



Parallel watershed partitioning: GPU-based hierarchical image segmentation

Varduhi Yeghiazaryan ^{a,b}, Yeva Gabrielyan ^{a,c}, Irina Voiculescu ^{b,*}

^a American University of Armenia, 40 Marshal Baghramyan Ave, Yerevan, 0019, Armenia

^b University of Oxford, Department of Computer Science, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

^c Denovo Sciences LLC, 138/2 Verin Antaran Str, Yerevan, 0009, Armenia



ARTICLE INFO

Keywords:

Watershed transform
GPU acceleration
Waterfall transform
Path reduction
Parallel union–find
Real-time image segmentation

ABSTRACT

Many image processing applications rely on partitioning an image into disjoint regions whose pixels are ‘similar.’ The watershed and waterfall transforms are established mathematical morphology pixel clustering techniques. They are both relevant to modern applications where groups of pixels are to be decided upon in one go, or where adjacency information is relevant. We introduce three new parallel partitioning algorithms for GPUs. By repeatedly applying watershed algorithms, we produce waterfall results which form a hierarchy of partition regions over an input image. Our watershed algorithms attain competitive execution times in both 2D and 3D, processing an 800 megavoxel image in less than 1.4 sec. We also show how to use this fully deterministic image partitioning as a pre-processing step to machine-learning-based semantic segmentation. This replaces the role of superpixel algorithms, and results in comparable accuracy and faster training times. The code is publicly available at <https://github.com/hamemm/PRUF-watershed.git>.

1. Introduction

Image partitioning was originally known in Computer Vision under the name of ‘image segmentation’. Nowadays, ‘image segmentation’ often refers to semantic segmentation, where an object is segmented out of an image. The *fully deterministic* techniques proposed in this work refer to *partitioning an image* into disjoint regions of interest—with the ultimate downstream goal of improving segmentation or classification algorithms.

With a constant increase in the resolution of input into imaging algorithms, many techniques resort to processing multiple pixels in one go. This is achieved through downsampling the input image gradually through the depths of the layers of a machine learning network, or through grouping pixels together into superpixels [1,2]. Pixel grouping is carried out as a one-off pre-processing step and merits particular attention due to the quality of its results influencing the downstream pipeline.

In the 1990s, image partitioning was synonymous with the watershed transform, which has been utilized for segmentation and feature extraction [3–6]. The watershed has also been used as a pre-processing [7,8] or post-processing [9–11] step in various deep learning (DL) applications. However, watersheds have two main drawbacks:

over-segmentation (regions are too small) and slow processing times (if executed sequentially).

Our previous work [12,13] proposed three fast GPU watershed algorithms: PRW, APRUF and PRUF, which bring the watershed up to modern requirements. This paper builds on that previous work by speeding up and extending it to employ watershed algorithms multiple times to achieve a hierarchical waterfall transformation, mitigating over-segmentation.

Our goal is to construct hierarchical segmentations (or partitions) of decreasing granularity, from input images of ultra-high resolution. This is achieved through deploying fast, parallel GPU-based watershed and waterfall transforms. These cluster pixels into equivalence classes through Union–Find, and then cluster these classes into larger partitions. The steps are: 1. smooth the input image (minimally), 2. construct a gradient magnitude image, 3. apply the watershed transform, and 4. repeatedly apply the waterfall transform. Fig. 1 illustrates output from this pipeline on two sample images: cameraman and knee magnetic resonance imaging (MRI) slice. The segmented region boundaries, called watershed lines, in yellow are overlaid on the original greyscale images.

The results of our experiments carried out on multiple datasets and GPUs demonstrate that the proposed algorithms outperform previous watershed algorithms for GPU execution in both effectiveness and effi-

* Corresponding author.

E-mail address: irina@cs.ox.ac.uk (I. Voiculescu).

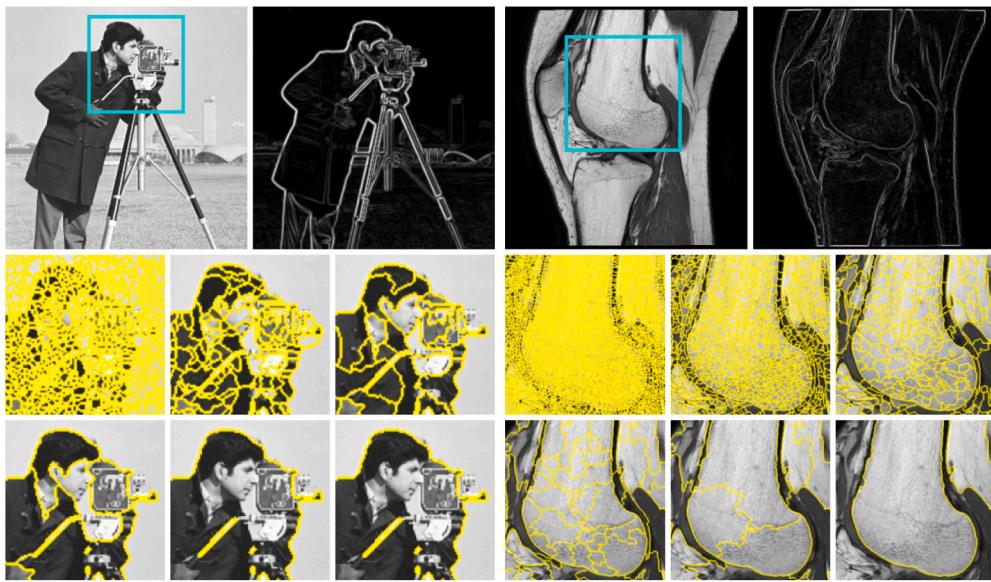


Fig. 1. Hierarchical output on two input images: cameraman and knee MRI slice. Top: raw greyscale inputs, gradient magnitude images (after Gaussian blurring); middle and bottom: cropped originals overlaid with watershed/waterfall lines in yellow for $NL=6$ hierarchical layers.

ciency. Furthermore, we show that the proposed algorithm for the hierarchical waterfall transform can be used as a replacement for superpixel algorithms in the pre-processing step of DL approaches to hyperspectral image classification applications (up to $3.9\times$ speedup). This work highlights the versatility, relevance, and continued utility of parallel watershed/waterfall algorithms in Computer Vision.

The main contributions of this paper are as follows:

- (i) three GPU watershed algorithms (PRW, PRUF, APRUF); earlier versions in our conference papers [12,13];
- (ii) novel GPU waterfall that mitigates over-segmentation;
- (iii) analysis of the effect of parameters (connectivity);
- (iv) Computer Vision applications of waterfall algorithm.

The remainder of the article is arranged as follows. Sec. 2 presents relevant literature. Sec. 3 and 4 describe our algorithms in detail. Experimental results are given in Sec. 5 and 6, followed by Conclusions in Sec. 7.

2. Related work

The watershed is a greyscale image transformation from mathematical morphology that partitions the image into multiple small regions. Images are interpreted as topographic terrain maps: white pixels correspond to the highest and black to the lowest altitudes. The watershed transform segments the image into disjoint regions called catchment areas, or basins. These regions are formed around regional minima in the topographic terrain. The boundary pixels separating the different catchment basins are called watershed lines, or watersheds. When the watershed transform is applied to the gradient magnitude image, the boundaries of catchment basins are placed at pixels with high gradient magnitude values (these correspond to boundaries of regions of interest).

A detailed discussion of watersheds is given in [14]. Implementations of watersheds with distributed memory parallelism [15,16], shared memory parallelism [17,18] and GPUs [19–22] have also been explored. Watersheds on GPUs have been the subject of recent research due to the increasing demand for faster image processing. Initially GPU implementations of the watershed used ‘flooding’ [23,19,24,25], but hybrid [26,27] (named DW in the experimental comparison below) and parallel depth-first watershed algorithms [28,20,22,29] have also been proposed.

Körbes et al. showed in [20] that the CUDA implementation of their DW algorithm (original implementation accessed from [30]) was superior to an algorithm proposed by Kauffmann and Piché [19,24]. A more detailed description of DW, including procedure pseudocode, can be found in [28].

Quesada-Barriuso et al. [22,29] introduced another parallel watershed for GPU based on cellular automata (named CA in the experimental comparison below), the implementation of which was kindly provided to us by the authors. We do not compare directly against [23–25] because they have been superseded.

Computer Vision applications of the watershed [31,3–6] include image segmentation, object boundary detection, image restoration. It is also commonly used as a pre- or post-processing step for various applications working with microscopic images [9,10], micro-CT images [5].

The waterfall [32,33] is a hierarchical segmentation transform based on the watershed. It is a merging technique which addresses over-segmentation. The main idea of waterfall is that catchment basins from the watershed get turned into new plateaux and the watershed is re-applied. The resulting image contains a reduced number of regional minima. This recursive process produces a series of image partitionings with a monotonically decreasing number of catchment basins, until a single region corresponds to the whole image. Aside from the waterfall that can be directly applied on an image, a number of graph-based waterfall algorithms have been proposed for fast sequential implementation [34,35].

In order to mitigate the shortcomings of classical waterfall (early removal of important contours from the hierarchy and the difficulty of finding a ‘good’ depth of hierarchy) three modifications of the waterfall transform have been proposed. The standard and the P algorithms [36] replace the hidden parameter called gain with an explicit one. Enhanced waterfall transform [37] re-introduces selected contours into the watershed results of each level of the hierarchical segmentation based on specific rules. An intuitive application of the classical waterfall transform is the construction of image partition forests (IPFs) [38,39].

3. Proposed watershed algorithms

We present three parallel algorithms for the watershed transform: path reducing watershed (PRW), path reducing UNION–FIND watershed (PRUF) and alternative PRUF watershed (APRUF). In the next section we show that, with the addition of two CUDA kernels, it is possible to produce a waterfall algorithm from each of these watershed algorithms.

