International Journal on Document Analysis and Recognition (IJDAR) (2018) 21:41–58
https://doi.org/10.1007/s10032-017-0293-7

ORIGINAL PAPER

Efficient document image binarization using heterogeneous computing and parameter tuning

Florian Westphal[1] · Håkan Grahn[1] · Niklas Lavesson[1]

Received: 15 December 2016 / Revised: 19 October 2017 / Accepted: 19 December 2017 / Published online: 13 January 2018

Abstract
In the context of historical document analysis, image binarization is a first important step, which separates foreground from background, despite common image degradations, such as faded ink, stains, or bleed-through. Fast binarization has great significance when analyzing vast archives of document images, since even small inefficiencies can quickly accumulate to years of wasted execution time. Therefore, efficient binarization is especially relevant to companies and government institutions, who want to analyze their large collections of document images. The main challenge with this is to speed up the execution performance without affecting the binarization performance. We modify a state-of-the-art binarization algorithm and achieve on average a 3.5 times faster execution performance by correctly mapping this algorithm to a heterogeneous platform, consisting of a CPU and a GPU. Our proposed parameter tuning algorithm additionally improves the execution time for parameter tuning by a factor of 1.7, compared to previous parameter tuning algorithms. We see that for the chosen algorithm, machine learning-based parameter tuning improves the execution performance more than heterogeneous computing, when comparing absolute execution times.

Keywords Image binarization · Heterogeneous computing · Automatic parameter tuning · Historical documents

1 Introduction

Historical handwritten documents have been archived on microfilm for several decades. More recently, the possibility to digitize such documents has made them available to a broader audience via online services of companies and government agencies. However, this increase in people accessing historical document images also increases the demand for simplified access to these images. Pattern matching and recognition algorithms can help with this by making document images searchable [42,53] or by transcribing them [11,28]. Many of these algorithms require that the written text is first separated from its background. This procedure is called image binarization. Since pattern matching and recognition algorithms will only process image pixels, which were classified as written text, this first step is vital for further processing.

With this motivation, it is natural to choose the best available binarization algorithm for this important first step. However, while binarization performance is important to extract as much information as possible, the execution performance of the used binarization algorithm has also great significance when dealing with large amounts of document images. Therefore, efficient binarization is especially relevant for companies and government agencies, whose document image collections are growing continuously, easily surpassing 50 million images and more. Provided that tuning the parameters of a binarization algorithm and performing the binarization takes 15 seconds per image, it will take more than 23 years to binarize those 50 million images. The main challenge in speeding up the execution performance of a binarization algorithm is

This work is part of the research project "Scalable resource-efficient systems for big data analytics" funded by the Knowledge Foundation (Grant: 20140032) in Sweden.

✉ Florian Westphal
florian.westphal@bth.se

Håkan Grahn
hakan.grahn@bth.se
http://grahn.cse.bth.se/

Niklas Lavesson
niklas.lavesson@bth.se
http://www.bth.se/people/nla

1 Department of Computer Science and Engineering, Blekinge Institute of Technology, Karlskrona, Sweden

to find ways which do not affect its binarization performance.

This paper describes how to reduce the execution time of Howe's binarization algorithm (HBA) [18,19], a state-of-the-art binarization algorithm, without reducing its binarization performance, by:

- mapping it to a heterogeneous platform consisting of one CPU and one GPU
- efficiently tuning its parameters using multivariate regression

We experimentally evaluate our proposed solutions using seven standard benchmark datasets from the Document Image Binarization Contest (DIBCO) and the Competition on Handwritten Document Image Binarization (H-DIBCO). We show that HBA can be executed on average 3.5 times faster when running completely on the GPU of a heterogeneous platform than when running only on the CPU. Additionally, we show that a similar execution performance can be achieved by executing one part of the algorithm on the CPU and the remaining parts on the GPU.

Furthermore, we analyze the correlation between binarization quality and execution time with respect to different binarization parameters and evaluate the performance of our proposed parameter tuning algorithm for HBA. Our analysis shows that there is a clear negative correlation between execution time and binarization quality and that our proposed parameter tuning algorithm chooses parameters on average 1.7 times faster than Howe's parameter tuning algorithm [19] with comparable binarization results.

We provide more information on HBA, heterogeneous computing, and other topics relevant for this paper in Sect. 2. Section 3 describes how we map HBA to a heterogeneous platform, and Sect. 4 illustrates our parameter tuning algorithm. We describe the design of the different experiments performed to evaluate our proposed solutions in Sect. 5 and analyze the obtained results in Sect. 6. Section 7 discusses related work, and Sect. 8 concludes this paper.

2 Background

In this section, we introduce a few basic concepts, which are used throughout the paper. Those concepts are image binarization, heterogeneous computing, and multivariate regression. We describe image binarization in Sect. 2.1 together with Howe's binarization algorithm (HBA) [18,19] and two graph cut algorithms. A general introduction to heterogeneous computing and its challenges is given in Sect. 2.2.

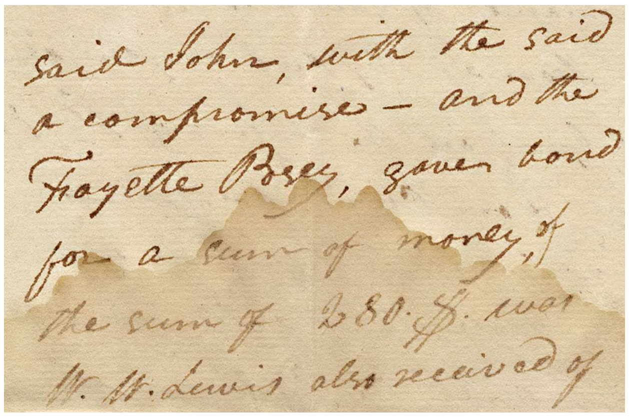
said John, with the said
a compromise – and the
Fayette Bsey, gave bond
for a sum of money, of
the sum of 280 $. was
W. M. Lewis also received of

Fig. 1 Image from the H-DIBCO 2012 dataset illustrating common image degradations, such as faded ink and stains covering the written text

Section 2.3 describes how random forests [4] can predict two continuous values.

2.1 Image binarization

Image binarization is the process of classifying image pixels as foreground or background pixels. For images of handwritten documents, this means to separate the written text from its background. This is especially challenging when dealing with images of historical documents, which can have various degradations. Figure 1 illustrates common degradations, such as faded ink and stains covering the written text. Ink bleeding through from the other side of the page, as shown in Fig. 6b, is another common problem. A binarization algorithm for historical documents has to be able to cope with these degradations, which increase the risk that foreground pixels are accidentally classified as background pixels or vice versa.

Binarization algorithms by Otsu [31], Niblack [29], and Sauvola et al. [43] are well known. These algorithms are commonly used as baseline for comparison with new algorithms, for example, in competitions, such as the Document Image Binarization Contest (DIBCO) held at the International Conference on Document Analysis and Recognition (ICDAR) [12,37,39] or the Competition on Handwritten Document Image Binarization (H-DIBCO) held at the International Conference on Frontiers in Handwriting Recognition (ICFHR) [30,36,38,40]. In the last two binarization competitions, in 2014 and 2016, the binarization algorithm by Howe [19] and derivations thereof using different preprocessing steps have won these contests. Therefore, we decided to work with Howe's binarization algorithm.

2.1.1 Howe's binarization algorithm (HBA)

HBA labels image pixels as foreground or background pixels by minimizing a global energy function. This function

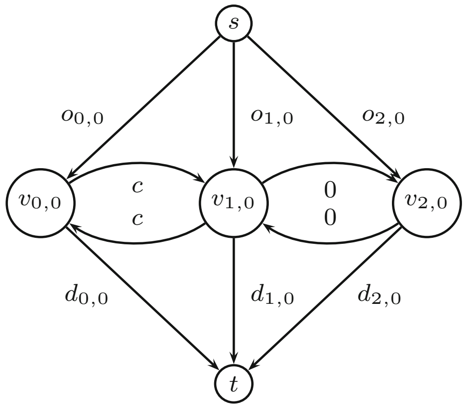

Fig. 2 Example graph constructed for a 3×1 intensity image with an image edge between $v_{1,0}$ and $v_{2,0}$

penalizes labelings that do not conform with the image's Laplacian, that is, it penalizes pixels in intensity valleys (ink), which are classified as background and pixels in intensity plateau or peak areas (background), which are classified as foreground. Additionally, to this simple separation based on the divergence of the gradient, the energy function penalizes labeling discontinuities, unless they take place at an edge, as detected by the Canny edge detection algorithm [5]. This addition improves the stability of the binarization algorithm and encourages the continuity of foreground and background areas.

In order to minimize the energy function, HBA finds the minimal cut to separate foreground and background pixels with help of a graph cut algorithm. Therefore, HBA needs to build a directed graph $G = (V, E)$ based on the image's Laplacian and edge image on which the graph cut algorithm can operate. For an $m \times n$ intensity image of pixel values $i_{x,y}$ with $0 \leq x < m$ and $0 \leq y < n$, the graph is built by adding a vertex $v_{x,y}$ for each image pixel $i_{x,y}$. Furthermore, a vertex for the source s and the sink t of the graph are added. Every vertex $v_{x,y}$ is connected with s and t using directed edges from s to $v_{x,y}$ and from $v_{x,y}$ to t. Figure 2 illustrates this construction for a 3×1 intensity image. The capacities of the directed edges from s to $v_{x,y}$ are denoted as $o_{x,y}$ to indicate the flow origin, while the capacities of the edges from $v_{x,y}$ to t are denoted as $d_{x,y}$ to indicate the flow destination. These capacities are computed for each image pixel $i_{x,y}$ based on its Laplacian value $\bigtriangledown^2 i_{x,y}$ as follows:

$$o_{x,y} = L - \bigtriangledown^2 i_{x,y} d_{x,y} = L + \bigtriangledown^2 i_{x,y}. \quad (1)$$

It is to note that L represents an adjusting constant to avoid negative edge capacities and the mentioned Laplacian values are biased by assigning a high value to bright outlier, i.e., background pixels. Additionally, each vertex $v_{x,y}$ is connected with its four neighbors (top, bottom, left and right) using edges in both directions. The capacities of those edges are set to zero if an edge was detected between the neighboring pixels. Otherwise, the capacities are set to a predefined penalty value c. This is illustrated in the example graph shown in Fig. 2, which assumes that an image edge was detected between $v_{1,0}$ and $v_{2,0}$. Due to this construction, the applied graph cut algorithm will assign foreground and background labels by trading off the similarity between assigned label and the image's Laplacian with labeling continuity.

