# Path Reducing Watershed for the GPU

Anonymous WACV submission

Paper ID ****

## Abstract

*The watershed transform is a popular image segmentation procedure from mathematical morphology used in many applications of computer vision. This paper proposes a novel parallel watershed procedure designed for GPU implementation. Our algorithm constructs paths of steepest descent and reduces these paths into direct pointers to catchment basin minima in logarithmic time, also crucially incorporating successful resolution of plateaux. Three implementation variants and their parameters are analysed through experiments on 2D and 3D images; a comparison against the state-of-the-art shows a runtime improvement of around 30%. For 3D images of 128 megavoxels execution times of approximately 1.5–2 seconds are achieved.*

## 1. Introduction

Sequential watershed algorithms have been used for years [14] in many computer vision applications. The watershed transform interprets input greyscale images as topographic terrain maps where the greyscale value of a pixel is used as an altitude value. The image gets divided into regions around local minima, where a downhill path exists between every pixel in a segmented region and its region minimum.

Modern segmentation applications usually involve large amounts of data. In fields like medical image analysis and hyperspectral image analysis 3D images of hundreds of millions of voxels are not rare. Conventional sequential algorithms prove very slow for these. In recent years, with the advent of affordable programmable graphics processing unit (GPU) devices, other parallel watershed algorithms have been designed specifically for GPU implementation. Pioneering papers like [11, 7, 8, 18] mainly developed flooding or immersion watershed procedures where regions are initialised in local minima and expanded to cover the whole image meeting at the watersheds.

More recent work suggests hybrid [15, 10] or fully parallel watershed algorithms by topographical distance [16, 17, 9, 12, 13]. For each pixel in the image, a path of steepest descent is constructed and can be followed down to the nearest minimum. Labels of minima are then propagated uphill, grouping pixels with the same label into a region around the minimum with that label.

We introduce a novel parallel watershed procedure designed for GPU implementation. This algorithm constructs paths of steepest descent for all image pixels and reduces these paths into direct pointers to catchment basin minima in logarithmic time. Our watershed simplifies label initialisation in minimal plateaux and converts label propagation into fast path reduction. Other watersheds usually expect pre-processed input in the form of a lower-complete image in order to resolve non-minimal plateaux. By contrast, our proposed algorithm deals successfully with non-minimal plateaux in the original input.

We propose three different GPU implementation variants of our parallel watershed. Our algorithm is easy to understand and implement, and it outperforms existing GPU watersheds. We thoroughly discuss the merits of our watershed for images of different sizes and textures.

## 2. Path reducing parallel watershed

### 2.1. Path reducing watershed

The suggested parallel watershed procedure consists of four major steps: (S1) initialisation of pixel states and labels, (S2) resolution of non-minimal plateaux, (S3) label propagation, or path reduction, and (S4) merging of minimal plateau labels.

This procedure can be classified as a watershed transform based on topographic distance: it relies on paths of steepest descent. An alternative view on the four steps is: (S1) initialisation of paths of steepest descent; (S2) resolution of paths on non-minimal plateaux; (S3) path reduction into pointers to local minima; (S4) addition of extra paths inside minimal plateaux to achieve single pointer target (i.e. label) per catchment basin. Algorithm 1 provides details of the four steps of the path reducing watershed.

If $I$ is the set of all pixel positions in the image then for every $p \in I$ we can consider the local neighbourhood $N(p)$—the set of positions that are a unit distance away

**Algorithm 1** Path reducing parallel watershed

**for all** $p \in I$ in parallel **do** ▷ Step S1: initialisation of labels & states
  $q \leftarrow max\{s \in N(p) | \forall t \in N(p), img(s) \leq img(t)\}$
  **if** $img(q) < img(p)$ **then**
    $label(p) \leftarrow q$; $state(p) \leftarrow 0$
  **else if** $img(q) > img(p)$ **then**
    $label(p) \leftarrow p$; $state(p) \leftarrow 1$
  **else if** $q > p$ **then**
    $label(p) \leftarrow q$; $state(p) \leftarrow 2$
  **else if** $q < p$ **then**
    $label(p) \leftarrow p$; $state(p) \leftarrow 3$
  **end if**
**end for**

**repeat** ▷ Step S2: resolution of non-minimal plateaux
  **for all** $p \in I$ in parallel **do**
    **if** $state(p) \geq 2$ **and** $\exists q \in N(p), state(q) = 0 \wedge img(q) = img(p)$ **then**
      $label(p) \leftarrow q$; $newstate(p) \leftarrow 0$
    **else**
      $newstate(p) \leftarrow state(p)$
    **end if**
  **end for**
  $swap(state, newstate)$
**until** $\neg change$

**repeat** ▷ Step S3: label propagation, or path reduction
  **for all** $p \in I$ in parallel **do**
    **for** $i \leftarrow 1, RR$ **and** $label(p) \neq label(label(p))$ **do**
      $label(p) \leftarrow label(label(p))$
    **end for**
  **end for**
**until** $\neg change$

**repeat** ▷ Step S4: merging of minimal plateau labels
  **for all** $p \in I$ in parallel **do**
    **if** $state(p) \geq 2$ **then**
      **for all** $q \in N(p), state(q) \geq 2$ **do**
        $lp \leftarrow label(p)$; $lq \leftarrow label(q)$
        **while** $label(lp) \neq label(lq)$ **do**
          $label(lp) \leftarrow min(label(lp), label(lq))$
          $label(lq) \leftarrow min(label(lp), label(lq))$
        **end while**
      **end for**
    **end if**
    **while** $label(p) \neq label(label(p))$ **do**
      $label(p) \leftarrow label(label(p))$
    **end while**
  **end for**
**until** $\neg change$

Figure 1. Simple 2D neighbourhoods: Moore (left) and von Neumann (right). Inclusion of central grid point is task dependent.

from $p$ in the image. The 2D 8-pixel (3D 26-voxel) Moore neighbourhood and 2D 4-pixel (3D 6-voxel) von Neumann neighbourhood are the popular options, see Figure 1.