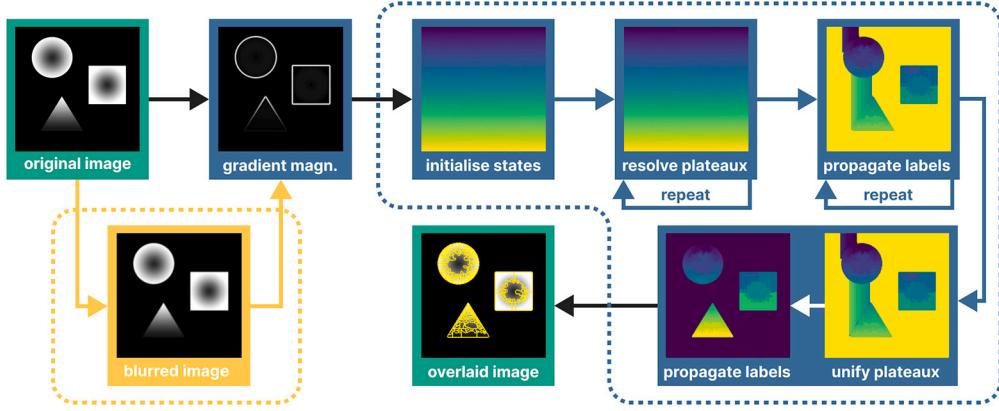


Fig. 2. PRUF pipeline inside blue dashed outline. Each of steps I–III corresponds to one CUDA kernel: I executed once, II and III repeated iteratively. Step IV is achieved through two CUDA kernels, each executed once. Input image smoothing and use of gradient magnitude image are optional.

Each one of our parallel watersheds consists of four major steps: I: initialise pixel states and labels, II: resolve non-minimal plateaux, III: propagate labels, or reduce paths, and IV: merge minimal plateau labels. This process belongs to the class of topographic-distance-based watershed transforms; it exploits paths of steepest descent. The four steps can be alternatively viewed as: I: initialise paths of steepest descent; II: resolve paths on non-minimal plateaux; III: reduce paths into pointers to local minima; IV: add extra paths inside minimal plateaux to achieve a single pointer target, i.e. label, for each catchment basin.

Fig. 2 illustrates the general workflow for PRUF. Details of PRW and APRUF are described later in the text.

3.1. Path-reducing, UNION–FIND watershed

Algorithm 1 Path reducing, UNION–FIND watershed.

```

1: for all  $p \in P$  in parallel do {Step I: initialise labels & states}
2:    $q := \max\{r \in N(p) \mid \forall n \in N(p), I(r) \leq I(n)\}$ 
3:   if  $I(q) < I(p)$  then
4:      $L(p) := q; S(p) := 0$ 
5:   else if  $I(q) > I(p)$  then
6:      $L(p) := p; S(p) := 1$ 
7:   else if  $q > p$  then
8:      $L(p) := q; S(p) := 2$ 
9:   else if  $q < p$  then
10:     $L(p) := p; S(p) := 3$ 
11: repeat {Step II: resolve non-minimal plateaux}
12:   for all  $p \in P$  in parallel do
13:     if  $S(p) \geq 2$  and  $\exists q \in N(p), S(q) = 0 \wedge I(q) = I(p)$  then
14:        $L(p) := q; S'(p) := 0$ 
15:     else
16:        $S'(p) := S(p)$ 
17:     swap( $S, S'$ )
18:   until  $\neg\text{change}$ 
19: repeat {Step III: propagate labels, or reduce paths}
20:   for all  $p \in P$  in parallel do
21:     for  $i := 1, RR$  and  $L(p) \neq L(L(p))$  do
22:        $L(p) := L(L(p))$ 
23:     until  $\neg\text{change}$ 
24:   for all  $p \in P$  in parallel do {Step IV: merge min plateau labels}
25:     if  $S(p) \geq 2$  then
26:       for all  $q \in N(p), q > p, S(q) \geq 2$  do
27:         UNION( $L, p, q$ )
28:     for all  $p \in P$  in parallel do
29:        $L(p) := \text{FIND}(L, p)$ 

```

Algorithm 1 details PRUF. Let P be the set of all pixel positions in the image, and for every $p \in P$ let the local neighbourhood $N(p)$ be the set of pixels in the image that are a unit distance away from p . The

obvious choices are either Moore neighbourhood with 8 pixels in 2D and 26 voxels in 3D, or von Neumann neighbourhood with 4 pixels in 2D and 6 voxels in 3D, both reported hereafter.

During step I: initialisation, all pixels are classified into four groups, called states (S), using the image values of the neighbouring pixels. Our algorithm requires that all pixels in the image be ordered; our implementation uses row-major order. Let $q \in N(p)$ be the pixel neighbouring p that has the smallest input image value. If the minimum value occurs at multiple neighbours, let q be the last in the fixed order among all pixels in that neighbourhood that have the minimum image value:

$$q = \max\{r \in N(p) \mid \forall n \in N(p), I(r) \leq I(n)\}, \quad (1)$$

where $I(p)$ is the image value at pixel $p \in P$. Fig. 3 details the steps of the watershed on a small example.

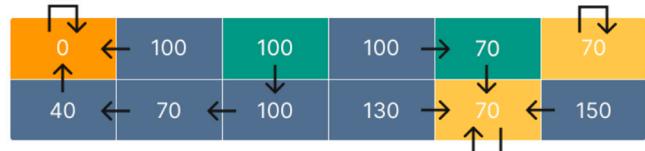
- If $I(q) < I(p)$, then the direction of steepest descent from p is q . The initial label L of p is set to q ; later it will borrow its eventual label from q . Thus, pixel p is assigned to be in state 0 to indicate it has a lower neighbour ($S(p) = 0$).
- If $I(q) > I(p)$, then p is a local minimum. We set $L(p) = p$, so that later this value can be used to label the whole catchment basin. The set of local minima is indicated by their states fixed at 1 ($S(p) = 1$).
- Lastly, if $I(q) = I(p)$, then p is inside a plateau. We differentiate such pixels into two states:
 - a plateau pixel that has at least one equal neighbour (row-major order) after it ($S(p) = 2$) and
 - a plateau pixel that have equal neighbours (row-major order) only before it ($S(p) = 3$).

If the plateau is eventually, after step II, categorised as minimal, state-3 plateau pixels will lend their label to state-2 plateau pixels. Hence, $L(p) = q$ if $S(p) = 2$ and $L(p) = p$ forms a self-loop if $S(p) = 3$.

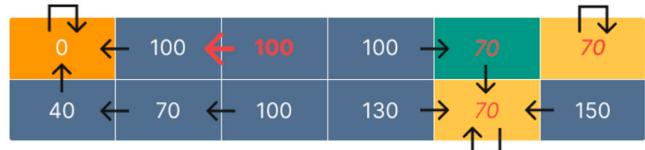
Step II resolves non-minimal plateaux, which cannot be set apart by looking only at immediate neighbours. This step propagates labels inwards into a non-minimal plateau from its boundaries. For every state-2 or state-3 pixel p , $S(p) \geq 2$, if $\exists q \in N(p)$ such that $S(q) = 0$ and if $I(q) = I(p)$, then $S(p)$ is reset to 0 and $L(p)$ is updated to equal q . We repeat this process iteratively until convergence with no state updates. The states are buffered during iterations and swapped between iterations to achieve consistent propagation of labels into non-minimal plateaux (see further discussion in Sec. 3.4).

After step II the pixel states can be seen in a new way, slightly different from the initial interpretation. State 0 corresponds to pixels with a lower neighbour and non-minimal plateau pixels. States 2 and 3 only mark pixels within minimal plateaux. Since labels point to ei-

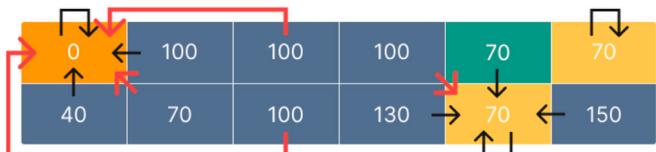
0	100	100	100	70	70
40	70	100	130	70	150

(a) 6×2 greyscale input with expected watershed line

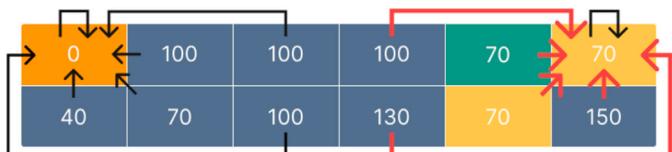
(b) step I: initialise states (colours) and labels (arrows)



(c) step II: update non-minimal plateau states and labels



(d) step III: propagate minima labels uphill to all pixels



(e) step IV: merge minimal plateau labels

Fig. 3. Simple 6×2 greyscale input image illustrates our path reducing watershed algorithms (4-pixel neighbourhood). State and label updates are highlighted in red boldface; red italics indicate that an update was considered but not performed. Pixel states are colour-coded: state 0, state 1, state 2, and state 3.

ther a neighbouring pixel or themselves and are acyclic, they can be followed recursively to create directed paths between pixels. The labels of all state-0 pixels form paths leading to minima, specifically state-1 or state-3 pixels. These steepest descent paths define different catchment basins. Additionally, within minimal plateaus, the labels of state-2 pixels create further paths to state-3 pixels. Therefore, any label path starting from any pixel will ultimately lead to a state-1 or state-3 pixel, where the label forms a self-loop.

In step III, all label paths are reduced into direct pointers from any image pixel into its corresponding catchment basin minimum state-1 or state-3 pixel. This process is performed independently and in parallel for each pixel, following the label pointers until a self-loop is encountered. However, this procedure requires time proportional to the length of the longest label path, making the worst-case time complexity linear relative to the image size. To reduce this complexity to logarithmic, we synchronize all labels after a fixed number of updates and repeat the process iteratively. This fixed number of updates between synchronizations is called *reduction rate (RR)*. The label of each pixel p is updated to its label's label RR times; the new label value is recorded instead of the original $L(p)$ and made available to other pixels to read. This iterative process continues

until no further label changes occur. The overall number of operations is $O(RR \times \lceil \log_{RR+1} Len \rceil) = O(\lceil \log_{RR+1} Len \rceil) \leq O(\log_{RR+1} |P|)$, where RR is the reduction rate, Len is the length of the longest path, $|P|$ is the image size. Fig. 4 demonstrates path reduction.

For a 1D image, the label array produced after steps I-III of the algorithm constitutes a valid watershed result. This is because each minimal plateau in a 1D image forms an interval, and consequently, the only state-3 pixel within such a plateau is the rightmost pixel. As a consequence, after step III, each catchment basin in the label array has a single value. Still, this property does not hold for higher-dimensional images. Based on the shape of a minimal plateau, there may be several state-3 pixels in it. Fig. 5 shows a few minimal plateau shapes with state-3 pixels highlighted.

Step IV addresses the problem of multiple labels for minimal plateaus by merging those labels. Once steps I-III are complete, the pixel labels can be viewed as a *disjoint-set* data structure. A disjoint set of pixels is formed for each one label, and all set elements point directly to the set representative. One or more disjoint sets correspond to every catchment basin in the watershed partitioning. When several labels, i.e. disjoint sets, overlap within the same catchment basin, it indicates that the basin has a plateau minimum containing multiple state-3 pixels (which can occur in 2D or higher-dimensional images). To address this, we use a parallel UNION-FIND implementation [40] to merge all disjoint sets that represent the same catchment basin. Step IV performs a UNION whenever it detects neighbouring minimal-plateau pixels with varying labels, thus merging the different labels within the same catchment basin. Finally, it updates each pixel's label to the index, or label, of the set representative by executing a FIND operation.