Howe identified two HBA parameters with the highest impact on the binarization result: the high threshold t_{hi} for the Canny edge detection algorithm and the penalty value c for penalizing labeling discontinuities. Due to their high impact on the binarization result, Howe developed an algorithm to tune these parameters automatically [19]. The general idea for tuning the penalty value c is to minimize the energy function for a series of different c values and then to compare every two consecutive images based on an instability measure. This measure is the normalized difference between two images. Then, the image with the smallest instability value is chosen as final result. Along with the parameter c, t_{hi} is tuned. Howe argues that it is sufficient to pick between two high threshold values τ_1, τ_2 to speed up the parameter tuning process. However, tuning t_{hi} requires to tune the parameter c as described above for τ_1, τ_2 and their mean value τ_0. This results in three binarized images B_0, B_1 and B_2. The previously described instability measure is used on B_0, B_1 and B_0, B_2 to compare each of the high threshold candidate values with their mean. The high threshold with the lowest instability value is chosen as value for t_{hi}. While this tuning algorithm produces sufficient binarization results, it is computationally expensive, since tuning the parameter c requires 33 trial binarizations. Therefore, the complete algorithm requires 99 trial binarizations, since c is tuned for each of the three t_{hi} values. This is only feasible, because the graph for the energy minimization can be reused for all trial binarizations performed for one t_{hi} value

2.1.2 Graph cut algorithms

As mentioned before, energy minimization with help of a graph cut algorithm is at the core of HBA. This class of algorithms, also known as min-cut/max-flow algorithms, was originally developed to analyze the maximum flow through a given transportation or communication network [10]. Nevertheless, those algorithms have also been widely applied in computer vision for image restoration [15], stereo analysis [41], and segmentation [20].

The key idea behind the graph cut algorithms is that some resource is flowing from the source vertex s of the graph to the sink vertex t. The directed edges between s, t and their intermediate vertices model connections through which flow can travel from source to sink. The capacity of each edge indicates how much flow can travel through this particular connection. The question is then how much flow can maximally be pushed from source to sink, given the particular graph with its directed edges and edge capacities.

To solve this problem, two different strategies can be applied. One of these strategies is the Ford–Fulkerson method [10], and the other is the push–relabel approach by Goldberg and Tarjan [14]. In the following, we will introduce both strategies shortly.

Ford–Fulkerson Algorithms following this method find the maximum flow in a given graph by iteratively picking paths from the source s to the sink t and augmenting the flow along those paths. For each path, the maximum flow along this path is added to the overall flow through the graph and the capacities of each edge on this path are reduced by this maximum flow. The maximum flow along one path is equal to the smallest residual capacity of the edges constituting this path. Therefore, paths are invalid if the residual capacity of one of their edges is zero, simply because nothing can flow from source to sink along this path. For example in Fig. 2, the path $(s, v_{0,0}, v_{1,0}, t)$ is valid for $c > 0$, while the path $(s, v_{2,0}, v_{1,0}, t)$ is invalid. The algorithm terminates when no valid path between source and sink can be found.

There are several variations of this method, which differ mostly in how paths are picked for augmentation. A few examples for algorithms following this strategy are the original Ford–Fulkerson algorithm [10], the Edmonds–Karp algorithm [9], and the algorithm proposed by Boykov and Kolmogorov [3].

Push–Relabel Algorithms following the push–relabel approach use a different intuition than Ford–Fulkerson style algorithms. While Ford–Fulkerson views vertices as pipe junctions, push–relabel views them as reservoirs. The key difference in these views is that Ford–Fulkerson's view requires the flow into a vertex to be as high as the flow out of this vertex. Push–relabel, on the other hand, allows that a vertex receives more flow than what can flow out of this vertex. This additional flow is called excess flow and is recorded for every vertex. The ultimate goal of the algorithm is to distribute this excess flow between the reservoirs, so that a maximum amount of flow reaches the sink. Another vertex property not found in Ford–Fulkerson is the notion of height. The idea is that each reservoir is located at a certain height level and that flow can only be pushed from one reservoir $v_{x,y}$ to another reservoir $v_{x+1,y}$ if $v_{x,y}$ is located higher than $v_{x+1,y}$.

At the beginning of the algorithm execution, all vertices between source and sink have a height of zero and no excess flow. The height of the source is fixed to the number of vertices in the graph and the sink's height is set to zero. The algorithm begins then to push as much flow from the source to its connected vertices as allowed by the respective edge capacity $o_{x,y}$. After this initialization step, the algorithm picks vertices and applies one of two possible operations on them. Those operations are called *push* and *relabel*.

The *push* operation distributes excess flow from the chosen vertex to one of its neighbors. This operation is only applicable, if the chosen vertex actually has excess flow and if the directed edge through which the flow is to be pushed has a residual capacity greater than zero. As mentioned before, pushing flow is also only possible if the chosen vertex is higher than the vertex to which flow should be pushed. Therefore, after initialization, the *push* operation can only be used to push flow from the vertices $v_{x,y}$ to the sink, since all vertices $v_{x,y}$ have initially the same height.

The *relabel* operation adjusts the height of a vertex to enable the distribution of excess flow between the vertices $v_{x,y}$. It is only applicable to vertices, which have an excess flow and whose height is lower than or equal to the heights of all neighboring vertices. This only considers connected neighbors, i.e., neighbors which are connected to the chosen vertex by a directed edge outgoing from the chosen vertex, if the residual capacity of this edge is greater than zero. The height of the chosen vertex is then set to the height of its lowest connected neighbor plus one. So, if $v_{x,y}$ is the chosen vertex and $v_{x+1,y}$ and $v_{x,y+1}$ are its connected neighbors with the respective heights $h[v_{x,y}] = 0$, $h[v_{x+1,y}] = 4$ and $h[v_{x,y+1}] = 2$, the height of $v_{x,y}$ is set to 3.

By iteratively applying both operations, as much flow as possible reaches the sink, limited by the graph's edge capacities. The remaining excess flow is pushed back to the source, since the source's height is fixed while the height of the vertices $v_{x,y}$ only grows. The push–relabel algorithm terminates if there is no vertex left to which either *push* or *relabel* can be applied.

In contrast to Ford–Fulkerson which needs to look at the whole graph to pick paths from source to sink, push–relabel operates locally only on the chosen vertex and its connected neighbors. Therefore, push–relabel is very well suitable for parallel execution. This property of the push–relabel approach was used by Vineet et al. [52] and Peng et al. [34] to implement graph cuts on the GPU.

2.2 Heterogeneous computing

A heterogeneous system combines different processor types [32]. In a typical computer, these types are multicore CPUs and GPUs. Heterogeneous computing uses those different processors simultaneously to improve the execution speed of an algorithm. One simple way to achieve this is to distribute different parts of the algorithm to different processors. This allows two levels of parallelism, namely the parallel execution of those different parts on different processors and on the other hand the parallel execution of each of these parts on its respective processor.

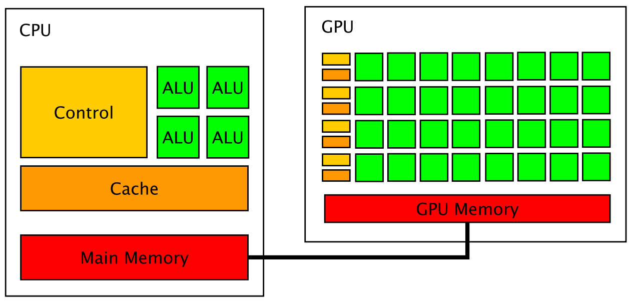

Fig. 3 Illustration of a heterogeneous platform consisting of CPU and GPU

Despite this potential for increasing the execution speed, mapping an algorithm onto a heterogeneous platform entails many challenges. One of these challenges is memory management. As shown in Fig. 3, the CPU uses the computer's main memory, while the GPU possesses its own, smaller memory space. Moving data between those different memory spaces inflicts high performance penalties. If not considered carefully, this performance loss can outweigh the speedup gained by parallel execution.

Another challenge is the diversity of execution models of the different processor types. This diversity makes it necessary to carefully select and optimize algorithms depending on which processor is executing them. For instance, GPUs are optimized for data parallel execution by larger amounts of arithmetic logic units (ALU), but smaller control units and caches, as shown in Fig. 3. This means that GPUs perform well if the same sequence of operations has to be performed on different portions of the data, while they perform poorly if this is not the case.

Apart from these challenges, the code execution on different processor types has to be coordinated. One way to do this is to use the OpenCL standard [49]. This standard defines a unified programming interface, which allows to program and coordinate the execution of programs on many different processor types.

2.3 Multivariate regression

In machine learning, multivariate regression refers to the task of estimating several real-valued target variables based on a set of independent variables. The corresponding learning problem is to learn a mapping $\hat{f}$ with $\hat{f} : \mathscr{X} \rightarrow \mathbb{R}^n$ from training data. Here, $\mathscr{X}$ denotes the instance space and n denotes the number of target variables. An instance $\mathbf{x} \in \mathscr{X}$ is a vector of independent variables, called features.

While there is a large variety of machine learning algorithms which can be used to learn such a mapping $\hat{f}$, we will focus only on one particular algorithm, namely *random forests* [4]. The general idea behind random forests is to build a predefined number of decision trees from random subsets of the training data. Each tree is then built from a randomly selected set of features from one of the random subsets. The regression result $\mathbf{y} \in \mathbb{R}^n$ is then computed by averaging over the regression results of each of the decision trees.

3 Heterogeneous binarization

There are different paths to improve HBA's execution performance. In this section, we focus on implementation-based speedup. Therefore, we describe how HBA can be implemented on a heterogeneous platform to parallelize image binarization. For this, we decompose HBA and map it onto such a platform consisting of a CPU and a GPU, Sect. 3.1. Additionally, we motivate why global relabel, a common heuristic for the push–relabel algorithm, is crucial when binarizing images of text documents, Sect. 3.2.