The initialisation step relies on the classification of the image pixels into four groups, or states, based on the image values in the pixel neighbourhoods. A fixed ordering of all the image positions is essential to the suggested algorithm; here we consider row-major order. Let $q$ be the pixel position in the neighbourhood $N(p)$ with the smallest input image value. If the minimum is not uniquely identified, $q$ is set to the largest (according to the chosen ordering) among the positions with the minimum image value in that neighbourhood:

$$q = max\{s \in N(p) | \forall t \in N(p), img(s) \leq img(t)\} \quad (1)$$

where $img(t)$ is the input image value at position $t \in I$.

If the value of the input image at $q$ is smaller than at $p$, then $q$ is the direction of steepest descent from $p$. We can set the initial label of $p$ to $q$, so that it later borrows its final label from $q$. In addition, we assign pixel $p$ into the first group of pixels with lower neighbours ($state(p) = 0$). If the input image value at $q$ is larger than at $p$, then $p$ is a local minimum. Then $label(p) = p$ and later the whole catchment basin will borrow this value for a label. The set of local minimum pixels is the second group with state fixed at 1. Finally, if the image values at $p$ and $q$ are the same, then $p$ is part of a plateau. We differentiate these into two groups: plateau pixels with at least one equal neighbour ordered after them (state 2) and plateau pixels with equal neighbours ordered only before them (state 3). In case the plateau is confirmed as minimal after step S2, plateau pixels at state 2 will borrow their label from plateau pixels at state 3. Thus, $label(p)$ is initialised as $q$ if $state(p) = 2$ and $label(p) = p$ is a self-loop if $state(p) = 3$. Figure 2 details the procedure of the path reducing watershed on a small example step by step.

The second step of the algorithm resolves non-minimal plateaux by iteratively propagating information inwards from the plateau boundaries. For every pixel $p$ in states 2 or 3, $state(p) \geq 2$, if there is a pixel $q$ in its neighbourhood such that $state(q) = 0$ and the input image values are the same at $p$ and $q$, then the state of $p$ is reset to 0 and its label is updated to equal $q$. This process is repeated iteratively, using a buffer for the states and swapping the two state arrays after each iteration, until there are no further state changes.

By the end of step S2 the states of the image pixels can be interpreted in a new way, slightly different from the initial state groups. State 0 indicates either a pixel with a lower neighbour or a non-minimal plateau pixel. A pixel in state 1 is still a local minimum. Finally, states 2 and 3 correspond to pixels only in minimal plateaux. Since labels are essentially pointers from one pixel to a neighbouring one or itself, these labels can be followed recursively to form directed paths between pixels. For all the pixels in state 0 their labels form paths leading to minima: pixels in states 1–3. These are the paths of steepest descent forming the different catchment basins. Furthermore, inside minimal plateaux the labels of state 2 pixels form additional paths into the state 3 pixels. Thus, any label path starting at any pixel will even-

| | | | | | |
|---|---|---|---|---|---|
| 100 | 105 | 105 | 105 | 104 | 104 |
| 102 | 104 | 105 | 106 | 104 | 107 |
| I | II | III | IV | V | VI |
| VII | VIII | IX | X | XI | XII |

(a) 6×2 input with expected watershed line; pixel indices

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 2 | 0 | 2 | 3 |
| 0 | 0 | 0 | 0 | 3 | 0 |
| I | I | IX | V | XI | VI |
| I | VII | VIII | XI | XI | XI |

(b) step S1: initial states and labels

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 2 | 3 |
| 0 | 0 | 0 | 0 | 3 | 0 |
| I | I | II | V | XI | VI |
| I | VII | VIII | XI | XI | XI |

(c) step S2: non-minimal plateau state and label updates

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 2 | 3 |
| 0 | 0 | 0 | 0 | 3 | 0 |
| I | I | I | XI | XI | VI |
| I | I | I | XI | XI | XI |

(d) step S3: minima labels propagated uphill to all pixels

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 2 | 3 |
| 0 | 0 | 0 | 0 | 3 | 0 |
| I | I | I | VI | VI | VI |
| I | I | I | VI | VI | VI |

(e) step S4: minimal plateau labels merged

Figure 2. Simple 6×2 example with middle greyscale input image values detailing the procedure of path reducing watershed with von Neumann neighbourhood. Pixels with a change are highlighted in red; considered but not changed pixels are green.

tually lead to a pixel in state 1 or 3, where the label is a self-loop.

Step S3 reduces all the label paths into direct pointers from any image pixel into the corresponding catchment basin minimum, i.e. state 1 or 3 pixel. This can be done independently for every pixel in parallel by following the label pointers until a self-loop is reached. However, such a procedure is linear in the length of the longest label path, hence linear in the image size in the worst case. Instead, we reduce the complexity to logarithmic in the image size by synchronising all labels every fixed number of updates and repeating the procedure iteratively. We call this number of updates between synchronisations the *reduction rate* ($RR$). For every pixel $p$ its label is updated to its label's label $RR$ times, and then the new value of the label is recorded instead of the original $label(p)$ and made available to other pixels to read. This process is repeated iteratively until there are no further label changes. The overall number of operations is $O(RR{\times}\lceil log_{RR+1}L\rceil) = O(\lceil log_{RR+1}L\rceil) \leq O(log_{RR+1}|I|)$, where $RR$ is the reduction rate, $L$ is the length of the longest path, $|I|$ is the image size. Figure 3 illustrates the path reduction process.