When we introduced PRW [12] we showed it was significantly faster than existing approaches. It differs from PRUF only in step IV. The pseudocode for that step is presented in Algorithm 2. A further improvement over [12] and [13] is to consider only half of the neighbourhood in step IV ($q > p$ on line 26 of Algorithm 1 and line 27 of Algorithm 2).

Algorithm 2 Path reducing watershed.

{Steps I-III are the same as in Algorithm 1}

24: **repeat** {Step IV: merge minimal plateau labels}

25: **for all** $p \in P$ **in parallel do**

26: **if** $S(p) \geq 2$ **then**

27: **for all** $q \in N(p), q > p, S(q) \geq 2$ **do**

28: $lp := L(p); lq := L(q)$

29: **while** $L(lp) \neq L(lq)$ **do**

30: $L(lp) := \min(L(lp), L(lq))$

31: $L(lq) := \min(L(lp), L(lq))$

32: **while** $L(p) \neq L(L(p))$ **do**

33: $L(p) := L(L(p))$

34: **until** $\neg change$

For adjacent pixels p, q within a minimal plateau, $lp = L(p), lq = L(q)$ are the representative labels of the corresponding sets. If $lp \neq lq$, then we merge the two sets by setting the smaller label as the parent pointer of the larger label. Since our procedure is designed for parallel implementation, more than two pixel sets can be merged simultaneously. This check and subsequent label update are performed repeatedly until $L(lp)$ equals $L(lq)$. A label path reduction loop is then carried out for all pixels to convert label paths into direct pointers to representative labels. This merging and reduction process continues iteratively until no further label changes occur. Upon completing step IV, each catchment basin will have a single label value, and for each pixel p , $L(p)$ indicates the catchment basin to which p belongs.

3.2. Alternative path-reducing, UNION-FIND watershed

A slightly revised version of the PRUF algorithm, known as APRUF, has been developed for relatively *smaller* images (typically those under 10 megapixels/megavoxels). It alters the implementation of step III by

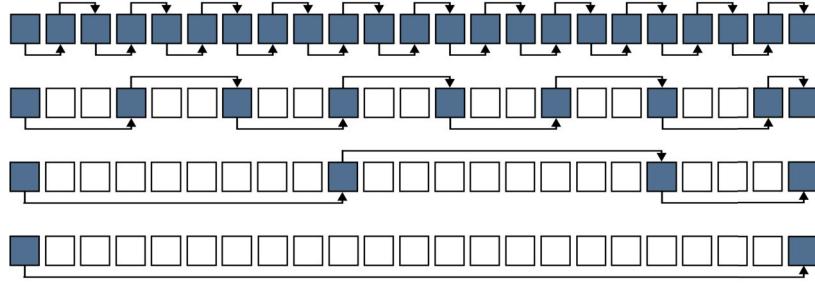


Fig. 4. Length 22 path with $RR = 2$, in only $\lceil \log_3 22 \rceil = 3$ iterations.

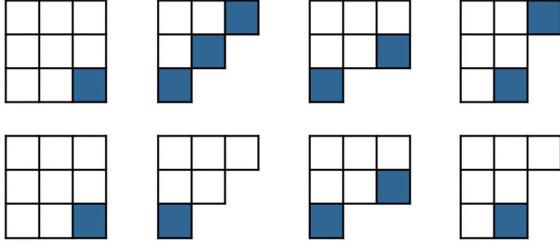


Fig. 5. Minimal plateau shapes with state-3 pixels highlighted in a blue shade. 4-pixel neighbourhood on the left; 8-pixel on the right.

substituting lines 19–23 in Algorithm 1 with lines 28–29. In smaller images, catchment basins are generally smaller, resulting in shorter label paths to minima. While PRUF requires global synchronization to make updated labels visible to other threads, APRUF improves execution time by eliminating this global synchronization: labels are independently acquired by each thread.

3.3. Synchronous implementation for the GPU

Using the CUDA Toolkit by NVIDIA, we have produced three different implementations of our PRUF watershed in Algorithm 1. The initial and the most basic implementation of PRUF is described first; referred to as *synchronous* PRUF ($\text{PRUF}_{\text{sync}}$). The remaining two implementations are discussed in Sec. 3.4. Each of the three algorithm variants assigns a dedicated thread to each pixel in the image.

$\text{PRUF}_{\text{sync}}$ relies on a host code function and five CUDA kernels: one for each step of the process, plus an additional one for step IV. A kernel is a function that runs on numerous threads on the GPU in parallel. The five kernels in $\text{PRUF}_{\text{sync}}$ implement lines 1–10, 12–16, 20–22, 24–27, and 28–29 in Algorithm 1, respectively.

The most expensive step of PRUF (for any implementation), and $\text{PRUF}_{\text{sync}}$ in particular, is the resolution of non-minimal plateaux. Unlike steps I and III–IV, the number of iterations of the code in lines 11–18 in Algorithm 1 may be linear in the image size in the worst case. For instance, in the case of $(2n + 2) \times 1$ example

$$75 \underbrace{89 \ 89 \ 89 \ \dots \ 89 \ 89}_{2n} \ 81,$$

then it requires n iterations to complete step II of $\text{PRUF}_{\text{sync}}$ due to the costly global inter-block synchronization that occurs after each state update. Pixel states are initialised in step I as

$$1 \ 0 \ \underbrace{2 \ 2 \ \dots \ 2}_{2n-2} \ 0 \ 1;$$

after $n - 2$ iterations of step II we have

$$1 \ \underbrace{0 \ 0 \ \dots \ 0}_{n-1} \ 2 \ 2 \ \underbrace{0 \ 0 \ \dots \ 0}_{n-1} \ 1.$$

During the $(n - 1)^{\text{th}}$ iteration, the remaining 2-states are converted to 0, and the final n^{th} iteration, which does not introduce any changes, completes the resolution of non-minimal plateaux.

Moreover, because state update for the plateau pixels in lines 13–16 of Algorithm 1 occurs in parallel and the order of update completion is non-deterministic, we require the buffer state memory S' to ensure correctness. The buffer also ensures a fair distribution of non-minimal plateaux among neighbouring catchment basins. For the given example at $n = 5$, the label values after step II will be

$$1 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \quad 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1,$$

where the numbers represent pixel states and the arrows indicate the labels.

Therefore, the sizes of non-minimal plateaux within an image are anticipated to significantly impact the execution time of parallel watershed processing. Such trends are documented in existing literature [29,20]; our results in Sec. 5 corroborate this assumption.

3.4. Block-asynchronous and balanced versions

Alongside $\text{PRUF}_{\text{sync}}$ we introduce two additional PRUF watershed implementations designed to enhance execution speed. These implementations leverage CUDA's intra-block synchronization mechanisms. Block-level shared memory offers quicker access to data shared among threads, and in-block synchronization is less costly than global synchronization. By utilizing these block-level tools, we aim to minimize the number of global iterations required for resolving non-minimal plateaux.

The *block-asynchronous* PRUF watershed ($\text{PRUF}_{\text{async}}$) is different from $\text{PRUF}_{\text{sync}}$ in the second step of the algorithm. Its step II CUDA kernel uses block-level synchronisation to perform multiple state updates.

The non-minimal plateau resolution kernel of $\text{PRUF}_{\text{async}}$ begins by loading data into shared and local memory. Because each pixel requires access to both the input image and the state values of neighbouring pixels to update its own state, this information is stored in shared memory. Each pixel p is tasked with copying its $I(p)$ and $S(p)$ values into the shared memory. Additionally, pixels located at the block's boundary also copy the values of their neighbours outside the block.

Each thread block operates with its own shared memory space, and there is no communication between different blocks during kernel execution. Data updates from one block become available to threads in other blocks only after being written to global memory and upon completion of kernel execution, which signifies global synchronization points. Consequently, the shared memory values for the additional band of pixels remain unchanged during kernel execution (they are updated only within their respective block) and retain the values from the last global synchronization point. Information is transferred from one block to another once per global iteration.

Our third variant, *balanced* block-asynchronous PRUF (PRUF_{bal}), combines block-level synchronization, similar to $\text{PRUF}_{\text{async}}$, with the fair watershed lines positioning of, as $\text{PRUF}_{\text{sync}}$. We expand the 0–3 state range to include negative numbers: when $S(p) < 0$ then value $|S(p)|$ indicates the distance of a non-minimal plateau pixel from the

plateau boundaries, reflecting the number of state update operations required for information to reach p . Algorithm 3 outlines the non-minimal plateau resolution step (kernel) for PRUF_{bal}.

Algorithm 3 Non-minimal plateau resolution, PRUF_{bal}.

```

1: GlobalToLocal(L) {Copy label from global to local memory}
2: GlobalToShared(I) {Copy image from global to shared}
3: GlobalToShared(S) {Copy state from global to shared}
4: repeat
5:   SyncThreads {Synchronise threads in block}
6:   SharedToLocal(S) {Copy state from shared to local}
7:   if ( $S(p) \geq 2$  and  $\exists q \in N(p), S(q) \leq 0 \wedge I(q) = I(p)$ ) or
     ( $\exists q \in N(p), S(p) + 1 < S(q) \leq 0 \wedge I(q) = I(p)$ ) then
8:      $L(p) := q; S(p) := S(q) - 1$ 
9:   SyncThreads {Synchronise threads in block}
10:  LocalToShared(S) {Copy updated state to shared memory}
11: until  $\neg blockchange$ 
12: LocalToGlobal(L) {Copy updated label to global memory}
13: SharedToGlobal(S') {Copy updated shared state to global}
  
```

After a non-minimal-plateau pixel's state changes from 2 or 3 to a negative number, it can be updated iteratively to larger (but still negative) values during subsequent in-block iterations or following global synchronization points. The resolution step is deemed complete only when no additional improvements can be made to the states of the non-minimal plateau pixels. In the case of the simple 14×1 pixel example

75 89 89 89 89 89 89 89 89 89 89 89 89 81

assuming block size 3, the first global iteration of step II produces states

$\boxed{1} \quad 0 \quad -1 \quad \boxed{2} \quad 2 \quad 2 \quad \boxed{2} \quad 2 \quad 2 \quad \boxed{-3} \quad -2 \quad -1 \quad \boxed{0} \quad 1.$

Following the second global iteration, we get

$\boxed{1} \quad 0 \quad -1 \quad \boxed{-2} \quad -3 \quad -4 \quad \boxed{-6} \quad -5 \quad -4 \quad \boxed{-3} \quad -2 \quad -1 \quad \boxed{0} \quad 1;$

and this is corrected after iteration 3

$\boxed{1} \quad 0 \quad -1 \quad \boxed{-2} \quad -3 \quad -4 \quad \boxed{-5} \quad -5 \quad -4 \quad \boxed{-3} \quad -2 \quad -1 \quad \boxed{0} \quad 1.$

PRUF_{bal} is slightly slower than PRUF_{async} due to (a) additional state updates and (b) extra global iterations of resolution. Overall, PRUF_{bal} shows a large runtime improvement upon PRUF_{sync}, especially on images with large plateaux.