3.1 Algorithm decomposition and mapping

When mapping an algorithm to a heterogeneous platform, it is important to identify parts of the algorithm which can run independently. This refers to parts of the algorithm which do not interact with each other to compute their result. In case of HBA, one example for such independent parts are the Laplacian computation and the Canny edge image computation. These two parts can be computed without any interaction between them. Therefore, they can be computed on different processing units without the need for synchronization or sharing of data, which would slow down the computation.

Another important consideration are the data dependencies between those independent parts. The reason for this is the limited bandwidth between the computer's main memory and GPU memory. Therefore, the amount of data exchanged between different independent parts should be limited as much as possible to avoid the slow data transfer. One example for this is HBA's Laplacian computation and the biasing of the Laplacian image for certain background pixels. The computation of the Laplacian image and the identification of bright outlier pixels can be computed independent from each other. However, separating them into two different steps introduces an additional data exchange, which outweighs the benefit of the possible execution on different processing units.

With those considerations in mind, we decompose HBA into three different parts: (1) Laplacian image and background bias computation, (2) Canny edge image computation, and (3) energy minimization. Apart from these parts, Fig. 4 shows the exchanged data, as well as the dependencies between parts. Due to this decomposition, the only data to be exchanged are the initial grayscale image, the Laplacian image, the Canny edge image, and the binarized image. Compared to the computation performed in each part, this constitutes a reasonable tradeoff between computation and

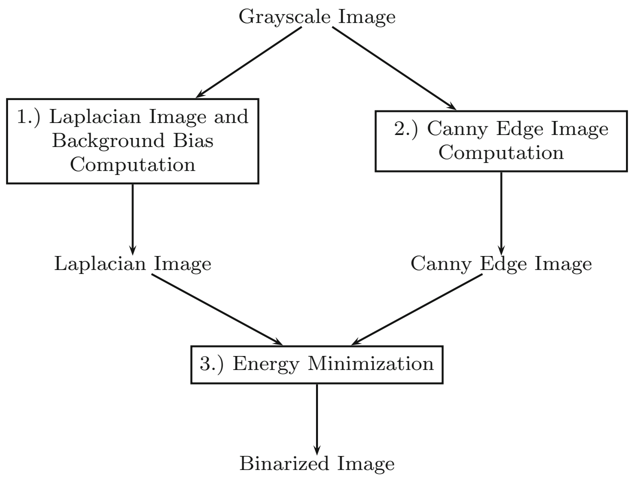

Fig. 4 Decomposition of HBA into three independent parts

Table 1 Mapping of algorithm parts to CPU and GPU. GPU has been typeset in bold font solely for the purpose of better readability

	I	II	III	IV	V	VI	VII	VIII
(1)	CPU	CPU	**GPU**	**GPU**	CPU	CPU	**GPU**	**GPU**
(2)	CPU	**GPU**	CPU	**GPU**	CPU	**GPU**	CPU	**GPU**
(3)	CPU	CPU	CPU	CPU	**GPU**	**GPU**	**GPU**	**GPU**

data transfer. Additionally, each of the three parts is independent from the other two, so that no data is exchanged during computation of any of those parts. However, while part (1) and (2) can be executed directly based on the grayscale image, part (3) depends on the results of the previous two steps and can therefore only start after those two steps completed.

Due to the chosen decomposition, it is possible to execute the three different parts of HBA on different processing units. In this paper, we focus only on CPU and GPU as processing units. Therefore, we obtain eight different mappings of HBA parts to a heterogeneous platform, as shown in Table 1.

After conceptually decomposing HBA and mapping it to a heterogeneous platform, we choose an implementation for each of the three parts. Here, it is important to keep in mind that different processing units have different execution models. Therefore, we choose different implementations or even different algorithms depending on if one part is executed on the CPU or the GPU.

For part (1), the implementation is straightforward, since the Laplacian image computation and the identification of bright outlier pixels can be implemented using convolution. This can be implemented similarly for CPU and GPU with help of OpenCL with the only difference of cache management, which has to be done manually in the GPU implementation.

Part (2) is similar to part (1) in that the same algorithm can be used for CPU and GPU with exception of the cache management. However, while we use our own implementation for part (1), part (2) is a modification of the Canny implementation from the OpenCV[1] library. We extend OpenCV's implementation to make it possible to use relative values for the high threshold t_{hi} and the low threshold t_{lo} in contrast to the absolute values of the original implementation. This makes it easier to find fitting values for those thresholds, since the same relative threshold value will be suitable for more images, despite different absolute gradient values.

So far, we could use the same algorithm for the CPU and the GPU. This is different for the energy minimization performed in the third part. In his paper, Howe uses Boykov and Kolmogorov's graph cut algorithm [3] to perform the minimization of the energy function [18]. While this is an efficient algorithm when executed on a CPU, it is not suitable for execution on a GPU due to its global tree data structure. A graph cut algorithm, which is more suitable for execution on a GPU is the JF-cut algorithm by Peng et al. [34]. This algorithm is based on a push–relabel approach, which allows parallel local updates, in contrast to sequential updates to a global data structure, making it therefore more suitable for the GPU. For this reason, we are using these two different algorithms for CPU and GPU in our implementation.

3.2 GPU energy minimization

As mentioned before, the locally applicable *push* and *relabel* operations of the push–relabel algorithm make it suitable for GPU execution. However, Anderson and Setubal [2] have shown that a periodically executed global step, called *global relabel*, greatly improves the execution performance of the push–relabel algorithm. They also emphasize that this global step should not be used too frequently, since it can affect execution performance negatively, due to the required synchronization. Therefore, Anderson and Setubal propose to execute global relabel only after $|V|$ or $2|V|$ *push* and *relabel* operations depending on the ratio between number of edges and vertices, where $|V|$ is the number of vertices in the graph. While this is a useful heuristic, it is time consuming to keep track of the number of performed *push* and *relabel* operations in implementations for the GPU, since a large number of these operations is executed in parallel. Vineet and Narayanan [52] reduce the number of global relabel executions in their GPU implementation by executing it only after a fixed amount of iterations of *push* and *relabel* phases. However, Peng et al. [34] claim that global relabel benefits execution performance only in the first stages of their GPU implementation and can be omitted thereafter. While this is true for the type of image segmentation they analyzed, we argue that HBA requires global relabeling steps throughout the energy minimization part.

[1] http://opencv.org.

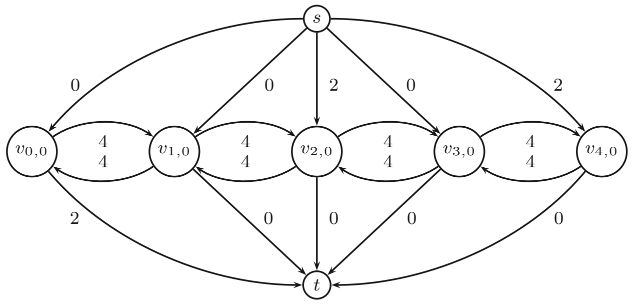

Fig. 5 Subgraph of a graph built for an $m \times n$ image representing the first five pixels of the first image row

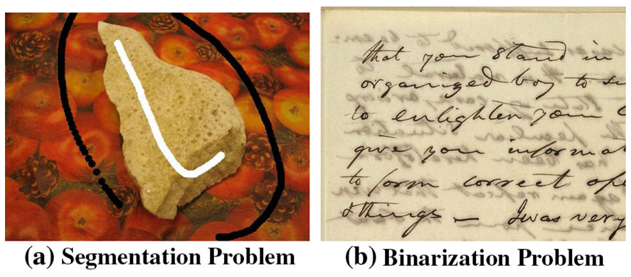

Fig. 6 Images illustrating typical image segmentation and binarization problems. Image **a** is part of a benchmark dataset made by Szeliski et al. [51]. Image **b** is part of the DIBCO 2013 dataset [39]. Additionally, image **b** illustrates bleed-through

In order to understand the difference between the problem of image segmentation Peng et al. looked at and document image binarization, we first need to understand what global relabel actually does. Take the three vertices $v_{0,0}$, $v_{2,0}$, and $v_{4,0}$ in Fig. 5, which depicts a subgraph of a graph constructed for an $m \times n$ intensity image corresponding to the first five pixels of the first image row. Remember that in the push–relabel algorithm, we imagine that each of the vertices has a water reservoir attached to it, which is located at a certain height level. According to the graph, the reservoir of $v_{0,0}$ is empty and can still forward water to the sink. However, it can only forward as much water as the reservoir of $v_{2,0}$ holds, namely 2 units of water. The reservoirs of $v_{2,0}$ and $v_{4,0}$ both hold 2 units of water, $v_{0,0}$ is their closest connection to the sink, and $v_{2,0}$ is closer to $v_{0,0}$ than $v_{4,0}$.

For a dramatic effect, you can imagine a large number of empty reservoirs, which cannot forward water to the sink, instead of the shown two vertices $v_{1,0}$ and $v_{3,0}$. The edge capacities connecting the reservoirs are sufficient to transport all water in $v_{2,0}$ and $v_{4,0}$–$v_{0,0}$. When we restrict ourselves to local *push* and *relabel* operations, we might now increase the height of $v_{2,0}$ and $v_{4,0}$ by one and start pushing the water downhill. On a local scale, however, we do not know if we should push the water in $v_{2,0}$ toward $v_{0,0}$ or toward $v_{4,0}$, since both alternatives seem equally suitable. On a global scale, we know that the closest path to the sink is via $v_{0,0}$. Therefore, global relabel, as proposed by Goldberg and Tarjan [14], computes for each reservoir the distance to the closest sink and sets the reservoir's height level accordingly. In this way, global relabel builds a ramp rising from $v_{0,0}$ via $v_{2,0}$–$v_{4,0}$ and it becomes clear for local *push* operations to push water downhill in the direction of $v_{0,0}$. Obviously, providing an initial direction toward the closest sink makes the push–relabel algorithm more efficient almost regardless of the problem it is applied to. This is the case, since we avoid locally correct, but globally wrong decisions, such as pushing water from $v_{2,0}$ toward $v_{4,0}$.

