 images. The *Split* and the *Merge* steps have similar execution times and can therefore be pipelined in a balanced manner on two cores: the *Split* step on a first core and the other steps (labeling, adjacency list computation, *Merge*) on the second core. A "real-time" video rate of 25 images per second (40 ms per image) would then be reached.

5. CONCLUSION

In this article, we have proposed a new *Merge* algorithm that significantly improves the execution speed of the *Split & Merge* image segmentation approach.

List fusion, one of the key operations of the *Merge* step, is now performed without memory reallocation or array copy, using a TTA data structure. We also introduced a software cache mechanism, for which we added a flipflop iteration strategy to compensate for the loss of memory locality, and make best neighbor search as fast as possible.

Other than the raw execution time gains, both proposed mechanisms make *Split & Merge* segmentation more robust to intrinsic image characteristics. This makes them well-suited to applications on embedded systems, in which respecting a real-time video rate cadency (typically 25 images / second or 40 ms) is a key criterion, and requires a predictable processing time.

Finally, the performance improvement of the *Merge* step brings it close to the duration of the *Split* phase, which makes it possible to pipeline the entire algorithm on two cores in a balanced manner. As such, *Split & Merge* algorithms can be used at a real-time video rate on embedded systems and for large images.

6. REFERENCES

[1] Rui Hou, Jie Li, Arjun Bhargava, Allan Raventos, Vitor Guizilini, Chao Fang, Jerome Lynch, and Adrien Gaidon, "Real-Time Panoptic Segmentation from Dense Detections," in *Conference on Computer Vision and Pattern Recognition*, Apr. 2020. 1

[2] Shengfeng He, Rynson W. H. Lau, Wenxi Liu, Zhe Huang, and Qingxiong Yang, "SuperCNN: A Superpixelwise Convolutional Neural Network for Salient Object Detection," *Int J Comput Vis*, vol. 115, no. 3, pp. 330–344, Dec. 2015. 1

[3] Steven L. Horowitz and Theodosios Pavlidis, "Picture Segmentation by a Tree Traversal Algorithm," *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 368–388, Apr. 1976. 1

[4] A. Merigot, "Revisiting image splitting," in *12th International Conference on Image Analysis and Processing, 2003.Proceedings.*, Mantova, Italy, 2003, pp. 314–319, IEEE Comput. Soc. 1, 2

[5] Kanur Aneja, Florence Laguzet, Lionel Lacassagne, and Alain Merigot, "Video-rate image segmentation by means of region splitting and merging," in *2009 IEEE International Conference on Signal and Image Processing Applications*, Kuala Lumpur, Malaysia, 2009, pp. 437–442, IEEE. 1, 2

[6] Serge Beucher and Fernand Meyer, *The Morphological Approach to Segmentation: The Watershed Transformation*, pp. 433–481, CRC Press, first edition, 1993. 1

[7] Jos B.T.M. Roerdink and Arnold Meijster, "The Watershed Transform: Definitions, Algorithms and Parallelization Strategies," *Fundamenta Informaticae*, vol. 41, no. 1,2, pp. 187–228, 2000. 1

[8] Romaric Audigier and Roberto Lotufo, "Uniquely-Determined Thinning of the Tie-Zone Watershed Based on Label Frequency," *Journal of Mathematical Imaging and Vision*, vol. 27, no. 2, pp. 157–173, Feb. 2007. 1

[9] Yosra Braham, Yaroub Elloumi, Mohamed Akil, and Mohamed Hedi Bedoui, "Parallel computation of Watershed Transform in weighted graphs on shared memory machines," *Journal of Real-Time Image Processing*, vol. 17, no. 3, pp. 527–542, 2018. 1

[10] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, Apr. 1998. 1

[11] Ronald Jones, "Connected Filtering and Segmentation Using Component Trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, Sept. 1999. 1

[12] Nicolas Passat, Benoît Naegel, François Rousseau, Mériam Koob, and Jean-Louis Dietemann, "Interactive segmentation based on component-trees," *Pattern Recognition*, vol. 44, no. 10-11, pp. 2539–2554, Oct. 2011. 1

[13] Lifeng He, Yuyan Chao, and Kenji Suzuki, "A Linear-Time Two-Scan Labeling Algorithm," in *2007 IEEE International Conference on Image Processing*, San Antonio, TX, USA, 2007, pp. V – 241–V – 244, IEEE. 2

[14] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, Apr. 2004. 4

[15] Gabriel J. Brostow, Julien Fauqueur, and Roberto Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, Jan. 2009. 4

[16] Gabriel J. Brostow, Jamie Shotton, Julien Fauqueur, and Roberto Cipolla, "Segmentation and Recognition Using Structure from Motion Point Clouds," in *Computer Vision – ECCV 2008*, David Forsyth, Philip Torr, and Andrew Zisserman, Eds., vol. 5302, pp. 44–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, Series Title: Lecture Notes in Computer Science. 4

[17] Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 26, Jan. 1998. 5