Similar implementation variants can be considered for PRW and APRUF, producing PRW_{sync}, PRW_{async}, PRW_{bal}, APRUF_{sync}, APRUF_{async}, and APRUF_{bal}. The main parameters of the path reducing, UNION–FIND watershed affecting execution times are the thread block size and RR .

3.5. Implementation details

The memory requirements for PRUF_{sync} and PRUF_{async} are similar: per pixel one byte for input image value; one byte for pixel state and another for new state buffer; four bytes for pixel label. That sums up to seven bytes per pixel. In addition, PRUF_{async} also relies on block-level shared memory usage for algorithm step II. This includes input image values, pixel states and a flag for iterations. For a block of size 16×16 , the shared memory per thread block will be $(16 + 2) \times (16 + 2) \times (1 + 1) + 1 = 649$ bytes.

Since PRUF_{bal} uses a larger state space, the requirements for memory are 13 bytes per pixel: one byte for input image value; four bytes for pixel states and four bytes for new state buffer; four bytes for pixel label. Similarly, the shared memory size per 16×16 block is $(16 + 2) \times (16 + 2) \times (1 + 4) + 1 = 1,621$ bytes. For all three variants of the watershed we use only global memory to store information on the device. No texture or constant memory is used. Future experiments with other memory types

intended to fine-tune implementations may slightly decrease execution times.

The CUDA implementations of PRUF_{bal}, APRUF_{bal}, PRW_{bal}, and our waterfall transform based on PRUF_{bal} can be found at <https://github.com/hameemm/PRUF-watershed.git>. The CUDA experimental setup is described in Section 5.

4. GPU waterfall transform

While the watershed transform is popular in many applications, its main drawback of over-segmentation is inherent to this procedure. This over-segmentation is overcome by the waterfall transform (Fig. 1 and Fig. 6). Hierarchical segmentation relies on post-processing of the watershed transform to address the over-segmentation issue by merging non-significant catchment basins. The waterfall transform [33–35] constructs the adjacency graph of the watershed catchment basins and applies the watershed transform on the minimal spanning tree of this graph to achieve a new coarser segmentation of the input image. Recursive waterfall application results in a hierarchical segmentation of the image with coarser layers of fewer segmented regions produced at each iteration.

We propose a novel parallel waterfall reminiscent of the original [33] based on image reconstruction. After the primary watershed is applied on the input image, the image is modified by raising each of the catchment basins to a plateau the height of which equals the lowest pass point on the watershed line surrounding that catchment basin. In other words, the image modification comprises the two steps (kernels): V: identify new minima; VI: update image with new minima. Algorithm 4 provides the details of steps V–VI of the image reconstruction for the waterfall.

If P is still the set of pixel positions in the input image, I is the image value function, and $N(p)$ is the neighbourhood of the position $p \in P$, then for p and $q \in N(p)$ on the watershed line, i.e. $L(p) \neq L(q)$, we define the watershed height as $height(p, q) = \max(I(p), I(q))$.

Since our watershed does not explicitly assign any pixel as belonging to watershed lines and all pixels are put into catchment basins, we can speculate that the watershed lines pass between pixels at the level of the higher pixels. When the pair p and q belong to different catchment basins, then $height(p, q)$ is the local height of the watershed line separating the catchment basins. For a given watershed catchment basin, its lowest pass point is the part of the surrounding watershed line with smallest height. This means that the height of the lowest pass point of a catchment basin equals the minimum among $height(p, q)$, where p belongs to this catchment basin and q does not. In the terrain map abstraction, if we flooded this catchment basin, then it would leak into a neighbouring catchment basin only after reaching the level, or the height, of the lowest pass point. Step V identifies this minimal level for each of the catchment basins by considering the whole watershed line. Before step V the array with the new minimum values ($newmin$) is initialised to a large constant M such that M is at least the upper bound of the greyscale range of the input image.

Algorithm 4 Image reconstruction for parallel waterfall.

```

1: memset(newmin, M, |P|) {Initialise new min to a large constant}
2: for all  $p \in P$  in parallel do {Step V: identify new min}
3:   for all  $q \in N(p), L(q) \neq L(p)$  do
4:      $height(p, q) := \max(I(p), I(q))$ 
5:      $newmin(L(p)) := \min(newmin(L(p)), height(p, q))$ 
6: for all  $p \in P$  in parallel do {Step VI: update image with new min}
7:   if  $I(p) < newmin(L(p))$  then
8:      $I(p) := newmin(L(p))$ 
  
```

Step VI updates the input image based on the new minimum values. Every pixel below the new minimum for its catchment basin is raised to the level of that new minimum, i.e. $I(p) = newmin(L(p))$ if, before step VI, $I(p)$ was smaller than $newmin(L(p))$.

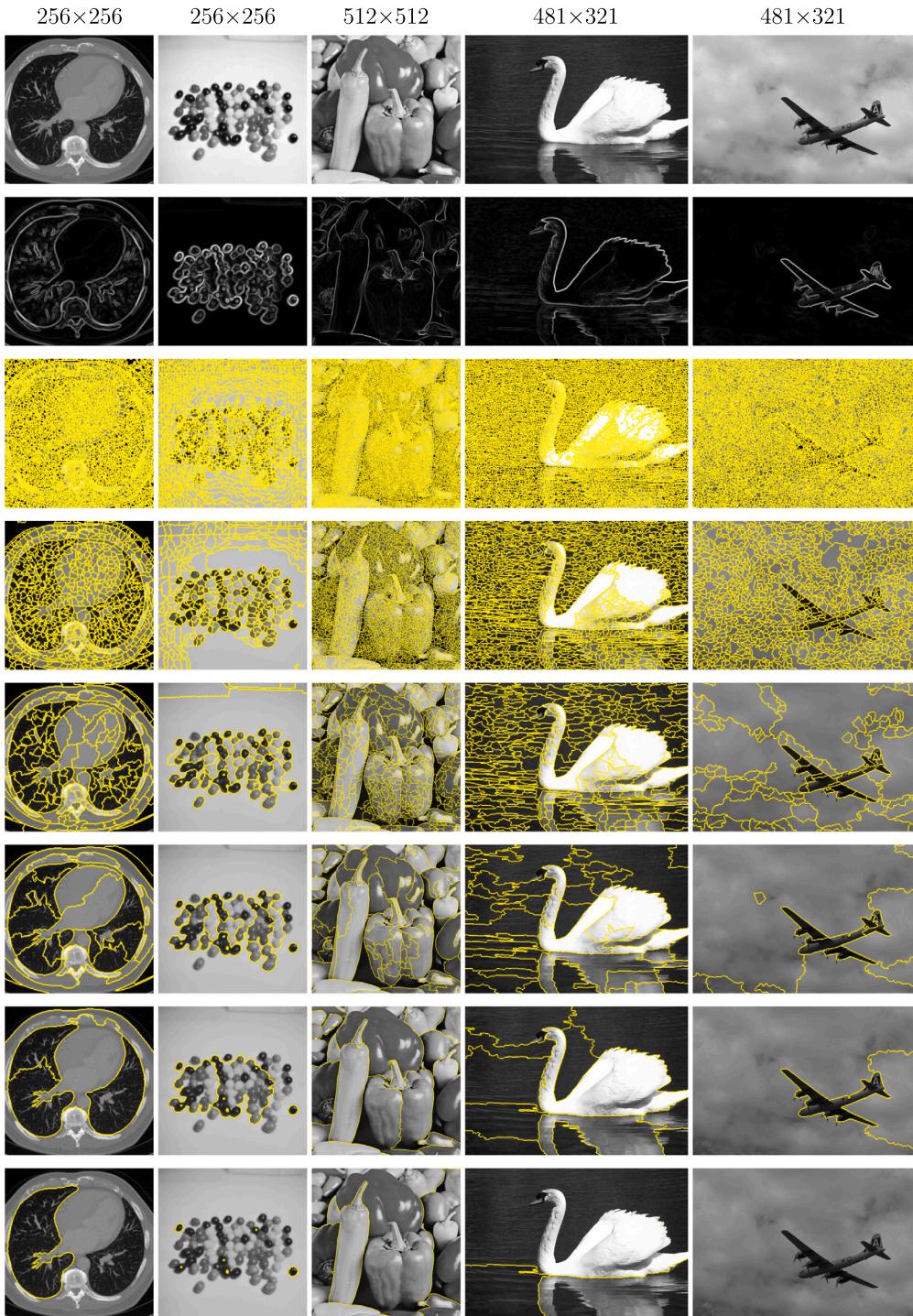


Fig. 6. Hierarchical output on six input images. Top-down: raw greyscale inputs; gradient magnitude images (after Gaussian blurring); originals overlaid with watershed lines in yellow; and five hierarchical waterfall layers, $NL=6$, in the same format.

Fig. 7 illustrates the image reconstruction process for the waterfall. It depicts the watershed of an input 1D image followed by three recursive waterfall applications: the image reconstruction and the watershed. The image boundaries and watershed lines are red. The blue shades correspond to the new minima for each old watershed region. Each shade shows an image reconstruction to update pixel elevations.

If the watershed transform is applied on the reconstructed image, the result will be a set of new catchment basins, where each corresponds to several catchment basins in the primary watershed transform. The image reconstruction and the following watershed transform comprise the waterfall. If applied recursively, this transform gradually reduces

the number of catchment basins, eventually grouping the whole image into a single region (Fig. 7).

Our waterfall implementation relies on PRUF_{bal} for the watershed steps I–IV. The image reconstruction steps V–VI are implemented with a single kernel each. Each pixel position in the image is still assigned to a separate execution thread; the parameters are set as per PRUF_{bal}. The host code of the procedure that applies watershed followed by a fixed number of waterfall applications is presented in Algorithm 5.