After motivating an initial global relabel step, the question is now: Why should we repeat the execution of global relabel? And why is this execution dependent on the problem at hand? Going back to our example in Fig. 5, we can see that after three *push* operations the water from $v_{2,0}$ will reach the sink t and will fill up $v_{0,0}$'s connection to the sink. Assuming that at the same time as we pushed the water from $v_{2,0}$, we also pushed the water downhill from $v_{4,0}$, the reservoir of $v_{1,0}$ will now hold 2 units of water. Even though, we know globally that the water in $v_{1,0}$ cannot reach the sink through $v_{0,0}$, the local *push* operation will still forward the water to this vertex. Only then, a local *relabel* operation will change the height level of $v_{0,0}$ to find another way to the sink. This is not a big issue if the nearest vertex with a not saturated connection to the sink lies in the same direction as $v_{0,0}$, i.e., it is close to $v_{0,0}$, such as $v_{0,1}$, which is the vertex next to $v_{0,0}$ in y direction in the complete graph. However, it becomes an issue if the nearest vertex is located in the opposite direction, for instance, $v_{9,0}$, not shown in the subgraph in Fig. 5. In the second case, local *push* and *relabel* operations would have to invert the previously built ramp one step at a time. It becomes immediately clear that global relabel would simplify this procedure by building a new ramp. Additionally, if we would have executed global relabel after the three push operations, we would not have pushed the water from $v_{1,0}$ to $v_{0,0}$, but instead, we could have started to push the water back in direction of $v_{9,0}$ directly. This may not seem significant in this small example. However, as mentioned before, there can be a large number of intermediate vertices between $v_{0,0}$, $v_{2,0}$ and $v_{4,0}$, increasing the number of superfluous *push* and *relabel* operations. Therefore, depending on the situation, it can be appropriate to repeat the execution of global relabel.

The two mentioned situations, i.e., the next closest sink connection lies in direction of the current ramp and the case it lies in the opposite direction, distinguish Peng et al.'s image segmentation problem from the binarization of document images. In the image segmentation case, as shown in Fig. 6a, we have one central object, which should be separated from the background. Therefore, we have one main region in which vertices have an excess flow and one region in which vertices can forward flow to the sink. The first region is marked with a white line in the image, while the latter region is marked with a black line. Due to these two clearly defined regions, the next closest sink connection will most likely lie in direction of the initially built ramp. In contrast, in the document image binarization case, there is no central region in which vertices with excess flow are located and no general direction toward a region of sink connected vertices. This is caused by Howe's energy function, which makes vertices of blank page regions carry excess flow, while vertices of ink or bleed-through regions are connected to the sink. With this in mind, one can see from Fig. 6b that there is no single sink region, but that vertices connected to the sink are scattered all across the image. Because of this structural difference, it is far more likely that the next closest sink connection will not lie in direction of the initially built ramp, but that a direction change is necessary. Therefore, it is beneficial for the execution performance to execute global relabel throughout HBA's execution, while this is unnecessary in the image segmentation case.

As mentioned before, it is important to perform global relabel not too frequently. Therefore, we propose a slightly simplified version of Anderson and Setubal's heuristic [2] to achieve reasonable performance using global relabel without the need of keeping track of the number of performed *push* and *relabel* operations. Instead of performing global relabel after a fixed number of iterations of *push* and *relabel* phases, as proposed by Vineet and Narayanan [52], we double the number of iterations after each execution of global relabel. In this way, we account for the fact that less and less *push* and *relabel* operations are performed with each iteration and approximate Anderson and Setubal's heuristic.

4 Parameter tuning

Another path toward improving HBA's execution performance is parameter tuning. In this section, we argue that the correct choice of HBA's parameters does not only benefit binarization performance, but also execution performance, Sect. 4.1. Furthermore, we describe a fast way to predict suitable parameters for a given document image, Sect. 4.2.

4.1 Benefits of parameter tuning

As mentioned before, the most important parameters for HBA are the Canny high threshold t_{hi} and the penalty or capacity value c. Howe argues in his paper [19] that selecting suitable parameters for a given image highly benefits the binarization performance. Additionally to this improvement of binarization quality, we argue that a fitting parameter choice benefits also the execution performance. In the following, we detail how poor choices for each of the two main parameters can negatively affect execution performance.

Canny high threshold An optimal t_{hi} value is found when the edges of most foreground elements, such as words are detected, while the edges of most background elements, such as bleed-through or stains, are ignored. Choosing a too low t_{hi} value results in selecting many edges from background objects. Since each detected edge results in a capacity of 0 in the corresponding edge of the energy function, this leads to a largely disconnected graph. The results are longer execution times for Ford–Fulkerson style algorithms as well as for graph cut algorithms, since it increases the length of the paths that flow needs to travel from the source to the sink. Additionally, such a largely disconnected graph has many narrow passages, where only a few graph edges have the capacity to forward flow, while most surrounding graph edges do not have the capacity. This leads to the situation that the edge capacities of those passages are used up fast, resulting in direction changes in the push–relabel algorithm, which are detrimental to the execution performance, as discussed in the previous section.

On the other hand, a too high t_{hi} value results in ignoring edges from foreground objects, keeping their corresponding vertices connected to the rest of the graph. Since the vertices belonging to foreground objects get high-capacity connections to the sink in the energy function, this means that more flow can be pushed from the source to the sink. This affects the execution performance negatively in Ford–Fulkerson-style algorithms, since more paths from source to sink have to be processed. In case of push–relabel algorithms, more *push* and *relabel* operations need to be executed to direct flow from many different, and potentially far away sources to the vertices of foreground objects, until the capacities of their sink or neighbor edges are consumed.

Penalty value An optimal value for c is high enough to equal out background noise, such as bleed-through or stains, but low enough to not erase faint ink strokes. Choosing a too low c value results in the situation that short paths to the sink get blocked fast, due to exhausted edge capacities. This increases the execution time of Ford–Fulkerson-style algorithms and push–relabel algorithms, due to the longer paths to the sink. Additionally, fast blocked paths lead to many direction changes in the push–relabel algorithm, which also slows down execution performance. However, if the value for c is near 0, processing will be fast, since most edges get saturated immediately not allowing any equalization. On the other hand, a too high value for c makes it almost impossible for edges to get saturated. Therefore, flow can travel over long distances to reach the sink, which leads to longer execution times. This is particularly problematic in combination with a high t_{hi} value, since then many high-capacity connections to the sink are available.

4.2 Parameter prediction

In the previous section, we discussed how choosing t_{hi} and c values lower or higher than the optimal value can negatively affect not only the binarization performance, but also the execution performance. Therefore, it is important to be able to estimate suitable parameter values for a given image. While Howe provides a parameter tuning algorithm, which works reasonably well [19], his algorithm is time consuming, since it needs to binarize the given image several times. In order to estimate HBA's parameters in a timely fashion, we propose to predict the parameters using multivariate regression based on selected image features.

Initially, we looked at 20 different image features, such as intensity mean, standard deviation, entropy or stroke width. Several of these features were inspired by the work of Dinç et al. [8], in which the authors select a binarization algorithm to binarize protein crystal images based on the image's features. Examples for those features are the mean intensity of the foreground region and the standard deviation of the background region. In order to reduce the time taken for feature extraction, we used the Breiman–Cutler permutation variable importance measure [4] for random forests to select 4 out of these 20 features. After measuring the variable importance, the four most important features were chosen for prediction. In the following, we describe those four selected features, while the other 16 features are described in "Appendix A."

Co-occurrence matrix contrast This feature is computed based on the image's co-occurrence matrix, which was first introduced by Haralick et al. [16] to extract textural features from images. The co-occurrence matrix records for each pair of intensity values, how often they occur in neighboring pixels in an image. In this context, neighboring pixels are all pixels, which are one pixel away in horizontal direction. The contrast of the co-occurrence matrix is a measure of local variations in the image.

Co-occurrence matrix homogeneity A second feature derived from the co-occurrence matrix is the image's homogeneity. This feature is a measure for the degree of smooth transitions between pixels with different intensity levels in an image.

Edge mean This feature is computed by computing the Canny edge image using a fixed value for the high threshold t_{hi}. Then, the mean intensity of the image pixels belonging to the detected edges is computed.

Background standard deviation In order to compute this feature, we use Otsu's binarization algorithm [31] to get a rough estimate of the image's foreground and background regions. Then, we compute the standard deviation of the image pixels classified as background.

Using these four image features, we use random forests to learn a mapping $\hat{f}$ from these features to suitable HBA parameters t_{hi} and c for a given image.

5 Experiment design

To verify that the proposed measures achieve our overall goal to speed up binarization without losing binarization performance, we designed several experiments. In this section, we introduce our general experiment setup (Sect. 5.1) and explain how we evaluate the execution performance of different global relabel strategies (Sect. 5.2) and different mappings of algorithm parts to CPU and GPU (Sect. 5.3). Additionally, we describe how we measure the importance of parameter tuning (Sect. 5.4) and how we evaluate the performance of our parameter prediction algorithm (Sect. 5.5).

5.1 Experiment setup

All experiments were performed on a computer with an Intel i7-6700K quad-core CPU @ 4.00 GHz, 32 GB DDR4 RAM and an Nvidia GeForce GTX 980. To execute parts of HBA on CPU and GPU, we use our own implementation of HBA, which uses the programming interface specified in the OpenCL standard [49] for executing code on CPU and GPU.

The document image binarization contest datasets from DIBCO 2009 [12], H-DIBCO 2010 [36], DIBCO 2011 [37], H-DIBCO 2012 [38], DIBCO 2013 [39], H-DIBCO 2014 [30] and H-DIBCO 2016 [40], were used in all experiments. These datasets contain 86 images in total, of which 65 images are handwritten and 21 images are printed documents. The images of printed documents are from the DIBCO 2009, DIBCO 2011, and DIBCO 2013 dataset.

For all experiments measuring execution time, we perform four consecutive time measurements and report the average time for the last three time measurements. In this way, we exclude the initial cold run and obtain more stable measurement results, which are not skewed by operating system caching. Furthermore, by averaging over the three warm runs, we obtain a time measurement, which is less affected by other operating system tasks running at the same time. Additionally, it should be noted that we only measure the execution time for the binarization algorithm, excluding the time for loading and storing the respective image.