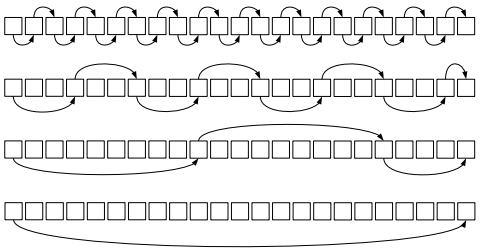

Figure 3. The reduction of a path of length 22 with a reduction rate 2 in only $\lceil log_3 22\rceil = 3$ iterations.

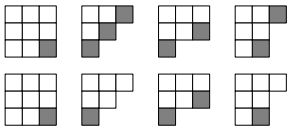

Figure 4. A few shapes for minimal plateaux with state 3 pixels highlighted in grey. Von Neumann neighbourhood in the first row; Moore neighbourhood in the second.

In the case of a 1D image, the label array after algorithm steps S1–S3 would be a valid watershed result. This is true because every (minimal) plateau in 1D is an interval and only the rightmost plateau pixel receives state 3. Hence the label array after algorithm step S3 will contain only one label value per catchment basin. However, this property breaks for higher dimensionality images: there may be several state 3 pixels in a minimal plateau, depending on its shape. Figure 4 contains a few shapes for minimal plateaux with highlighted state 3 pixels.

Step S4 solves the issue of multiple labels for minimal plateaux by merging those labels. The label array can be viewed as a disjoint set where the height of every tree is at most one after step S3. For neighbouring pixels $p, q$ in a minimal plateau, $lp = label(p), lq = label(q)$ are the roots of the corresponding trees. If $lp \neq lq$, then the two trees are merged by setting the smaller label as the parent of the larger label. Since the suggested procedure is for parallel implementation, more than two trees may be merged at the same time. Thus the check and the following value update are repeated until equality is reached between $label(lp)$ and $label(lq)$. A label path reduction loop follows for all pixels to reset the tree heights to maximum one, i.e. converting label paths into direct pointers to self-loop labels. This process of merging and reducing is repeated iteratively until there are no further label changes. After the completion of step S4, there is only one label value per catchment basin, and for each pixel $p$, $label(p)$ denotes which catchment basin $p$ belongs to.

## 2.2. GPU synchronous implementation

We consider three different implementations of the suggested path reducing watershed (PRW) in Algorithm 1. Our watershed implementations use CUDA [2], the parallel computing platform and programming model invented by NVIDIA. The first and the simplest implementation of PRW is described here and it is called *synchronous* path reducing watershed ($\text{PRW}_{\text{sync}}$). The other two are presented in Section 2.3. All three variants of the algorithm assign separate threads to every pixel in the image, and use von Neumann neighbourhood of size four in 2D and six in 3D.

$\text{PRW}_{\text{sync}}$ consists of a host code function and four CUDA kernels, one for each step of the procedure. A kernel is a function which is executed by multiple threads at the same time on the GPU device. The four kernels in $\text{PRW}_{\text{sync}}$ correspond to and implement the lines 2–11, 15–19, 25–27 and 32–43 in Algorithm 1, respectively. The host code is described in Algorithm 2 and is the same for all three variants of the watershed algorithm.

**Algorithm 2** The host code of the PRW algorithm

```
1: HostToDevice(img)                    ▷ Copy image from host to device
2: DimGrid, DimBlock                    ▷ Set grid and block dimensions
3: InitialisationKernel                 ▷ Step S1
4: repeat                               ▷ Step S2
5:     ResolutionKernel
6:     swap(state, newstate)
7: until ¬change
8: repeat                               ▷ Step S3
9:     ReductionKernel
10: until ¬change
11: repeat                              ▷ Step S4
12:     MergingKernel
13: until ¬change
14: DeviceToHost(label)                 ▷ Copy labels from device to host
```

The most expensive step of PRW, and $\text{PRW}_{\text{sync}}$ in particular, is the resolution of non-minimal plateaux. Unlike steps S1 and S3–S4, the number of iterations of the code block in lines 4–7 in Algorithm 2 (13–22 in Algorithm 1) may be linear in the image size in the worst case. For instance, if the image is the simple $(2n+2)\times 1$ example

$$105 \quad \underbrace{109 \quad 109 \quad 109 \quad \dots \quad 109 \quad 109}_{2n} \quad 106,$$

then it takes $n$ iterations to complete step S2 of $\text{PRW}_{\text{sync}}$ because of the expensive global inter-block synchronisation after each state update. The pixel states are initialised in step S1 as

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

after $n-2$ iterations of step S2 we have

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

An $(n-1)^{\text{th}}$ iteration converts the remaining 2-states into 0 and a final $n^{\text{th}}$ iteration does not make any changes and completes the resolution of non-minimal plateaux.