After the image is copied into the device and the thread blocks are set up, the hierarchical segmentation is constructed layer by layer. The layer generation starts with a watershed run (lines 4–7). For all layers

Table 1

Different watershed algorithms on 2D images; execution times in ms. Boldface highlight is applied to the best performance; the runner-up is underlined.

GPU	conn	source	C4L Image Dataset [41]						MicroCT	Berkeley [42]
			size	256×256	512×512	1024×1024	1920×1080	2560×1440	2560×1920	
1080 (GPU-a)	4	DW	0.560	1.011	2.442	5.020	7.754	9.968	45.513	0.738
		CA	0.633	1.817	6.161	10.796	18.790	24.948	380.041	1.187
		PRW	0.267	0.630	1.823	3.714	5.735	7.520	31.408	0.386
		APRUF	0.224	0.572	1.910	3.528	5.593	7.379	65.215	0.337
		PRUF	0.249	0.605	1.749	3.630	5.793	7.565	28.987	0.363
1080 (GPU-a)	8	DW	0.542	1.032	2.553	5.449	8.385	10.510	47.025	0.718
		CA	1.300	3.240	13.068	32.184	52.054	98.070	465.830	1.857
		PRW	0.262	0.668	1.875	4.258	6.223	8.158	33.840	0.399
		APRUF	0.221	0.613	1.890	4.096	6.169	8.041	67.851	0.360
		PRUF	0.246	0.642	1.825	4.202	6.361	8.275	31.203	0.387
3050 Ti M (GPU-b)	4	DW	0.626	1.085	2.444	5.115	7.910	10.341	43.497	0.620
		CA	0.459	1.300	4.768	7.769	14.756	18.761	508.295	0.861
		PRW	0.301	0.690	1.812	3.795	5.907	7.776	33.213	0.379
		APRUF	0.264	0.615	1.816	3.657	5.701	7.530	56.532	0.328
		PRUF	0.295	0.651	1.775	3.795	5.993	7.859	30.425	0.358
3050 Ti M (GPU-b)	8	DW	0.470	1.115	2.555	5.562	8.458	10.983	45.985	0.635
		CA	1.188	3.021	16.875	45.579	78.922	148.718	654.449	1.670
		PRW	0.264	0.747	1.861	4.525	6.455	8.569	36.055	0.416
		APRUF	0.212	0.685	1.847	4.395	6.341	8.270	72.390	0.372
		PRUF	0.242	0.700	1.844	4.526	6.614	8.626	34.496	0.401
3060 (GPU-c)	4	DW	1.133	1.592	2.802	4.925	7.451	9.502	34.430	1.310
		CA	0.600	0.997	3.103	4.823	8.431	11.022	267.857	0.763
		PRW	0.531	0.791	1.647	3.039	4.660	6.046	21.659	0.635
		APRUF	0.368	0.605	1.506	2.736	4.213	5.564	35.443	0.466
		PRUF	0.457	0.717	1.517	2.885	4.448	5.794	19.775	0.549
3060 (GPU-c)	8	DW	1.162	1.610	3.013	5.203	7.785	9.462	35.501	1.317
		CA	2.100	2.990	11.010	28.304	42.406	85.045	340.983	1.912
		PRW	0.537	0.843	1.804	3.495	5.036	6.202	23.351	0.668
		APRUF	0.389	0.666	1.628	3.210	4.668	5.748	45.046	0.512
		PRUF	0.470	0.769	1.674	3.359	4.894	5.991	21.820	0.593

Algorithm 5 The host code of the hierarchical segmentation.

```

1: HostToDevice(I) {Copy image from host to device}
2: DimGrid, DimBlock {Set grid and block dimensions}
3: for  $n \in [0, NL - 1]$  do
4:   InitialisationKernel {Step I}
5:   repeat ResolutionKernel {Step II}
6:   repeat ReductionKernel {Step III}
7:   MergingKernel and FullResolutionKernel {Step IV}
8:   if  $n < NL - 1$  then
9:     memset(newmin, M, |P|)
10:    IdentificationKernel {Step V}
11:    UpdateKernel {Step VI}
12: DeviceToHost(L) {Copy labels from device to host}

```

except the last one (if $0 \leq n < NL - 1$) steps V–VI follow (lines 8–11) to prepare for the next watershed application by reconstructing the image with new minima levels. Finally, the new layer of the segmentation is copied from the device to the host memory (line 12).

The memory requirements of the hierarchical segmentation are the same as for PRUF_{bal}: new variables or arrays are *not* used. This is achieved by reusing the available memory: the same I, L, S, S' arrays are used at every application of the watershed. This is allowed because: (a) L, S, S' are initialised from scratch during steps I–II and (b) L values are copied back into a new address in the host memory at each iteration. Furthermore, the S' array is reused under the name *newmin* in the image reconstruction steps V–VI.

The only new parameter that the hierarchical segmentation adds is the number of the waterfall applications. We denote the number of watershed applications (primary or as part of waterfall) NL , indicating the number of segmentation layers generated in the hierarchical segmentation. This parameter does not affect the execution of the individual waterfall applications. For a typical input image, a hierarchy leading to

a single region corresponding to the whole image is usually achieved in under ten waterfall applications: $NL=5$ to $NL=7$ is usually enough to get a semantically meaningful hierarchy of segmentations. This estimate is based empirically on multiple previous experiments in our research group on the application of sequential watershed/waterfall to various data; results in Fig. 1 agree with this.

In terms of dimensionality, this hierarchical segmentation mimics the watershed transform used—PRUF_{bal}, in this case. While a 3D implementation would work for 1D or 2D images, the memory consumption pattern of PRUF_{bal} is optimised in case of separate 2D and 3D implementations.

5. Evaluation and results

Datasets: Two public datasets were used: C4L [41] (greyscale images grouped by resolution), and the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500) [42], converted to greyscale. We also used three 3D images (128 mega-voxels each: 320×320×1250, 500×500×512, 4000×4000×8), and a 3D image of 800 megavoxels: 4000×4000×50, constructed from the same microCT [12]. The 4000×4000×8 image is also interpreted as a set of eight 2D images. For reproducibility, all images are raw (no smoothing).

Experimental Setup: The experiments were carried out on three different machines to include processors and GPUs spanning multiple generations and compute capabilities: (GPU-a) Intel Core i7-8700, NVIDIA GeForce GTX 1080, CUDA 11.6, (GPU-b) Intel Core i9-12900HK, NVIDIA GeForce RTX 3050 Ti Mobile, CUDA 11.8, (GPU-c) AMD Ryzen 7 3700X, NVIDIA GeForce RTX 3060, CUDA 11.7. Every reported watershed execution time is a *minimum* of ten runs per image (for reproducibility and to mitigate delays produced by other running processes), and an *average* across multiple images in the same dataset (where applicable). To confirm the equivalence of different watershed

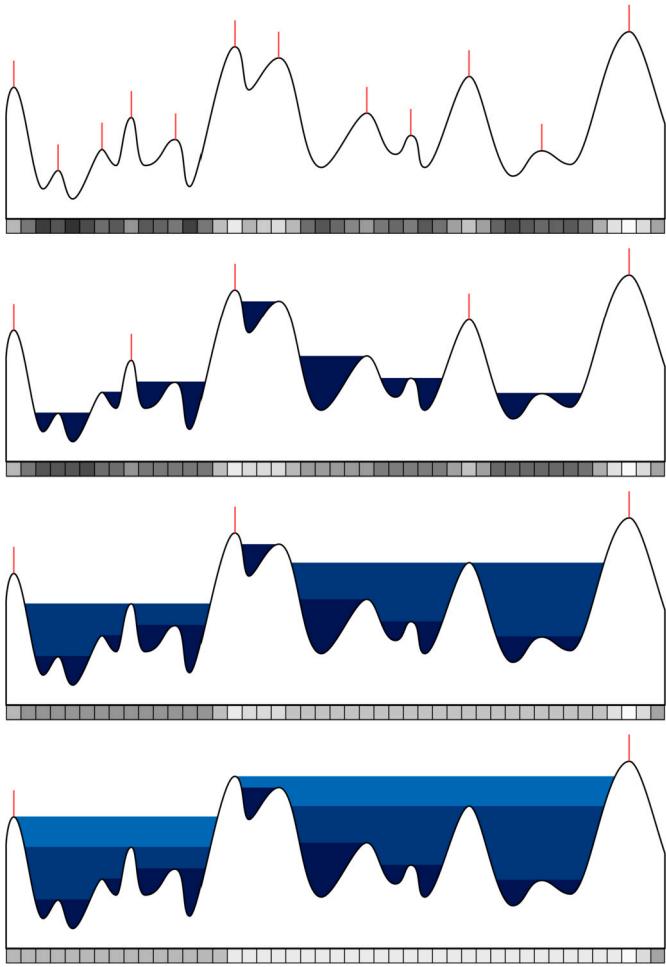


Fig. 7. Watershed transform of 1D image and three consecutive applications of waterfall. Curve elevation at each pixel corresponds to its grey value; shades of blue show updated pixel elevations after each image reconstruction; image boundaries and watershed lines are in red.

procedures for a given image, we verify that they consistently generate the same number of regions. Although there are more powerful GPUs available in advanced supercomputer configurations, our selection demonstrates that the proposed approach is already feasible with standard computing hardware.

The three algorithms use two types of parameters: the block size for CUDA and the reduction rate RR . The latter is only used in PRUF (Algorithm 1), and PRW. Only execution time is affected by these parameters; otherwise, the results of different setups are equivalent. We have (partly) reported in [12,43] that CUDA block sizes of 16×16 in 2D and $8 \times 8 \times 8$ in 3D, and $RR=6$ lead to fastest execution.

5.1. Watershed results in 2D and 3D

Table 1 shows the comparison among PRUF, APRUF, PRW, CA [22, 29] and DW [26,28] algorithms on 2D data. For all five algorithms 8-pixel and 4-pixel neighbourhoods were considered. For both C4L and BSDS500, APRUF (4-pixel on GPU-c) is the fastest: 0.37 ms for the smallest image group (256×256) and 5.56 ms for the largest image group (2560×1920). APRUF's improvement over DW is between 41.44% and 67.52%. CA is on a par with DW or is slower across the different setups, only occasionally running faster than DW. All three of our algorithms are consistently faster than CA.

PRUF (fastest) shows a 44.21% improvement over APRUF (slowest) for the 4000×4000 images, indicating effectiveness of the reduction rate on larger images (detailed analysis in [13]). The comparison of the exe-

cution times among the 8-neighbour implementations of the algorithms shows a pattern similar to the case of 4-neighbour implementations discussed above.