We measure the binarization performance using F-Measure, pseudo-F-Measure (F_{ps}), the peak-signal-to-noise ratio (PSNR) and the distance reciprocal distortion metric (DRD), which are commonly used in binarization competitions [30,38–40]. While F-Measure assesses binarization quality in terms of misclassified pixels regardless of the pixels' location, F_{ps} weights pixel misclassifications differently to emphasize readability. PSNR measures the similarity between ground truth image and respective binarized image and DRD measures visual distortions in the binarized image.

As we show in Sect. 6.3, the binarization parameters t_{hi} and c have a significant impact on binarization and execution performance. To allow an unbiased comparison of execution times, we use the binarization parameters found by Howe's parameter tuning algorithm [19] for all our time measurements.

5.2 Evaluating global relabel strategies

For our evaluation, we consider four different global relabel strategies for the push–relabel algorithm used on the GPU. Those strategies are None, Initial, Fixed, and Exponential as described in the following.

None We perform the energy minimization using the push–relabel algorithm without any global relabel step.

Initial As Peng et al. [34], we perform only one global relabel step in the beginning of the push–relabel algorithm.

Fixed As Vineet and Narayanan [52], we perform a global relabel step in fixed intervals with every eighth iteration of the push–relabel GPU algorithm.

Exponential As described before, we double the interval size after each global relabel step starting with an interval of eight iterations.

For each of these four strategies, we measure the execution time for the images of all datasets using the respective binarization parameters found using Howe's parameter tuning algorithm. For this evaluation, all parts of HBA are executed on the GPU.

5.3 Evaluating algorithm mappings

As mentioned before, there are eight different possibilities to map HBA onto a heterogeneous platform consisting of one CPU and one GPU. To identify the most suitable mapping, we measure the execution times for the images of all datasets. Based on our findings presented in Sect. 6.1, we use the global relabel strategy Fixed in all measurements.

Apart from the execution performance, we need to ensure that each of the eight mappings produces a comparable binarization performance. Therefore, we compare the obtained binarization results with their respective ground truth images using the four previously described measures. Additionally, we compare our results with Howe's MATLAB implementation,[2] executed on MATLAB version R2016b. This implementation incorporates Boykov and Kolmogorov's method for efficient graph cuts [3], as well as Kohli and Torr's approach for reusing minimization results after modifying the graph [25].

5.4 Evaluating parameter impact

In order to evaluate which impact HBA's most important parameters t_{hi} and c have on the binarization and execution performance, we binarize the images from all datasets using different values for these parameters. Here, we let t_{hi} range from 0.15 to 0.65 in steps of 0.1 and c from 20 to 1545 in steps of 25. For all combinations of these parameter values, we measure the execution time in seconds and the binarization performance using F-Measure.

Based on the results from Sects. 6.1 and 6.2, we execute all parts of the binarization algorithm on the GPU and use the global relabel strategy Fixed for all measurements.

5.5 Evaluating prediction performance

We evaluate the performance of our proposed parameter tuning approach in terms of binarization performance of the chosen parameters, as well as execution time. For our approach, we use the DIBCO 2013 dataset as training dataset to build ten random forest models for multivariate regression using ten different, randomly chosen seed values. In this way, we can show that our approach is stable against random variations in the construction of the random forests model. The training dataset was built by choosing for each image in DIBCO 2013 the binarization parameters t_{hi} and c, which produced the best binarization performance with respect to F-Measure in the previous experiment (Sect. 5.4). For building the models, we used the default settings of the *randomForestSRC* R package [21–23].

We use each of the 10 models to predict the binarization parameters for all images in the remaining datasets, DIBCO 2009, H-DIBCO 2010, DIBCO 2011, H-DIBCO 2012, H-DIBCO 2014, and H-DIBCO 2016. The execution time is measured for each image including feature extraction, parameter prediction, and image binarization. Here, we measure the execution time including loading and storing the respective image. Similarly, we measure the execution time for Howe's parameter tuning algorithm [19] from the time the image starts loading until the binarized image is stored.

In order to make it easier to compare our parameter tuning approach with Howe's algorithm, we use Howe's MATLAB implementation for image binarization, but we use the parameters predicted by our random forests model. Therefore, the only difference in binarization performance can be achieved by the chosen binarization parameters. To compare the binarization performance of Howe's algorithm and our approach,

[2] http://www.cs.smith.edu/~nhowe/research/code/.

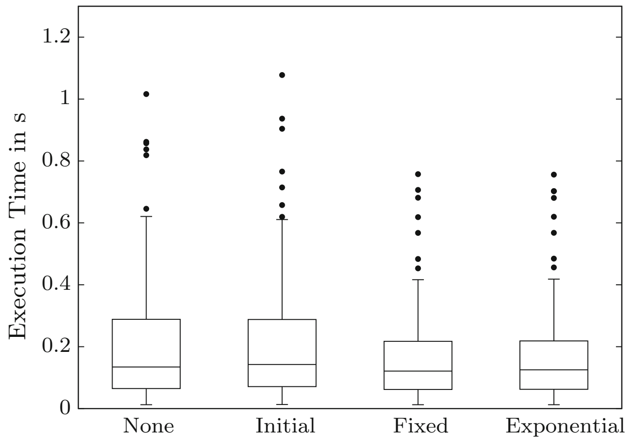

Fig. 7 Execution times for the global relabel strategies None, Initial, Fixed, and Exponential for the images of all datasets. Boxes illustrate the first, second, and third quartile; whiskers indicate the lowest and the highest value within the 1.5 interquartile range; dots mark outliers outside this range

we use the four previously described measures for each binarized image.

6 Results and analysis

In this section, we present the results obtained by carrying out the experiments described in the previous section and analyze those results. We evaluate the performance of the different global relabel strategies (Sect. 6.1) and the different algorithm mappings (Sect. 6.2). Furthermore, we show what impact the two parameters t_{hi} and c have on the binarization and execution performance (Sect. 6.3) and evaluate the performance of our parameter tuning algorithm in comparison to Howe's tuning algorithm (Sect. 6.4).

6.1 Performance of global relabel strategies

Figure 7 shows the measured execution times for the images from all datasets for all four global relabel strategies. From these box plots, we can see that there is a slight difference between the execution times of the strategies None and Initial compared to Fixed and Exponential.

Therefore, we conduct a repeated measures analysis of variances (ANOVA) to analyze this difference more closely. This is a suitable test in this situation, since we are analyzing more than two dependent samples and the depended variable, i.e., execution time, is continuous and approximately normally distributed. We avoid problems with possible violations of the sphericity assumption by building a multilevel linear model with maximum likelihood estimation for ANOVA using the *nlme* R package [35].

The performed ANOVA test indicates statistically significant differences in the execution times of the four analyzed global relabel strategies at the $p < 0.05$ level [$F(3, 255) = 15.19843$, $p < 0.0001$]. Here, $numDF$ and $denDF$ in $F(numDF, denDF)$ indicate the numerator degrees of freedom ($numDF$) and the denominator degrees of freedom ($denDF$) of the F-statistic, respectively. Both degrees of freedom are estimated by the *nlme* R package based on the input data.

The next step in our analysis is to identify strategies which differ significantly in their execution times. Therefore, we use the *multcomp* R package [17] to perform Tukey's test as post hoc analysis. This test shows that there is no statistically significant difference between the global relabel strategies None and Initial (p-value $= 0.926$), as well as between the strategies Fixed and Exponential (p-value $= 0.843$). However, clear statistically significant differences exist between all other combinations of these strategies with p-values < 0.001.

Based on the box plots in Fig. 7 and the results of our statistical analysis, we can see that repeated global relabel steps improve the execution performance of the push–relabel algorithm in the case of document image binarization. This supports our hypothesis that performing global relabel steps throughout the execution of HBA's energy minimization part improves the execution performance. However, none of the two analyzed strategies, Fixed and Exponential, which repeatedly perform global relabel steps, performs significantly better than the other. Therefore, we decide which strategy to use in further experiments simply based on the lowest median value. In this case, this strategy is the global relabel strategy Fixed.

6.2 Performance of algorithm mappings

As described in Sect. 5.3, we measure the execution times for binarizing each of the images in all analyzed datasets to find the most suitable mapping of HBA's parts to CPU and GPU. Figure 8 shows the measurement results for all mappings as shown in Table 1, as well as the time measurements for Howe's original MATLAB implementation (MATLAB). Based on these box plots, one can see that there are certain differences between the execution times of different mappings and the reference implementation.

To analyze these differences in more detail, we perform a repeated measures ANOVA, as in the previous section. This test indicates statistically significant differences in the execution times for the eight analyzed mappings and the reference implementation at the $p < 0.05$ level [$F(8, 619) = 34.91975$, $p < 0.0001$]. Therefore, we perform Tukey's test as post hoc analysis, to identify between which mappings there are differences in execution performance.

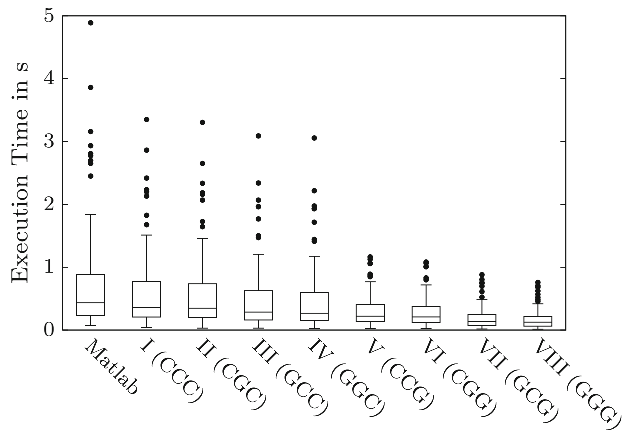

Fig. 8 Execution times for all eight different mappings of HBA to CPU and GPU for the images of all datasets. Boxes illustrate the first, second and third quartile; whiskers indicate the lowest and the highest value within the 1.5 interquartile range; dots mark outliers outside this range

Tukey's test indicates that there is a statistical significant difference at the $p < 0.05$ level between implementations of HBA, which perform the energy minimization step on the CPU, i.e., the reference implementation and mapping I, II, III, and IV, and implementations which perform this step on the GPU, i.e., mapping V, VI, VII and VIII. Additionally, we can see statistically significant differences between mappings, which perform the Laplacian and Canny edge detection on the CPU, i.e., the reference implementation and mapping I and V, and mappings which perform these computations on the GPU, i.e., mapping IV and VIII. Apart from those differences, there are no other statistically significant differences between mappings. From these results and the box plots in Fig. 8, one can see that mapping all HBA parts to the GPU yields the best execution performance. With this mapping, it is possible to speed up the image binarization on average by a factor of 3.5 compared to executing the whole algorithm on the CPU.