In addition, since the state update for the plateau pixels in lines 15–19 of Algorithm 1 takes place in parallel and the order of update completion for various pixels is non-deterministic, we need the buffer state memory $newstate$ to ensure correctness. The use of the buffer also entails fair division of non-minimal plateaux among adjacent catchment basins. For the considered example at $n$=5, the label values after the completion of step S2 are guaranteed to be

$$1 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \leftarrow 0 \qquad 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 1,$$

where the numbers are the pixel states, arrows are the labels. The final labelling after step S3 (step S4 is not required in 1D), using roman numerals instead of arrows, would be

$$\text{I} \quad \text{I} \quad \text{I} \quad \text{I} \quad \text{I} \quad \text{I} \qquad \text{XII} \quad \text{XII} \quad \text{XII} \quad \text{XII} \quad \text{XII} \quad \text{XII}.$$

Thus, the sizes of non-minimal plateaux inside an image are expected to largely affect the parallel watershed execution time. Such trends are documented in existing literature [13, 9]; our results in Section 3 agree with this assumption.

## 2.3. Block-asynchronous and balanced versions

In addition to $\text{PRW}_{\text{sync}}$, we introduce two more implementations of PRW which attempt to speed up the watershed. This is done by using the intra-block synchronisation mechanisms of CUDA. Block-level shared memory provides faster access to data to be shared between different threads; in-block synchronisation is cheaper than global synchronisation. We use these block-level instruments to reduce the number of global iterations of the non-minimal plateau resolution step.

The *block-asynchronous* path reducing watershed ($\text{PRW}_{\text{async}}$) is different from $\text{PRW}_{\text{sync}}$ in the second step of the algorithm. While the host code is the same, as given in Algorithm 2, the CUDA kernel is responsible for multiple state updates using synchronisation at block level.

The non-minimal plateau resolution kernel of $\text{PRW}_{\text{async}}$ first loads the data into the shared and local memory. Since every pixel needs access to the input image and state values of the neighbouring pixels to update its own state, this data is stored in shared memory. Every pixel $p$ is responsible for copying its $img(p)$ and $state(p)$ values into the shared memory. In addition, pixels at the boundary of the block, also copy the corresponding values of their neighbours outside of the block.

Each thread block has a separate shared memory space; during kernel execution there is no communication between different blocks. Updates from one block are not accessible to threads in other blocks until the data is copied into global memory and kernel execution is completed, i.e. global synchronisation points. This means the values in shared memory corresponding to the additional band of pixels are never changed during kernel execution (they are changed in *their own* block) and maintain their values from the previous

global synchronisation point. Information is propagated from one block to another once per global iteration.

The third variant of our watershed—*balanced* block-asynchronous path reducing watershed ($\text{PRW}_{\text{bal}}$)—makes use of the block-level synchronisation mechanisms, like $\text{PRW}_{\text{async}}$, while maintaining the fairness of watershed line positioning, similar to $\text{PRW}_{\text{sync}}$. We extend the 0–3 state space into negative numbers thus: when $state(p) < 0$ then value $|state(p)|$ denotes the distance of a non-minimal plateau pixel from the plateau boundaries, i.e. the number of state update operations for information to reach $p$. Algorithm 3 describes the non-minimal plateau resolution step (kernel) for $\text{PRW}_{\text{bal}}$.

**Algorithm 3** Non-minimal plateau resolution, $\text{PRW}_{\text{bal}}$

```
1: GlobalToLocal(label)        ▷ Copy label from global to local memory
2: GlobalToShared(img)         ▷ Copy image from global to shared
3: GlobalToShared(state)       ▷ Copy state from global to shared
4: repeat
5:    SyncThreads              ▷ Synchronise threads in block
6:    SharedToLocal(state)     ▷ Copy state from shared to local
7:    if (state(p)≥2 and ∃q∈N(p), state(q)≤0 ∧ img(q)=img(p))
      or (∃q∈N(p), state(p)+1<state(q)≤0 ∧ img(q)=img(p)) then
8:       label(p) ← q; state(p) ← state(q) − 1
9:    end if
10:   SyncThreads              ▷ Synchronise threads in block
11:   LocalToShared(state)     ▷ Copy updated state to shared memory
12: until ¬blockchange
13: LocalToGlobal(label)       ▷ Copy updated label to global memory
14: SharedToGlobal(newstate)   ▷ Copy updated shared state into
    global newstate
```

After a non-minimal-plateau-pixel changes its state from 2 or 3 to a negative value, its state can be iteratively updated to larger values (still smaller than 0) during further in-block iterations or after global synchronisation points. The execution of the resolution step will complete only when no more improvements can be made to the states of the non-minimal plateau pixels. On the simple $14\times1$ pixel example

105 109 109 109 109 109 109 109 109 109 109 109 109 106

Assuming block size 3, the first global iteration of step S2 produces states

$$\underbrace{1\ \ 0\ -1}\ \underbrace{2\ \ 2\ \ 2}\ \underbrace{2\ \ 2\ \ 2}\ \underbrace{-3\ -2\ -1}\ \underbrace{0\ \ 1}$$

After the second global iteration we get

$$\underbrace{1\ \ 0\ -1}\ \underbrace{-2\ -3\ -4}\ \underbrace{-6\ -5\ -4}\ \underbrace{-3\ -2\ -1}\ \underbrace{0\ \ 1}$$

and this is corrected after iteration 3

$$\underbrace{1\ \ 0\ -1}\ \underbrace{-2\ -3\ -4}\ \underbrace{-\mathbf{5}\ -5\ -4}\ \underbrace{-3\ -2\ -1}\ \underbrace{0\ \ 1}$$

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

The main parameters of the path reducing watershed affecting execution times are the thread block size and the reduction rate $RR$, both discussed in Section 3.4.