The average execution times of PRW, PRUF, APRUF, and CA (with 6-neighbourhood only; 26-neighbourhood version not available) on 3D test images are reported in Table 2. (We were unable to find 3D implementations of DW.) The $4000 \times 4000 \times 50$ image contains 8×10^8 voxels, which is too large for GTX 1080 and RTX 3050 Ti Mobile GPUs.

Across all large 3D images (1.28×10^8 and 8×10^8 voxels) PRUF and PRW algorithms outperform APRUF, proving the effectiveness of the reduction rate on larger images. PRUF with 6-voxel connectivity (on GPU-c) outperforms PRW across all datasets, due to the parallel UNION–FIND implementation. In the best case ($500 \times 500 \times 512$) PRUF improves 30.77% over APRUF. Like in 2D, for small 3D images ($256 \times 256 \times 128$) APRUF outperforms the others. All three of our algorithms are consistently faster than CA—the difference reaching 2–3 orders of magnitude for the largest image volumes.

Across the 128-megavoxel images, the noticeable difference in execution times shows a correlation of the number of slices in the image and the execution time. Algorithms show longer execution times on images with more slices and smaller intra-slice resolutions due to the weaker locality.

5.2. Effect of connectivity choice

Table 3 analyses the number of catchment basins generated by 4- vs 8-pixel neighbourhood on different resolutions (128^2 to 4096^2) of the same image [44]. In 2D, 4-pixel neighbourhoods produce $1.8(\pm 0.1)$ times more catchment basins than 8-pixel. The relative number of regions generated by each neighbourhood style is independent of the image resolution.

Fig. 8 shows the qualitative comparison between 4- and 8-pixel neighbourhoods on the waterfall layer 2. The 4-pixel neighbourhood tends to capture small details in the image (region A) but also sometimes over-segments regions (region C). Region B illustrates the difference in the number of catchment basins: 8-pixel version generates larger regions, hence it is slower (Fig. 9).

5.3. Waterfall results

Fig. 1 illustrates the results of hierarchical segmentation with $NL=6$ on two 2D images: cameraman and knee MRI slice. The watershed (layer 0) and the waterfall transforms are applied on the gradient magnitude images of the smoothed input. The waterfall layers (1–5) demonstrate how smaller regions get gradually grouped together in higher layers of the segmentation. In each layer the remaining watershed lines trace the high-contrast edges in the image, thus moulding themselves around actual object boundaries. For instance, in segmentation layer 5 of the knee MRI slice, the femur is captured with a single waterfall region. For both images, segmentation layer 6 (not shown) contains only a single region corresponding to the whole image.

Table 4 presents numbers of regions in different layers of the hierarchical segmentation on variants of the same image [44]. Fig. 10 compares layer generation runtimes using PRUF on [44] with different image resolution (128^2 to 2048^2). Each measured layer generation time includes watershed with steps I–IV, image reconstruction with steps V–VI (except layer 7), and transfer of segmentation labels from device to host. The layer generation times increase with the layer number. The main runtime difference occurs in the duration of step II, which is due to larger non-minimal plateaux appearing after each image reconstruction step.

The comparison of average execution times (in ms) for the different waterfall algorithms for the C4L 1024×1024 test images is presented in Fig. 11. As discussed, APRUF watershed (layer 0) is the fastest when it comes to smaller images. However, as PRUF is more efficient at handling

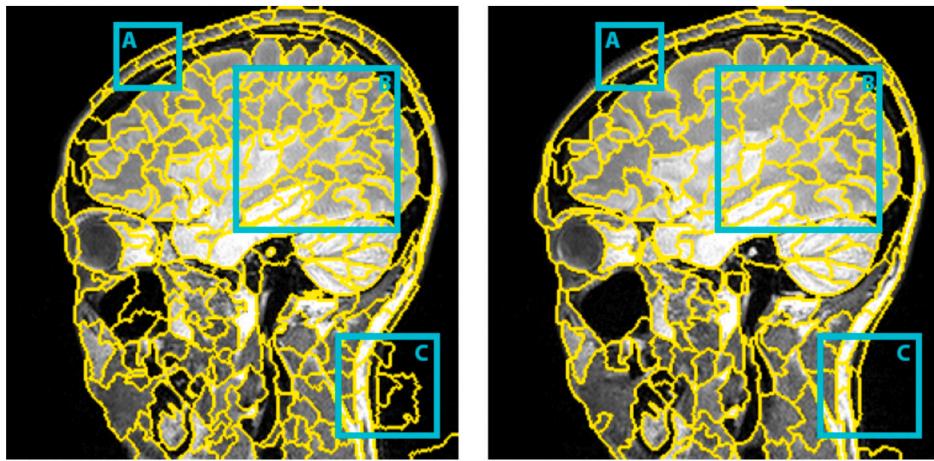


Fig. 8. 4- (left) vs 8-pixel (right) neighbourhood results on brain MRI, depicted as overlays with yellow waterfall layer 2 lines.

Table 2

Execution times (in ms) of the different watershed algorithms on 3D images. The best performance is in boldface; the runner-up is underlined.

GPU	conn	source	C4L	MicroCT	MicroCT
		# voxels	8.39×10^6	1.28×10^8	8×10^8
		volume	$256 \times 256 \times 128$	$320 \times 320 \times 1250$	$500 \times 500 \times 512$
1080 (GPU-a)	6	CA	74.345	9,808.556	24,876.996
		PRW	17.061	<u>359.172</u>	<u>303.864</u>
		APRUF	15.617	582.493	435.270
		PRUF	<u>16.121</u>	317.906	286.303
	26	PRW	35.299	<u>853.416</u>	695.573
		APRUF	33.613	935.466	<u>754.057</u>
		PRUF	<u>34.078</u>	784.110	634.653
		CA	78.495	16,346.377	10,728.934
	3050 Ti M (GPU-b)	PRW	15.514	<u>352.567</u>	<u>336.017</u>
		APRUF	14.325	629.491	440.831
		PRUF	<u>15.486</u>	323.475	319.391
		CA	44.144	7,265.816	4,793.789
3060 (GPU-c)	6	PRW	12.146	<u>285.549</u>	<u>222.612</u>
		APRUF	10.944	471.418	300.419
		PRUF	<u>11.379</u>	270.231	207.990
		CA	21.283	<u>565.747</u>	460.407
	26	PRW	20.151	687.934	528.559
		APRUF	<u>20.549</u>	559.511	453.554
		CA	44.144	7,265.816	4,793.789
		PRW	12.146	<u>285.549</u>	<u>222.612</u>

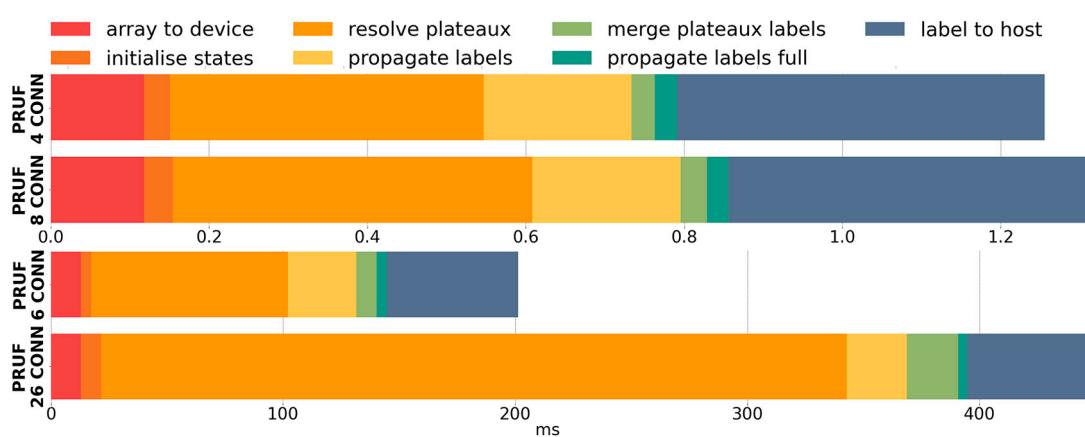


Fig. 9. Composition of PRUF execution times for 2D and 3D Moore and von Neumann neighbourhoods: data copy, initialise states (I), resolve plateaux (II), propagate labels (III), merge plateaux and propagate labels (IV), data copy back. Measured on GPU-c using 1024×1024 (top) and 500×500×512 (bottom) test images.

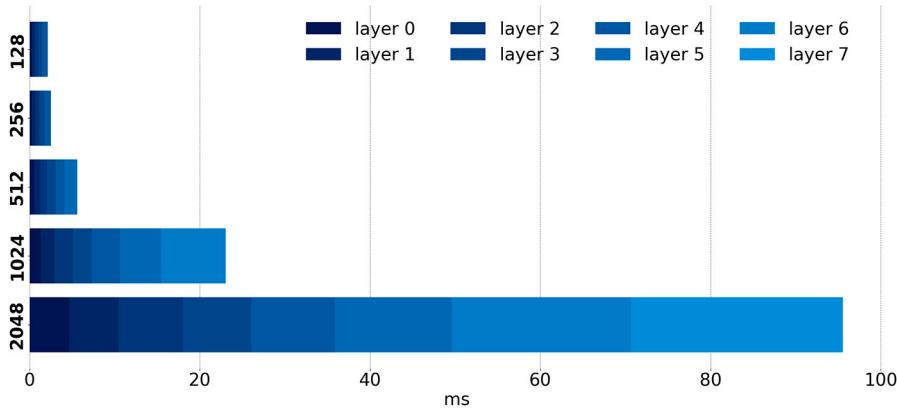


Fig. 10. Execution time (in ms) of each layer of waterfall (PRUF) per image resolution (128^2 to 2048^2), on the sample image [44] on GPU-c.

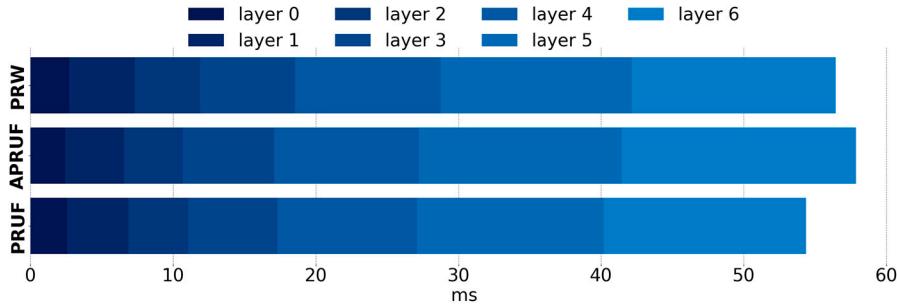


Fig. 11. Comparison of mean execution times (in ms) of the waterfall versions: PRW, APRUF, PRUF on C4L 1024×1024 images on GPU-c.