In order to ensure that the achieved speedup is not gained by decreasing binarization performance, we measure the binarization performance for the images of all datasets for all mappings, as well as the reference implementation using the four chosen binarization quality measures. As for the analysis of the execution time, we perform a repeated measures ANOVA to identify possible differences in binarization performance between the different mappings and the reference implementation for each quality measure. From the outcome of this test, as shown in Table 2, we can see that there is no statistically significant difference at the $p < 0.05$ level between any of the mappings and the reference implementation when measuring binarization performance with PSNR or DRD. However, when measuring the binarization performance using F-Measure or F_{ps}, we can observe a statistically significant difference in binarization performance.

Table 2 Results of the ANOVA test for the four used binarization quality measures for all eight mappings and the reference implementation

Measure	$F(8, 680)$	p
F-Measure	2.185	0.0268
F_{ps}	6.396	< 0.0001
PSNR	0.3718	0.9355
DRD	1.58421	0.1259

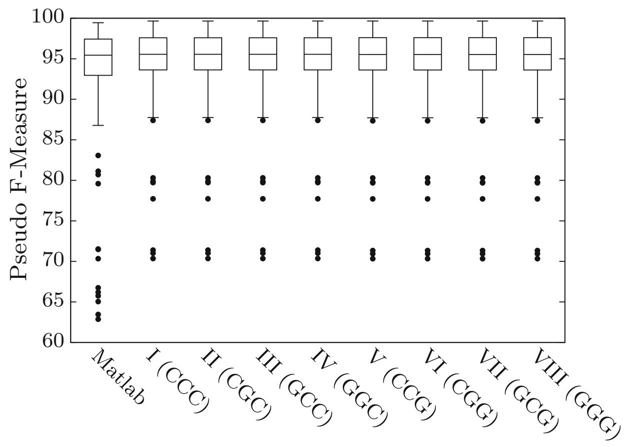

Fig. 9 F_{ps} results for all 8 different mappings of HBA to CPU and GPU for the images of all datasets. Boxes illustrate the first, second, and third quartile; whiskers indicate the lowest and the highest value within the 1.5 interquartile range; dots mark outliers outside this range

We analyze this difference by performing Tukey's test as post hoc test. From those test results, we can see that there is a statistically significant difference only between the reference implementation and the different mappings, but not among the mappings, for both binarization measures. Therefore, we can conclude that the speedup was not achieved by decreasing binarization performance. This is also shown in Fig. 9, which shows the F_{ps} results for the images of all datasets for the different mappings and the reference implementation. The shown box plots also indicate that the found difference between the binarization performance of the reference implementation and the different mappings is caused by the fact that the reference implementation produces worse binarization results compared to our implementation for the chosen binarization parameters.

6.3 Parameter impact

By varying the binarization parameters t_{hi} and c for the images from all datasets, we can observe that all images tend to have an approximately concave response surface with respect to binarization performance. This makes these parameters particularly suitable for parameter tuning, since small deviations from an "optimal" parameter setting cause only

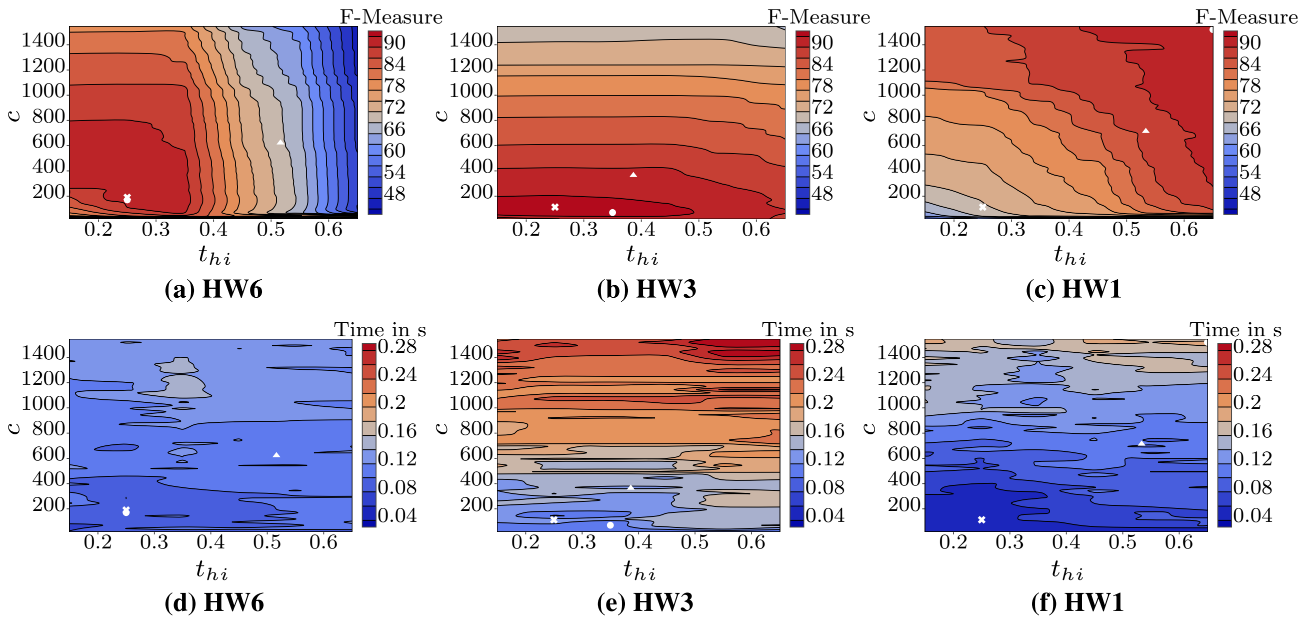

Fig. 10 Images illustrating the response surface for binarization performance (top row) and execution performance (bottom row) for different parameter settings of t_{hi} and c for selected images from the DIBCO 2011 dataset [37]. The different markers show the "optimal" parameter setting (circle), the parameter setting chosen by Howe's tuning algorithm [19] (cross) and our tuning algorithm (triangle)

small degradations in binarization performance. The approximately concave response surfaces is shown in Fig. 10a, b, c, which shows the obtained F-Measure values for different parameter settings for images from the DIBCO 2011 dataset. One should note that while these three images were chosen for illustration purposes, their response surfaces show characteristics, which can be found in all analyzed images.

When comparing the response surfaces for the binarization performance with the surfaces for the execution performance (Fig. 10d, e, f), one can see that the parameter settings yielding the best binarization performance, as indicated by the white circle, also tend to have low execution times. Therefore, we analyze the correlation between binarization performance and execution time for all images using the Pearson product-moment correlation coefficient. In order to make the F-Measure and time measurements of the different images comparable, we apply standard score normalization to all measurements of one image. By using this normalized data, we see that there is a clear negative correlation between binarization performance and execution performance [$r = -0.194$, $n = 31{,}992$, $p < 0.0001$]. This supports our initial hypothesis that well chosen binarization parameters in terms of binarization performance also yield reasonable execution performance.

6.4 Prediction performance

As described in Sect. 5.5, we use the results of the previous section to construct ten random forest models for prediction of the binarization parameters. In order to ensure that there are no statistically significant differences in binarization performance between the binarization parameters predicted by each of these models, we perform a repeated measures ANOVA, as described in Sect. 6.1. Additionally, we include in this analysis the measured binarization performance of Howe's parameter tuning algorithm.

Table 3 Results of the ANOVA test for the four used binarization quality measures for the 10 random forest models and the reference implementation

Measure	$F(10, 715)$	p
F-Measure	0.065	1.0000
F_{ps}	0.219	0.9946
PSNR	0.1291	0.9994
DRD	0.00715	1.0000

The performed ANOVA test shows that there is no statistically significant difference between the predictions of any of the random forest models and Howe's tuning algorithm at the $p < 0.05$ level for any of the four used binarization quality measures, as shown in Table 3. This is shown as well in Fig. 10, which shows for three selected images which parameters were chosen by Howe's algorithm (cross) and by our prediction (triangle), as well as which parameter settings are "optimal" (circle) for the respective image. Optimal parameter settings are the settings resulting in the highest F-Measure value, which were found during the previ-

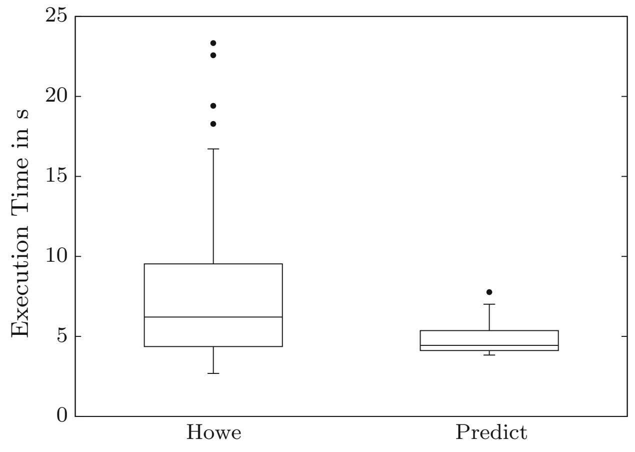

Fig. 11 Execution time in seconds for Howe's parameter tuning algorithm and our prediction algorithm for the images from all datasets, except DIBCO 2013, which was used as training set. Boxes illustrate the first, second and third quartile; Whiskers indicate the lowest and the highest value within the 1.5 interquartile range; Dots mark outliers outside this range

ous experiment (Sect. 6.3). From this image, one can see that the parameters chosen by Howe's algorithm and the parameters found by our algorithm are close to each other for all images.

Next, we analyze the execution times for our ten random forest models and for Howe's parameter tuning algorithm. Figure 11 shows the execution time of Howe's algorithm for the images from all datasets, except DIBCO 2013, which was used for training, compared to the random forests model, which had the highest mean execution time among all ten random forest models. From these box plots, one can see that there is a clear difference in execution time between Howe's algorithm and our prediction algorithm.