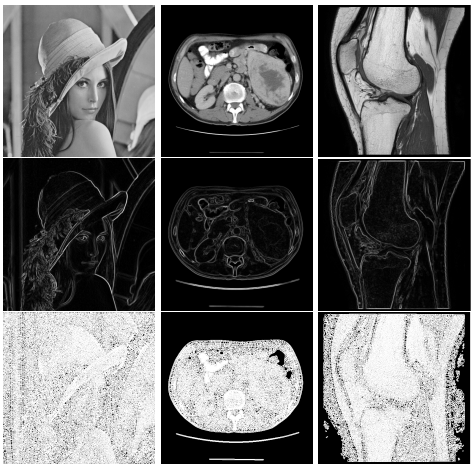

Figure 5. $\text{PRW}_{\text{bal}}$ results on three sample images: Lena, an abdominal CT slice, and a knee MRI slice. Original greyscale images; gradient magnitude images after minimal Gaussian blurring; and original images overlaid with watershed lines in white.

## 3. Results

For reproducibility reasons, the Berkeley Segmentation Data Set and Benchmarks 500 (BSDS500) [5, 1] is used to evaluate the performance of the suggested watersheds. We converted the images to greyscale without any further processing before running the watersheds.

All experiments were carried out using CUDA 8.0.61 on a x86-64 Linux machine with an NVIDIA GeForce GTX 960M graphics card of 640 CUDA cores and 2GB memory, NVIDIA driver 375.74. Every reported watershed execution time is an average of ten runs. For a given image, the equivalence of the different watershed procedures is established by checking that they consistently produce the same number of regions. For BSDS500 the number of watershed catchment basins is in the range 3,372–22,094. For clarity of presentation the images are sorted in increasing order of the $\text{PRW}_{\text{sync}}$ execution times.

### 3.1. Qualitative results

Our main interest in watershed is its application to 2D or 3D medical scan images to construct hierarchical segmentations. When applied on large data, the efficiency of watershed is crucial. The accepted pipeline is: smoothing the input image, constructing the gradient magnitude image and only then applying the watershed transform for watershed lines to correspond to object boundaries. Figure 5 illustrates the watershed results on three sample images.

The first image in Figure 5 is the classic Lena test image [4] in greyscale, the second is a slice from an abdominal CT scan, and the third—a slice from a knee MRI scan. The results are illustrated in the form of the original images on which the watershed lines are highlighted in white. These images are constructed by taking the greyscale values from

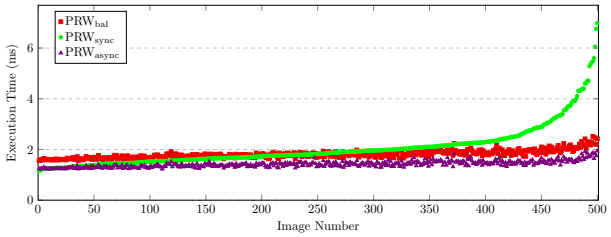


Figure 6. The execution times (average of ten runs per image) of the three variants of our watershed compared on the BSDS500 images. For clarity of presentation the images are sorted in increasing order of the PRW$_{sync}$ execution times.

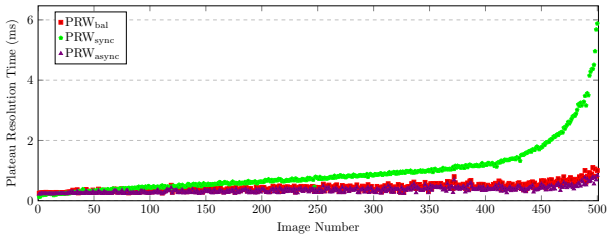


Figure 7. Execution times of non-minimal plateau resolution (algorithm step S2) of the three variants of our watershed compared on the BSDS500 images.

the original input image for pixels inside watershed catchment basins and by setting the maximum greyscale value for pixels on boundaries of catchment basins.

Our quantitative experiments presented below skip the smoothing and the gradient magnitude constructing steps, directly applying watershed on the input greyscale images.

### 3.2. Synchronous vs. block-asynchronous

We compare the performance of all three variants of PRW. Their execution times on the BSDS500 images are reported in Figure 6 sorted in increasing order of the PRW$_{sync}$ runtimes. The block size is fixed at $16\times16$ for this experiment and $RR=5$. The PRW$_{sync}$ implementation is the slowest with the sum of the execution times over the 500 images at 1,023.40 ms. PRW$_{async}$ with 709.54 ms is faster than PRW$_{bal}$ (906.25 ms) on every image in the dataset.

The average PRW$_{sync}$ execution time over BSDS500 stands at 2.047 ms, starting at 1.165 ms on image number 1 and reaching 6.989 ms on image 500. PRW$_{async}$ runs for an average of 1.419 ms; the full range of its execution times is 1.227 ms to 1.945 ms. The average execution time of PRW$_{bal}$ stands at 1.813 ms; its full range is 1.563–2.516 ms.

Figure 7 shows the execution times of the non-minimal plateau resolution step of the three variants of our watershed. Plateau resolution, being the main difference among the three implementations and the most time-consuming step, is behind the variability in the execution times. There is an obvious pattern between the results in Figures 6 and 7. The watershed runtimes excluding the plateau resolution step are stable for PRW$_{sync}$, PRW$_{async}$ and PRW$_{bal}$ across the 500 images. These values are slightly higher for PRW$_{bal}$; this is explained by the four times larger memory used for storing pixel states to accommodate the larger state space.