Table 3

Numbers of catchment basins for 4- and 8-pixel neighbourhoods for image resolutions 128^2 to 4096^2 , on [44].

conn	128	256	512	1024	2048	4096
4	1,748	6,523	24,217	98,107	273,728	413,901
8	926	3,458	13,196	52,191	165,869	241,572

Table 4

Numbers of catchment basins in different waterfall layers (PRUF) in terms of image resolution (128^2 to 4096^2), on [44].

layer	128	256	512	1024	2048	4096
0	1,748	6,523	24,217	98,107	273,728	413,901
1	277	1,089	3,703	13,808	41,233	56,164
2	44	194	578	2,158	6,230	8,652
3	10	34	94	320	952	1,334
4	2	6	16	50	138	202
5	1	1	4	10	21	29
6	1	1	1	3	5	5
7	1	1	1	1	2	2
8	1	1	1	1	1	1

label propagation for larger image regions, its execution becomes faster on further layers of waterfall.

6. Improving semantic segmentation using APRUF

The rather sterile analysis in Sec. 5 illustrates performance on isolated examples. Modern-day computing tasks no longer require partitioning isolated images deterministically as a stand-alone application. Rather, real-life modern Computer Vision use of the waterfall algorithm is envisaged in circumstances where groups of pixels can be decided upon in one go, thereby speeding up the time taken by a step in a larger task (such as training a DL model). For this reason, the obvious choice of comparison is with superpixel-based algorithms [2]. A superpixel is defined as a perceptually homogeneous region in the image. Usage of superpixel regions dramatically reduces the number of image primitives, thus improving the computational efficiency of a DL network.

With the inherent regular shape and the grid-like distribution of segments, conventional superpixel algorithms ignore meaningful image edges by placing them inside a superpixel (Fig. 12). Since partitioning serves as a basis for later-stage processing, obtaining effective output is important, making watershed the preferred choice.

In order to show the versatility and applicability of these results, we compare the waterfall against superpixel using two fundamentally different DL tasks: hyperspectral image (HSI) *classification* and noisy label *semantic segmentation*. In each of these experiments we have retained the same setup and hyper-parameters as set out in each original article proposing the method.

Classification. CEGCN [45] is a heterogeneous deep network that combines convolutional neural network (CNN) with graph convolutional network (GCN) for HSI classification. The CNN module is responsible for feature learning on small-scale regular regions, whereas the GCN, given the superpixel graph of the image, captures information from a larger-scale irregular structure.

In the setting of CEGCN, reproducing the original setup from [45], we compare waterfall against existing superpixel algorithms: SLIC [46], quick shift (QS) [47], Felzenszwalb and Huttenlocher's (FH) method [48], linear spectral clustering (LSC) [1], and SEEDS [49]. The comparison is conducted on three different HS images: Indian Pines, University of Pavia, and Salinas [45]. Each experiment is repeated ten times, reporting overall accuracy (OA), average accuracy (AA), and kappa coefficient (κ) and the training time. All the superpixel segmentation methods use the hyperparameters provided in [45]. The waterfall algorithm (4-connectivity with $NL=4$ for Indian Pines, University of Pavia datasets and 8-connectivity with $NL=3$ for Salinas dataset) is applied on the gradient magnitude of the compressed images after applying Gaussian blurring. The compressed image of the HSI is constructed by averaging values of all spectral bands. As proposed in [45], superpixel algorithms are executed on the HSI prepossessed using linear discriminant analysis (LDA). We found that skipping the preprocessing steps improved the results of the waterfall. Table 5 reports that the water-

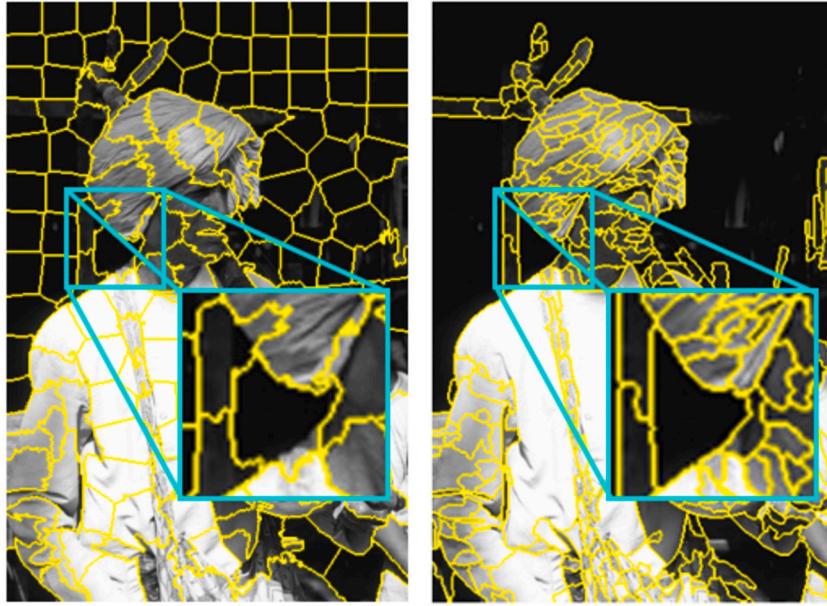


Fig. 12. Edge preservation in SLIC (left) and waterfall layer 2 (right).

Table 5

Comparison of CEGCN with different superpixel segmentation methods vs waterfall method.

Dataset	Metric	SLIC	FH	LSC	QS	SEEDS	Waterfall
Indian Pines	OA (%)	98.826 \pm 0.006	99.028 \pm 0.002	99.096 \pm 0.002	99.058 \pm 0.002	99.147 \pm 0.001	99.173\pm0.001
	AA (%)	96.856 \pm 0.026	97.021 \pm 0.012	97.106 \pm 0.008	97.238 \pm 0.008	98.423\pm0.005	97.881 \pm 0.008
	κ ($\times 100$)	98.662 \pm 0.007	98.891 \pm 0.002	98.969 \pm 0.002	98.927 \pm 0.002	99.027 \pm 0.001	99.057\pm0.001
	time (s)	7.866	7.904	7.995	8.213	7.776	7.744
University of Pavia	OA (%)	99.582 \pm 0.001	99.508 \pm 0.001	99.432 \pm 0.002	99.379 \pm 0.002	99.546 \pm 0.001	99.596\pm0.001
	AA (%)	99.371 \pm 0.002	99.241 \pm 0.002	99.060 \pm 0.003	99.026 \pm 0.003	99.343 \pm 0.002	99.387\pm0.003
	κ ($\times 100$)	99.447 \pm 0.001	99.347 \pm 0.001	99.247 \pm 0.003	99.177 \pm 0.003	99.398 \pm 0.002	99.464\pm0.002
	time (s)	104.038	83.181	89.200	81.097	87.892	83.477
Salinas	OA (%)	99.412 \pm 0.002	99.367 \pm 0.003	99.474 \pm 0.002	99.787 \pm 0.001	99.538 \pm 0.002	99.838\pm0.001
	AA (%)	99.490 \pm 0.001	99.505 \pm 0.001	99.529 \pm 0.002	99.732 \pm 0.001	99.596 \pm 0.001	99.772\pm0.001
	κ ($\times 100$)	99.345 \pm 0.002	99.295 \pm 0.004	99.414 \pm 0.002	99.762 \pm 0.001	99.486 \pm 0.002	99.820\pm0.001
	time (s)	43.851	44.850	45.254	44.556	40.501	35.268

Table 6

SLIC superpixel semantic segmentation [2] vs waterfall-based semantic segmentation on GPU-c.

Method	metric	ISIC skin	JSRT lung	JSRT heart	JSRT clavicle
SLIC	DSC (%)	84.79\pm0.167	96.10 \pm 0.036	90.87 \pm 0.033	85.98\pm0.049
	time	5 h 43 m 48 s	10 h 41 m 46 s	10 h 21 m 47 s	6 h 8 m 5 s
Waterfall	DSC (%)	83.07 \pm 0.166	96.41\pm0.033	91.61\pm0.044	85.86 \pm 0.049
	time	10 h 49 m 8 s	3 h 14 m 13 s	3 h 11 m 55 s	1 h 34 m 15 s

fall outperforms the other algorithms. The waterfall regions tend to be larger, thus they help speed up the training without loss of accuracy.

Segmentation. A further set of experiments is conducted on the task of medical image segmentation from noisy labels [2]. The authors propose a robust iterative learning strategy that combines noise-aware training and noisy label refinement, all guided by superpixels. The design of the network mitigates the impact of noisy training labels by exploiting structural information. The model is built upon the assumption that pixels belonging to the same superpixel share similar ground truth labels. We believe that better object boundary preservation makes waterfall superior compared to superpixels.

We compare waterfall to SLIC on the ISIC (skin) and JSRT (lung, heart, clavicle) datasets (cropped as per [2]), with the mild label noise setting (noise ratio $\alpha=0.3$, noise level $\beta=0.5$). We evaluate the predicted masks with Dice similarity coefficient (DSC) [50] averaged over the last 10 epochs (out of 200). Hyperparameters of SLIC are as described in [2]. Waterfall is applied on the gradient magnitude of the images with: 4-

connectivity for ISIC, 8-connectivity for JSRT clavicle and heart; $NL=4$ for JSRT and $NL=2$ for ISIC. Training times and DSC scores are reported in Table 6.

Waterfall shows DSC results comparable to SLIC, outperforming it on JSRT lung and heart datasets. Waterfall under-performance on the ISIC skin and JSRT clavicle data is due to the nature of the images/labels. Images in the ISIC skin dataset do not have clearly defined edges, making it more suitable for grid-like segmentation. A similar issue occurs on the JSRT clavicle dataset, where the given label does not have a clear edge separation on the images of the dataset. Waterfall, due to a smaller-sized segmentation graph, shows a significant improvement in the training time of the network, outperforming the SLIC model 3.9 times on the JSRT clavicle dataset. Note that for the ISIC skin dataset, with some loss of accuracy ($75.41 \pm 0.263\%$), the training time can be decreased by around 5 hours by increasing $NL=4$.

7. Conclusion

Through systematic experiments on several datasets of images of varying dimensions, we compared the relative pros and cons of the three path-reducing (and UNION-FIND-based) algorithms that we proposed, and showed a consistent speedup in comparison to the previous state of the art. In particular, APRUF offered fastest execution when applied on smaller images (typically up to 10 million pixels/voxels). On larger images, PRUF was usually the fastest, occasionally losing to PRW. For 3D images of 800 megavoxels, using PRUF algorithm, we achieved execution times of approximately 1.4 seconds. All experiments were conducted on three different GPUs to ensure the stability and generalizability of the reported results. More modern GPU nodes would no doubt offer much improved performance, warranting the inclusion of the proposed methods in Computer Vision and Image Processing libraries.