To confirm this observation, we perform a repeated measures ANOVA for the execution times of all ten random forest models and Howe's tuning algorithm. This analysis shows a statistically significant difference in execution times at the $p < 0.05$ level $[F(10, 690) = 47.7139,\ p < 0.0001]$. The Tukey test shows that the detected differences in execution times are only between Howe's tuning algorithm and the random forest models, with p values < 0.0001, but not between the random forest models.

Therefore, we can conclude that our prediction algorithm chooses binarization parameters, which produce comparable binarization performance to Howe's algorithm, while executing on average 1.7 times faster. This speedup can be explained by the fact that Howe's tuning algorithm has to perform the computationally expensive binarization step several times, while our approach requires only one binarization step per image. Thus, the average speedup increases to 2.5, when looking only at the larger images from the analyzed datasets, with no statistically significant difference to the binarization performance of Howe's tuning algorithm in any of the four measures. Here, the larger images are all images from the analyzed datasets with an area at or above the third quartile.

7 Related work

In this section, we discuss previous work related to improved execution performance of image binarization algorithms due to their implementation (Sect. 7.1), as well as work related to automatic parameter tuning for image binarization algorithms (Sect. 7.2).

7.1 Efficient image binarization

Most research on image binarization has focused more on the quality of the binarization result than on the execution time. This is also reflected in the numerous binarization competitions, in which the execution time is either not reported [12,39] or not used as evaluation criterium [30, 38]. Nevertheless, there have been a few implementations of common binarization algorithms utilizing the parallel computing capabilities of GPUs. For example, Singh et al. proposed GPU implementations for the binarization algorithms by Niblack, Otsu and Sauvola et al. [45–47] to speed up these commonly used algorithms. In their implementations, Singh et al. treat image binarization as a data parallel problem, where each image pixel is classified as foreground or background pixel by its own GPU thread. With this parallelization approach, they report up to 20 times faster binarization in comparison to a single threaded implementation of the respective algorithm. Recently, Chen et al. [6] proposed another GPU implementation of an algorithmically improved version of Sauvola's binarization algorithm using integral images, as proposed by Shafait et al. [44]. With this implementation, Chen et al. report up to 38 times faster execution performance compared to a CPU implementation. Furthermore, GPU implementations of other binarization algorithms were proposed by Peña-Cantillana et al. [33] and Soua et al. [48].

The main difference between the approaches mentioned above and our approach is that we analyze potential benefits of mapping different parts of a binarization algorithm to different processors of a heterogeneous platform. This is important to identify how the available compute platform can be utilized most efficiently.

Work most closely related to our approach is described in the paper by Westphal et al. [54], which is a work-in-progress report describing the general idea of mapping HBA to a heterogeneous platform including preliminary results for execution and binarization performance. This paper extends that report by analyzing the importance of global relabel steps during the energy minimization part of HBA, as well as by exploring the relation between the choice of HBA's parameters and the execution time. Furthermore, we propose an efficient way of choosing these parameters and analyze the obtained results for all performed experiments more extensively.

7.2 Automatic parameter tuning

Many binarization algorithms, such as the algorithm by Sauvola and Pietikäinen [43], Kim et al. [24] or Su et al. [50], have parameters, which need to be fine-tuned depending on a given image to achieve their upper bound binairzation performance. This fine-tuning is normally done manually for a sample of the document images to binarize and is then fixed to the best parameter setting.

This required manual tuning is a limitation of these approaches, which adds extra work to other document analysis tasks, which use image binarization as one preprocessing step. Additionally, the fixed parameter setting might not be optimal for all images within one dataset. For this reason, Gatos et al. [13] and Afzal et al. [1] propose different binarization algorithms, which do not require manual parameter tuning.

An alternative to parameterless binarization algorithms is the approach by Mesquita et al. [27], which tunes the parameters for Mesquita et al.'s perception of objects by distance (POD) preprocessing step for binarization [26]. The parameter tuning is performed using the Iterated F-Race approach on a training dataset to find a suitable parameter configuration to be used for all images in the target dataset. Here, it is important that the test dataset is representative of the target dataset to find a reasonable parameter configuration. This approach differs from our own approach in that it does not tune the parameters of the used binarization algorithm directly and that it does not adapt the parameter configuration to the specific characteristics of the currently binarized image.

Another approach to address the parameter tuning problem was taken by Cheriet et al. [7], which is more closely related to our own approach. In their paper, Cheriet et al. propose a machine learning framework to add automatic parameter tuning to any image binarization algorithm. The core of their framework is a set of image features, which they use as input to several univariate models built for each binarization parameter with support vector regression. In contrast to the approach by Cheriet et al., we use a different and smaller set of image features, which is used to build a random forests model for multivariate regression. This smaller set of features is beneficial for execution performance, since less time is spent on extracting different features from an image.

8 Conclusions and future work

In this paper, we have shown that the execution performance of a state-of-the-art binarization algorithm can be considerably improved with help of heterogeneous computing and prediction-based parameter tuning, while not affecting the binarization performance.

After analyzing different ways of mapping parts of Howe's binarization algorithm (HBA) onto CPU and GPU, we come to the conclusion that the whole algorithm should be executed on the GPU to achieve on average a 3.5 times higher execution performance compared to executing the complete algorithm on the CPU.

We have also presented a new parameter tuning algorithm for HBA based on multivariate regression using random forests, which chooses parameter values producing comparable binarization performance to Howe's parameter turning algorithm. However, we have shown, that our algorithm performs on average 1.7 times faster than Howe's original algorithm when considering all images from the analyzed datasets, and 2.5 times faster when considering only large images, i.e., images with an area at or above the third quartile. The main reason is that we only need to binarize a given image once in contrast to Howe's approach.

When comparing the execution time improvements achieved by heterogeneous computing and parameter tuning in absolute numbers, it becomes clear that the improvement of the parameter tuning algorithm has the greater impact on the overall execution performance. While we can save on average 0.44 s using heterogeneous computing, we save 3.22 s using our parameter tuning algorithm, or 8.73 s when considering only large images. Going back to our initial example with 50 million images, this improvement translates to 6 years of saved execution time for comparatively small images or 14 years for images with sizes similar to the largest images in the analyzed datasets.

Apart from these two main contributions to the execution performance of HBA, we have also shown that the continuous application of global relabel steps during the push–relabel algorithm is important for the execution performance when dealing with document images. However, our results do not show any significant differences between the two analyzed global relabel strategies, which performed global relabel steps either in fixed intervals or in exponentially growing intervals.

Finally, we have shown that there is a statistically significant negative correlation between execution time and binarization performance for different parameter settings of HBA. This is an interesting finding, since it means that it is sufficient to tune HBA's parameters with respect to binarization performance to achieve also a reasonable execution performance of the algorithm.

In future work, it may be interesting to investigate the possibility to improve the execution performance of the binarization of a batch of images by interleaving the binarization steps for these images. This could lead to potential speedups, since the Canny edge detection for the next image could run on the CPU in parallel to the energy minimization of the previous image, running on the GPU.

Another direction for future work would be to further improve the prediction-based parameter tuning. Since the current implementation of the feature extraction and prediction process is not optimized for fast execution, it would be interesting to investigate how much more time can be saved when improving the execution performance of these two steps.

Appendix A: image features

In this section, we describe the 16 image features, which have been considered as base for our parameter prediction approach, described in Sect. 4.2, but which have not been chosen. We describe these 16 features in the following.

Intensity mean This feature calculates the mean over all intensity values in the analyzed grayscale image.

Intensity standard deviation Similar to the previous feature, this feature is computed over all intensity values in the grayscale image, deriving their standard deviation.

Co-occurrence matrix correlation We derive this feature from the image's co-occurrence matrix, which was first introduced by Haralick et al. [16]. It measures the linear dependency between the co-occurring intensity values and thus assesses the general repetitiveness of certain patterns in the image. While repeated smooth transitions between intensity values lead to a positive correlation value, repeated jumps between intensity values lead to a negative correlation value.

Co-occurrence matrix energy As the previous feature, this feature is derived from the image's co-occurrence matrix. It is also known as uniformity and is a measure for the amount of reoccurring transitions between intensity levels. A high energy indicates that there are only a few, often reoccurring transitions between intensity levels. Thus, a constant image has the highest possible energy.

Number of connected components This feature counts the number of connected components above a certain size. The connected components are identified based on the binarized image, which was produced using Otsu's binarization algorithm [31].

Average connected components size For this feature, the connected components, which were identified for the previous feature, are used to compute this feature by averaging over the different connected component sizes.

Normalized connected components size The aim of this feature is to make the previous feature comparable between images by normalizing the average connected components size by image size.

Image size While not a particularly relevant feature in itself, the image size might help to put the values of other features into perspective, making different images better comparable.

Foreground mean To compute the foreground mean, we apply Otsu's binarization algorithm [31] to first get an estimated separation of foreground and background. Then, we compute the mean over the intensities of all pixels classified as foreground by Otsu's algorithm.

Background mean As for the previous feature, the image's foreground and background separation is estimated and then the intensity mean over all background pixels is computed.

Mean ratio In order to express how different estimated foreground pixels are from estimated background pixels, we compute the ratio between the previous two features, i.e., foreground mean and background mean.

Foreground standard deviation The foreground standard deviation is computed by estimating the separation between foreground and background pixels using Otsu's binarization algorithm. Then, the standard deviation over all detected foreground pixels is computed.

Number of edge pixels Using Canny's edge detection algorithm [5] with fixed parameters, we estimate the borders between foreground and background. For this feature, we simply count the number of pixels, which constitute this border.

Normalized number of edge pixels This feature normalizes the found number of edge pixels by the image size to make different images better comparable with respect to their edge pixel number.

Average line width This feature estimates the average width of all foreground elements in an image based on foreground elements identified using Otsu's binarization algorithm.

Entropy The image's entropy measures the degree of randomness present in an image. Therefore, it is another measure for the image's structure.