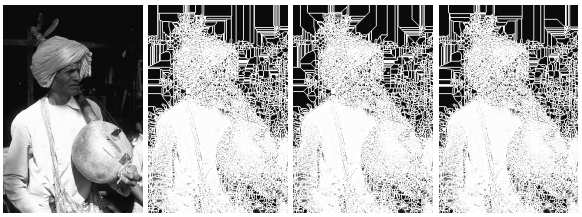

Figure 8. Image 498 and the corresponding PRW$_{sync}$, PRW$_{async}$ and PRW$_{bal}$ results, depicted as overlays with the watershed lines in white. Apparent differences in the location of the watershed lines can be observed on the left and at the top of the image.

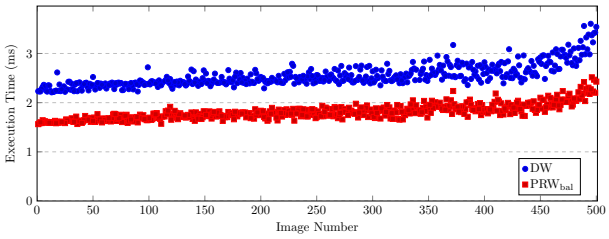


Figure 9. Comparison of execution times (average of ten runs per image) of the DW watershed and our PRW$_{bal}$, on BSDS500, excluding six images on which the DW implementation deadlocks. The sums of the average execution times over the 494 images are 1,257.83 ms and 895.30 ms respectively. Thus our algorithm shows an execution time improvement of more than 28.8%.

Figure 8 demonstrates the differences in watershed line locations on non-minimal plateaux on the example of image 498. Apparent differences in how the non-minimal plateaux are divided among catchment basins can be observed on the left and at the top of the image.

### 3.3. Comparison against the state-of-the-art

We compare our PRW$_{bal}$ transform against other watershed algorithms, suggested earlier for GPU implementation. Direct comparison of execution times reported in different papers is not immediately possible. First, the GPU devices used belong to different generations, contain different numbers of streaming machines, CUDA cores, different memory sizes, etc. Second, even small differences in input affect number of watershed catchment basins and execution time. The only genuine comparison is the execution of different implementations on the same images on the same device.

Körbes *et al.* showed in [9] that the CUDA implementation of their DW (original implementation available from [3]) algorithm was superior to CA—an algorithm suggested by Kauffmann and Piché [7, 8]. A more detailed description of DW, including pseudocode, can be found in [16, 17].

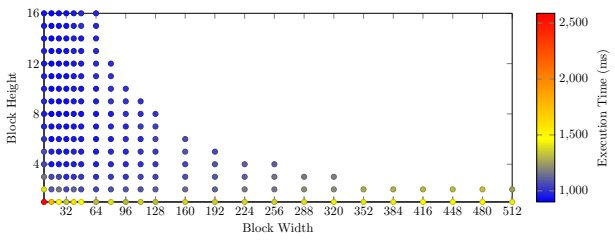


Figure 10. Comparison of execution times of our $\text{PRW}_{\text{bal}}$ with different 2D block sizes on BSDS500.

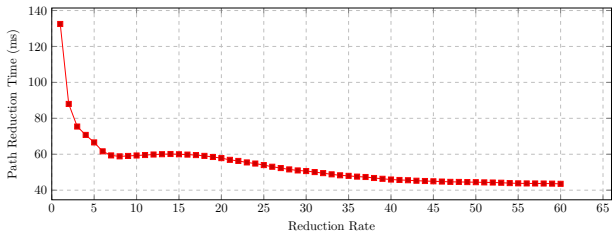


Figure 11. Comparison of overall execution times (sum of average step S3 runtimes) of the path reduction (algorithm step S3) of $\text{PRW}_{\text{bal}}$ for different $RR$ values on the BSDS500 images.

A comparison of the DW runtimes against our $\text{PRW}_{\text{bal}}$ is presented in Figure 9. This excludes six images (out of 500) on which the DW implementation [3] deadlocks. We suspect that it is a warp divergence issue but we have not attempted to debug their code to confirm or reject that.

Quesada-Barriuso *et al.* [12, 13] introduced another parallel watershed for GPU based on cellular automata, of which we were unable to find any implementations. However, the authors report 'similar speedup results' over CPU watershed for their 'block-asynchronous proposal of the CA-watershed described' and DW from [9]. From this we conclude that our PRW would easily outperform the CA-based watershed from [13] if executed on the same device.

We do not compare directly against [11, 8, 18] because they have been superseded.

### 3.4. Parameters

For fixed $RR$=5, we consistently change the CUDA block size for $\text{PRW}_{\text{bal}}$ and measure the overall execution time for the BSDS500 images (the sum of averages over ten runs per image). The results are reported in Figure 10. We consider block widths of 8 to 512 and block heights of 1 to 16. Block sizes vary from 8×1=8 to 512×2=1,024 threads—the maximum block size possible.

Setting the threshold at 920 ms, the number of valid block sizes decreases to 7. The best three configurations are 16×8, 16×14 and 16×16 with the runtimes between 901 ms and 904 ms. In this experiment **the best performance is achieved for a block size of** 16×16, which should be preferred for the general case.