Our extended experiments involving DL applications were selected for their use of edge preservation and fast training. We showed the robustness and superiority of the waterfall over superpixels.

To mitigate over-segmentation, we proposed additional parallel steps that render our algorithms applicable iteratively as waterfall transforms, producing hierarchical segmentations of the image with fewer and larger regions. We presented practical applications of our waterfall algorithm as a replacement for superpixel algorithms in DL pipelines, demonstrating a consistent training time alongside accuracy improvement. In scenarios where millions of image patches are being split thousands of times during training, the proposed time saving will have a significant overall effect.

CRediT authorship contribution statement

Varduhı Yeghiazaryan: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Yeva Gabrielyan:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Irina Voiculescu:** Writing – review & editing, Supervision, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at <https://doi.org/10.1016/j.jpdc.2025.105140>.

Data availability

For reproducibility reasons, most of the data used is publicly available. We have shared links to the data, and a repository with our code.

References

- [1] Z. Li, et al., Superpixel segmentation using linear spectral clustering, in: Proc. CVPR, IEEE, 2015, pp. 1356–1363.
- [2] S. Li, et al., Superpixel-guided iterative learning from noisy labels for medical image segmentation, in: Proc. MICCAI, Springer, 2021, pp. 525–535.
- [3] G.M. Tarhini, et al., Detection of brain tumor in MRI images using watershed and threshold-based segmentation, Int. J. Signal Process. Syst. 8 (1) (2020) 19–25.
- [4] A.C.B. Monteiro, et al., Detecting and counting of blood cells using watershed transform: an improved methodology, in: Proc. BTSym, Springer, 2019, pp. 301–310.
- [5] H. Xu, et al., Segmentation of bone vessels in 3d micro-ct images using the monogenic signal phase and watershed, in: Proc. ISBI, IEEE, 2020, pp. 741–744.
- [6] A. Tareef, et al., Multi-pass fast watershed for accurate segmentation of overlapping cervical cells, IEEE Trans. Med. Imaging 37 (9) (2018) 2044–2059.
- [7] A. Das, et al., Deep learning based liver cancer detection using watershed transform and Gaussian mixture model techniques, Cogn. Syst. Res. 54 (2019) 165–175.
- [8] M. Elbatel, et al., Seamless iterative semi-supervised correction of imperfect labels in microscopy images, in: Proc. MICCAI DART Wks, Springer, 2022, pp. 98–107.
- [9] D. Eschweiler, et al., CNN-based preprocessing to optimize watershed-based cell segmentation in 3D confocal microscopy images, in: Proc. ISBI, IEEE, 2019, pp. 223–227.
- [10] F. Lux, et al., DIC image segmentation of dense cell populations by combining deep learning and watershed, in: Proc. ISBI, IEEE, 2019, pp. 236–239.
- [11] J. Jiang, et al., Accurate 3D cell segmentation using deep features and CRF refinement, in: Proc. ICIP, IEEE, 2019, pp. 1555–1559.
- [12] V. Yeghiazaryan, et al., Path reducing watershed for the GPU, in: Proc. WACV, IEEE, 2018, pp. 577–585.
- [13] Y. Gabrielyan, et al., Parallel partitioning: path reducing and union–find based watershed for the GPU, in: Proc. ICIP, IEEE, 2022, pp. 1501–1505.
- [14] J.B. Roerdink, et al., The watershed transform: definitions, algorithms and parallelization strategies, Fundam. Inform. 41 (2000) 187–228.
- [15] M. Świercz, et al., Fast, parallel watershed algorithm based on path tracing, in: Proc. ICCVG, Springer, 2010, pp. 317–324.
- [16] S. Wu, et al., Parallelization research on watershed algorithm, in: Proc. ACAI, IET, 2012, pp. 1524–1527.
- [17] B. Wagner, et al., Parallel volume image segmentation with watershed transformation, in: Proc. SCIA, 2009, pp. 420–429.
- [18] J. Van Neerbos, et al., Towards a parallel topological watershed: first results, in: Proc. ISMM, Springer, 2011, pp. 248–259.
- [19] C. Kauffmann, et al., Cellular automaton for ultra-fast watershed transform on GPU, in: Proc. ICPR, 2008, pp. 1–4.
- [20] A. Körbes, et al., Advances on watershed processing on GPU architecture, in: Proc. ISMM, Springer, 2011, pp. 260–271.
- [21] M. Hučko, et al., Streamed watershed transform on GPU for processing of large volume data, in: Proc. SCCG, 2012, pp. 137–141.
- [22] P. Quesada-Barriuso, et al., Efficient GPU asynchronous implementation of a watershed based on cellular automata, in: Proc. ISPDPA, 2012, pp. 79–86.
- [23] L. Pan, et al., Implementation of medical image segmentation in cuda, in: Proc. ITAB, IEEE, 2008, pp. 82–85.
- [24] C. Kauffmann, et al., Seeded ND medical image segmentation by cellular automaton on GPU, Int. J. Comput. Assisted Radiol. Surg. 5 (2010) 251–262.
- [25] B. Wagner, et al., A parallel watershed-transformation algorithm for the GPU, in: Workshop on Applications of Discrete Geometry and Mathematical Morphology, 2010, pp. 111–115.
- [26] G.B. Vitor, et al., Fast image segmentation by watershed transform on graphical hardware, in: Proc. CILAMCE, vol. 1, 2009, pp. 1–14.
- [27] A. Körbes, et al., Analysis of the watershed algorithms based on the breadth-first and depth-first exploring methods, in: Proc. SIBGRAPI, IEEE, 2009, pp. 133–140.
- [28] G.B. Vitor, et al., Analysis of a step-based watershed algorithm using cuda, in: Nature-Inspired Computing Design, Development, and Applications, IGI Global, 2012, pp. 321–335.
- [29] P. Quesada-Barriuso, et al., Efficient 2D and 3D watershed on graphics processing unit, Comput. Electr. Eng. 39 (8) (2013) 2638–2655.
- [30] DW, adessowiki.fee.unicamp.br/adesso/wiki/watershed/ismm2011_dw/view/. (Accessed 9 April 2017).
- [31] S. Derivaux, et al., Supervised image segmentation using watershed transform, fuzzy classification and evolutionary computation, Pattern Recognit. Lett. 31 (15) (2010) 2364–2374.
- [32] S. Beucher, Segmentation d’images et morphologie mathématique, Ph.D. thesis, Ecole Nationale Supérieure des Mines de Paris, 1990.
- [33] S. Beucher, Watershed, hierarchical segmentation and waterfall algorithm, in: Proc. ISMM, Kluwer Ac. Publ., 1994, pp. 69–76.
- [34] B. Marcotegui, et al., Fast implementation of waterfall based on graphs, in: Proc. ISMM, Springer, 2005, pp. 177–186.
- [35] S.M. Golodetz, et al., Two tree-based methods for the waterfall, Pattern Recognit. 47 (10) (2014) 3276–3292.
- [36] S. Beucher, et al., P algorithm, a dramatic enhancement of the waterfall transformation, <https://minesparis-psl.hal.science/hal-00835019/document>, 2009.
- [37] S. Beucher, Towards a unification of waterfalls, standard and P algorithms, https://hal.science/hal-01403951v1/file/Unified_Segmentation.pdf, 2012.
- [38] S.M. Golodetz, Zipping and unzipping: the use of image partition forests in the analysis of abdominal CT scans, DPhil. Thesis, University of Oxford, 2010.
- [39] S. Golodetz, et al., Simpler editing of graph-based segmentation hierarchies using zipping algorithms, Pattern Recognit. 70 (2017) 44–59.
- [40] S. Allegretti, et al., Optimized block-based algorithms to label connected components on GPUs, IEEE Trans. Parallel Distrib. Syst. 31 (2) (2020) 423–438.
- [41] J. Bush, C4L image dataset, <https://dx.doi.org/10.21227/bc9m-f507>, 2021. (Accessed 2 January 2022).
- [42] P. Arbeláez, et al., Contour detection and hierarchical image segmentation, IEEE Trans. Pattern Anal. Mach. Intell. 33 (5) (2011) 898–916.
- [43] V. Yeghiazaryan, Parallel front propagation in medical image segmentation, DPhil. Thesis, University of Oxford, 2018.

- [44] Nighttime swirls, <https://earthobservatory.nasa.gov/images/145471/nighttime-swirls>. (Accessed 2 November 2022).
- [45] Q. Liu, et al., CNN-enhanced graph convolutional network with pixel- and superpixel-level feature fusion for hyperspectral image classification, *IEEE Trans. Geosci. Remote Sens.* 59 (10) (2020) 8657–8671.
- [46] R. Achanta, et al., SLIC superpixels compared to state-of-the-art superpixel methods, *IEEE Trans. Pattern Anal. Mach. Intell.* 34 (11) (2012) 2274–2282.
- [47] A. Vedaldi, et al., Quick shift and kernel methods for mode seeking, in: *Proc. ECCV*, Springer, 2008, pp. 705–718.
- [48] P.F. Felzenszwalb, et al., Efficient graph-based image segmentation, *Int. J. Comput. Vis.* 59 (2) (2004) 167–181.
- [49] M. Van den Bergh, et al., Seeds: superpixels extracted via energy-driven sampling, in: *Proc. ECCV*, Springer, 2012, pp. 13–26.
- [50] V. Yeghiazaryan, et al., Family of boundary overlap metrics for the evaluation of medical image segmentation, *J. Med. Imag.* 5 (1) (2018) 015006.

Varduh Yeghiazaryan is an associate professor at the Akian College of Science and Engineering, American University of Armenia. She received her PhD from Oxford Uni-

versity, United Kingdom, for research in medical image analysis. Her research interests include medical image analysis, parallel computation on GPUs, deep reinforcement learning, computation on graphs.

Yeva Gabrielyan is a research associate at the Akian College of Science and Engineering, American University of Armenia, and a senior data scientist at Denovo Sciences. They received their MS degree in Computer and Information Science from the American University of Armenia. Their research interests include deep reinforcement learning, medical image segmentation and AI in drug discovery.

Irina Voiculescu is an associate professor at Oxford University Department of Computer Science. She received her PhD from the University of Bath, United Kingdom, for research in constructive solid geometry. Apart from her main interest in medical imaging (MRI and CT scan analysis), she has also been conducting research in areas such as molecular modelling (protein docking) and polynomial root finding (interval arithmetic).