References

1. Afzal, M.Z., Pastor-Pellicer, J., Shafait, F., Breuel, T.M., Dengel, A., Liwicki, M.: Document image binarization using LSTM: a sequence learning approach. In: Proceedings of the 3rd International Workshop on Historical Document Imaging and Processing, pp. 79–84. ACM (2015)
2. Anderson, R.J., Setubal, J.C.: Goldberg's algorithm for maximum flow in perspective: a computational study. Netw. Flows Matching DIMACS Ser. Discrete Math. Theor. Comput. Sci. **12**, 1–18 (1993)
3. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Trans. Pattern Anal. Mach. Intell. **26**(9), 1124–1137 (2004)
4. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
5. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **PAMI–8**(6), 679–698 (1986)
6. Chen, X., Gao, Y., Huang, Z.: CUDA-accelerated fast Sauvola's method on Kepler architecture. Multimed. Tools Appl. **74**(24), 11809–11820 (2015)
7. Cheriet, M., Moghaddam, R.F., Hedjam, R.: A learning framework for the optimization and automation of document binarization methods. Comput. Vis. Image Underst. **117**(3), 269–280 (2013)
8. Dinç, I., Dinç, S., Sigdel, M., Sigdel, M., Pusey, M.L., Aygün, R.S.: Super-thresholding: supervised thresholding of protein crystal images. IEEE/ACM Trans. Comput. Biol. Bioinform. **14**, 986–998 (2016)
9. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM (JACM) **19**(2), 248–264 (1972)
10. Ford Jr., L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton (1962)
11. Frinken, V., Fischer, A., Martínez-Hinarejos, C.D.: Handwriting recognition in historical documents using very large vocabularies. In: Proceedings of the 2nd International Workshop on Historical Document Imaging and Processing, pp. 67–72. ACM (2013)
12. Gatos, B., Ntirogiannis, K., Pratikakis, I.: DIBCO 2009: document image binarization contest. Int. J. Doc. Anal. Recognit. (IJDAR) **14**(1), 35–44 (2011)
13. Gatos, B., Pratikakis, I., Perantonis, S.J.: Adaptive degraded document image binarization. Pattern Recognit. **39**(3), 317–327 (2006)
14. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM (JACM) **35**(4), 921–940 (1988)
15. Greig, D.M., Porteous, B.T., Seheult, A.H.: Exact maximum a posteriori estimation for binary images. J. R. Stat. Soc. Ser. B (Methodological) **51**(2), 271–279 (1989)
16. Haralick, R.M., Shanmugam, K., Dinstein, I.: Textural features for image classification. IEEE Trans. Syst. Man Cybern. **SMC–3**(6), 610–621 (1973)
17. Hothorn, T., Bretz, F., Westfall, P.: Simultaneous inference in general parametric models. Biom. J. **50**(3), 346–363 (2008)
18. Howe, N.R.: A laplacian energy for document binarization. In: 2011 International Conference on Document Analysis and Recognition (ICDAR), pp. 6–10. IEEE (2011)
19. Howe, N.R.: Document binarization with automatic parameter tuning. Int. J. Doc. Anal. Recognit. (IJDAR) **16**(3), 247–258 (2013)
20. Ishikawa, H., Geiger, D.: Segmentation by grouping junctions. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 125–131. IEEE (1998)
21. Ishwaran, H., Kogalur, U.: Random survival forests for R. R News **7**(2), 25–31 (2007). https://CRAN.R-project.org/doc/Rnews/
22. Ishwaran, H., Kogalur, U.: Random Forests for Survival, Regression and Classification (RF-SRC) (2016). https://cran.r-project.org/package=randomForestSRC. R package version 2.4.0
23. Ishwaran, H., Kogalur, U., Blackstone, E., Lauer, M.: Random survival forests. Ann. Appl. Statist. **2**(3), 841–860 (2008). http://arXiv.org/abs/0811.1645v1
24. Kim, I.K., Jung, D.W., Park, R.H.: Document image binarization based on topographic analysis using a water flow model. Pattern Recognit. **35**(1), 265–277 (2002)
25. Kohli, P., Torr, P.H.S.: Efficiently solving dynamic markov random fields using graph cuts. Tenth IEEE Int. Conf. Comput. Vis. (ICCV) **2**, 922–929 (2005). https://doi.org/10.1109/ICCV.2005.81
26. Mesquita, R.G., Mello, C.A., Almeida, L.: A new thresholding algorithm for document images based on the perception of objects by distance. Integr. Comput. Aided Eng. **21**(2), 133–146 (2014)
27. Mesquita, R.G., Silva, R.M., Mello, C.A., Miranda, P.B.: Parameter tuning for document image binarization using a racing algorithm. Expert Syst. Appl. **42**(5), 2593–2603 (2015)
28. Mioulet, L., Bideault, G., Chatelain, C., Paquet, T., Brunessaux, S.: Exploring multiple feature combination strategies with a recurrent neural network architecture for off-line handwriting recognition. In: IS&T/SPIE Electronic Imaging, pp. 94,020F–94,020F. International Society for Optics and Photonics (2015)
29. Niblack, W.: An Introduction to Digital Image Processing. Prentice-Hall, Englewood Cliffs (1986)
30. Ntirogiannis, K., Gatos, B., Pratikakis, I.: ICFHR2014 competition on handwritten document image binarization (H-DIBCO 2014). In: 14th International Conference on Frontiers in Handwriting Recognition (ICFHR), pp. 809–813. IEEE (2014)
31. Otsu, N.: A threshold selection method from gray-level histograms. Automatica **11**(285–296), 23–27 (1979)
32. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design. Morgan Kaufmann, Burlington (2013)
33. Peña-Cantillana, F., Díaz-Pernil, D., Berciano, A., Gutiérrez-Naranjo, M.A.: A parallel implementation of the thresholding problem by using tissue-like p systems. In: International Conference on Computer Analysis of Images and Patterns, pp. 277–284. Springer (2011)
34. Peng, Y., Chen, L., Ou-Yang, F.X., Chen, W., Yong, J.H.: JF-cut: a parallel graph cut approach for large-scale image and video. IEEE Trans. Image Process. **24**(2), 655–666 (2015)
35. Pinheiro, J., Bates, D., DebRoy, S., Sarkar, D., R Core Team: nlme: linear and nonlinear mixed effects models (2016). http://CRAN.R-project.org/package=nlme. R package version 3.1-128
36. Pratikakis, I., Gatos, B., Ntirogiannis, K.: H-DIBCO 2010-handwritten document image binarization competition. In: International Conference onFrontiers in Handwriting Recognition (ICFHR), pp. 727–732. IEEE (2010)
37. Pratikakis, I., Gatos, B., Ntirogiannis, K.: ICDAR 2011 document image binarization contest (DIBCO 2011). In: International Conference on Document Analysis and Recognition, pp. 1506–1510 (2011). https://doi.org/10.1109/ICDAR.2011.299
38. Pratikakis, I., Gatos, B., Ntirogiannis, K.: ICFHR 2012 competition on handwritten document image binarization (H-DIBCO 2012). In: 2012 International Conference on Frontiers in Handwriting Recognition (ICFHR), pp. 817–822. IEEE (2012)
39. Pratikakis, I., Gatos, B., Ntirogiannis, K.: ICDAR 2013 document image binarization contest (DIBCO 2013). In: 12th International

Conference on Document Analysis and Recognition (ICDAR), pp. 1471–1476. IEEE (2013)

40. Pratikakis, I., Zagoris, K., Barlas, G., Gatos, B.: ICFHR2016 handwritten document image binarization contest (H-DIBCO 2016). In: 15th International Conference on Frontiers in Handwriting Recognition (ICFHR), pp. 619–623 (2016). https://doi.org/10.1109/ICFHR.2016.0118
41. Roy, S., Cox, I.J.: A maximum-flow formulation of the N-camera stereo correspondence problem. In: Sixth International Conference on Computer Vision, pp. 492–499. IEEE (1998)
42. Rusiñol, M., Lladós, J.: Boosting the handwritten word spotting experience by including the user in the loop. Pattern Recognit. **47**(3), 1063–1072 (2014)
43. Sauvola, J., Pietikäinen, M.: Adaptive document image binarization. Pattern Recognit. **33**(2), 225–236 (2000)
44. Shafait, F., Keysers, D., Breuel, T.M.: Efficient implementation of local adaptive thresholding techniques using integral images. In: Electronic Imaging, pp. 681,510–681,510. International Society for Optics and Photonics (2008)
45. Singh, B.M., Sharma, R., Mittal, A., Ghosh, D.: Parallel implementation of Niblack's binarization approach on CUDA. Int. J. Comput. Appl. **32**, 22–27 (2011)
46. Singh, B.M., Sharma, R., Mittal, A., Ghosh, D.: Parallel implementation of Otsu's binarization approach on GPU. Int. J. Comput. Appl. **32**, 16–21 (2011)
47. Singh, B.M., Sharma, R., Mittal, A., Ghosh, D.: Parallel implementation of Souvola's binarization approach on GPU. Int. J. Comput. Appl. **32**(2), 28–33 (2011)
48. Soua, M., Kachouri, R., Akil, M.: GPU parallel implementation of the new hybrid binarization based on Kmeans method (HBK). J. Real-Time Image Process. 1–15 (2014)
49. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(1–3), 66–73 (2010)
50. Su, B., Lu, S., Tan, C.L.: Robust document image binarization technique for degraded document images. IEEE Trans. Image Process. **22**(4), 1408–1417 (2013)
51. Szeliski, R., Zabih, R., Scharstein, D., Veksler, O., Kolmogorov, V., Agarwala, A., Tappen, M., Rother, C.: A comparative study of energy minimization methods for markov random fields. In: 9th European Conference on Computer Vision, pp. 16–29. Springer, Heidelberg (2006)
52. Vineet, V., Narayanan, P.: CUDA cuts: fast graph cuts on the GPU. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 1–8. IEEE (2008)
53. Wang, P., Eglin, V., Garcia, C., Largeron, C., Lladós, J., Fornés, A.: A coarse-to-fine word spotting approach for historical handwritten documents based on graph embedding and graph edit distance. In: 22nd International Conference on Pattern Recognition (ICPR), pp. 3074–3079. IEEE (2014)
54. Westphal, F., Grahn, H., Lavesson, N.: Efficient binarization for historical document analysis (2016). Family History Technology Workshop (FHTW 2016)