The choice of the best $RR$ value is obviously data-specific. For block size fixed at 16×16, we consider variation in $RR$ values. The sums of the average path reduction step runtimes over the BSDS500 image set based on $RR$ values 1–60 are reported in Figure 11.

| w×h×d | 4K×4K×8 | 500×500×512 | 320×320×1250 |
|---|---|---|---|
| 8×8×8 | 1,424.52 | **1,597.34** | **2,138.56** |
| 8×8×16 | 1,981.13 | 1,798.06 | 2,196.38 |
| 8×16×8 | 1,586.81 | 1,846.73 | 2,184.11 |
| 16×8×8 | **1,360.30** | 1,652.04 | 2,165.99 |
| 16×16×1 | 1,795.43 | 3,376.66 | 5,424.09 |
| 16×16×2 | 1,470.40 | 2,325.75 | 3,317.66 |
| 16×16×3 | 1,564.14 | 2,150.56 | 2,935.89 |
| 16×16×4 | 1,435.23 | 2,004.19 | 2,484.55 |

Table 1. Comparison of execution times (in ms) of our $\text{PRW}_{\text{bal}}$ with different 3D w×h×d block sizes on the three 3D test images. The best performance for each image is highlighted.

Figure 11 shows that larger $RR$ is preferable for this specific image set: the worst overall step S3 runtime of 132.50 ms is observed when $RR$=1 and the 43.55 ms at $RR$=60 is the best among the considered values. The runtimes correlate well with the number of the global iterations. These results are explained by the fact that catchment basins in the images in BSDS500 are generally very small with short label paths from catchment basin boundary pixels to catchment basin minima. For these images the actual label updating process is a lot cheaper than the global synchronisation, hence fewer global iterations means smaller step S3 runtime. Furthermore, for BSDS500 images the setting $RR$=$\infty$ works better with overall execution time at 42.99 ms. This can be further decreased to 32.90 ms if we replace the iterative global synchronisation with a single kernel call of linear path reduction. It is important to remember that this setting is optimal only for small enough images with little catchment basins and short label paths.

### 3.5. Large 3D images

In order to analyse the performance of PRW on typical large 3D images we consider a micro CT image volume of total size 4,000×4,000×1,250 and construct three test subimages. A 4,000×4,000×8 image is made by taking the middle 8 slices of the original image; the middle 512 slices at a reduced resolution of 500×500 form a second test image of size 500×500×512; reducing the intra-slice resolution even further results in the third test image of size 320×320×1,250. The overall number of voxels is exactly the same in all three test images—128,000,000.

Fixing $RR$=5, different configurations for the block size are considered. If the block width and the block height are fixed at 16 as per the best setting from our 2D experiments, the block depth will have to be at most 4 because of the block size limitation at 1,024 threads. Thus we also consider block sizes with two or three dimensions fixed at 8.

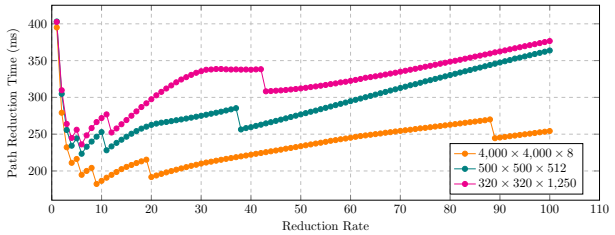


Figure 12. Comparing execution times (average step S3 runtimes) of the path reduction (algorithm step S3) of $\text{PRW}_{\text{bal}}$ for different $RR$ values on the 3D test images.

| | 4K×4K×8 | 500×500×512 | 320×320×1250 |
|---|---|---|---|
| $\text{PRW}_{\text{sync}}$ | 2,306.06 | 3,054.32 | 4,827.83 |
| $\text{PRW}_{\text{async}}$ | 1,226.01 | 1,463.46 | 1,912.05 |
| $\text{PRW}_{\text{bal}}$ | 1,389.85 | 1,595.47 | 2,120.08 |
| sync/as | 1.881× | 2.087× | 2.525× |
| sync/bal | 1.659× | 1.914× | 2.277× |

Table 2. The execution times (in ms, average of ten runs per image) of the three variants of our watershed compared on the 3D test images. The speedups of the block-asynchronous implementations over the synchronous procedure are reported in the last two rows.

$\text{PRW}_{\text{bal}}$ results on the three test images for all the tested block sizes are reported in Table 1. The cube-shaped block size 8×8×8 holds the lowest runtimes for two out of the three test images. For the test image of size 4,000×4,000×8 a better runtime is seen with 16×8×8 blocks, which is explained by the shape of the test image with only few slices.

In Figure 12 we examine the effect of various $RR$values on $\text{PRW}_{\text{bal}}$ step S3 runtimes for the three test images with thread block size set at 8×8×8. The best performance for the 500×500×512 and 320×320×1,250 test images is seen at $RR$=6, and for the 4,000×4,000×8 image $RR$=9 achieves lowest runtime. We conclude that for large 3D medical images with potentially large watershed catchment basins $RR$ **in the range 5–10 is optimal**.

Finally, the average execution times of all three variants of the path reducing watershed on the three 3D test images are reported in Table 2 with block size 8×8×8 and $RR$=6. All nine runtimes are below 5 s, and all six block-asynchronous runtimes (less than 2.2 ms) are faster than the three $\text{PRW}_{\text{sync}}$ runtimes. $\text{PRW}_{\text{async}}$ is on average 2.164 times faster than $\text{PRW}_{\text{sync}}$. For $\text{PRW}_{\text{bal}}$ the average speedup over $\text{PRW}_{\text{sync}}$ stands at a factor of 1.95.

Figure 13 illustrates the composition of the $\text{PRW}_{\text{bal}}$ execution times—data copy into the device, steps S1–S4, data copy from the device—on the 3D test images. The iterative steps with global synchronisation points, i.e. S2–S4, take the longest runtimes. Furthermore, the major variation in execution times across the three images is caused by the non-minimal plateau resolution step S2, reaching 1,255.06 ms for the image 320×320×1,250.

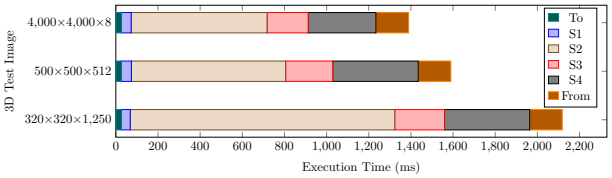


Figure 13. The composition of the $\text{PRW}_{\text{bal}}$ execution times—data copy into the device, steps S1–S4, data copy from the device—on the 3D test images.

In Tables 1 and 2, and in Figure 12, there is a correlation between the number of slices in the test images and the PRW execution times: the larger the number of slices and the smaller the intra-slice resolution, the longer the PRW execution times. The main reason behind this trend is the weaker memory locality for the images with more slices. While adjacent voxels in the x-direction occupy adjacent memory positions and may be loaded from the global memory at the same time, the data for two adjacent voxels in the z-direction needs to be loaded in at least two copies.

## 4. Conclusions

We introduced and discussed three variants (synchronous, block-asynchronous and balanced block-asynchronous) of a new parallel watershed transform for execution on the GPU. Our approach relies on the construction of paths of steepest descent and their logarithmic reduction into pointers to catchment basin minima. The procedure also successfully incorporates the identification and resolution of plateaux, thus no preprocessing is required for that.

We considered various parameter settings to reveal execution time patterns; compared the three variants of the path reducing watershed on a large number of images to find runtime ranges for fixed size images of different construction. Tests with CUDA implementations proved our path reducing watershed superior to state-of-the-art parallel watershed algorithms for the GPU. For 3D images of 128,000,000 voxels we achieved execution times of approximately 1.5–2 seconds, depending on the image dimensions.

An important limit to the performance of our watershed and similar ones [9, 13] is the size of the available memory: the whole image needs to be stored in the GPU memory. For 3D images significantly larger than the ones tested, an alternative approach of streamed watershed [6] may be more appropriate.

The extended version of this paper will also include a detailed discussion of the influence of the various parameters on the number of global iterations.

Future work will focus on the application of our watershed for constructing hierarchical segmentation graphs for medical scan data.

## References

[1] BSDS500. www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html. Accessed: 4/9/2017.

[2] CUDA™. www.nvidia.com/object/cuda_home_new.html. Accessed: 4/9/2017.

[3] DW. adessowiki.fee.unicamp.br/adesso/wiki/watershed/ismm2011_dw/view/. Accessed: 4/9/2017.

[4] Standard test images. www.ece.rice.edu/~wakin/images/. Accessed: 4/9/2017.

[5] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):898–916, May 2011.

[6] M. Hučko and M. Šrámek. Streamed watershed transform on GPU for processing of large volume data. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, pages 137–141, New York, NY, USA, 2012. ACM.

[7] C. Kauffmann and N. Piche. Cellular automaton for ultra-fast watershed transform on GPU. In *ICPR*, pages 1–4, 2008.

[8] C. Kauffmann and N. Piché. Seeded ND medical image segmentation by cellular automaton on GPU. *International Journal of Computer Assisted Radiology and Surgery*, 5(3):251–262, 2010.

[9] A. Körbes, G. B. Vitor, R. de Alencar Lotufo, and J. V. Ferreira. Advances on watershed processing on GPU architecture. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 260–271. Springer, 2011.

[10] A. Körbes, G. B. Vitor, J. V. Ferreira, and R. de Alencar Lotufo. A proposal for a parallel watershed transform algorithm for real-time segmentation. In *Proceedings of Workshop de Visão Computacional WVC*, 2009.

[11] L. Pan, L. Gu, and J. Xu. Implementation of medical image segmentation in cuda. In *2008 International Conference on Information Technology and Applications in Biomedicine*, pages 82–85, May 2008.

[12] P. Quesada-Barriuso, D. B. Heras, and F. Argüello. Efficient GPU asynchronous implementation of a watershed algorithm based on cellular automata. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 79–86, July 2012.

[13] P. Quesada-Barriuso, D. B. Heras, and F. Argüello. Efficient 2D and 3D watershed on graphics processing unit: block-asynchronous approaches based on cellular automata. *Computers and Electrical Engineering*, 39(8):2638–2655, 2013.

[14] J. B. Roerdink and A. Meijster. The watershed transform: Definitions, algorithms and parallelization strategies. *Fundamenta informaticae*, 41(1, 2):187–228, 2000.

[15] G. B. Vitor, J. V. Ferreira, and A. Körbes. Fast image segmentation by watershed transform on graphical hardware. In *XXX Iberian Latin American Congress on Computational Methods in Engineering - CILAMCE 2009*, volume 1, pages 1–14, Nov. 2009.

[16] G. B. Vitor, A. Körbes, R. de Alencar Lotufo, and J. V. Ferreira. Analysis of a step-based watershed algorithm using CUDA. *International Journal of Natural Computing Research (IJNCR)*, 1(4):16–28, 2010.

[17] G. B. Vitor, A. Körbes, R. de Alencar Lotufo, and J. V. Ferreira. Analysis of a step-based watershed algorithm using CUDA. In *Nature-Inspired Computing Design, Development, and Applications*, pages 321–335. IGI Global, 2012.

[18] B. Wagner, P. Müller, and G. Haase. A parallel watershed-transformation algorithm for the GPU. In *Workshop on Applications of Discrete Geometry and Mathematical Morphology*, pages 111–115, 2010.