Comparative Analysis of Image Binarization Algorithms: A Parallelized Approach

Algorithm Engineering 2024 Project Paper

Rayk Kretzschmar
Friedrich Schiller University Jena
Germany
rayk.kretzschmar@uni-jena.de

Robert Josef Domogalla
Friedrich-Schiller-University Jena
Germany
robert.josef.domogalla@uni-jena.de

Yassine Jabrane
Friedrich Schiller University Jena
Germany
yassine.jabrane@uni-jena.de

ABSTRACT

This paper represents the culmination of the "Algorithm Engineering" course offered at Friedrich Schiller University in Jena, and it delves into the critical area of image binarization optimization. The significance of this study lies in its dual focus: enhancing the quality of binarized images while simultaneously improving the computational efficiency of the binarization algorithms. To achieve these objectives, the program has been developed in C++, a language chosen for its powerful capabilities in handling complex computational tasks and its versatility in optimizing performance. Furthermore, the integration of OpenMP is a strategic choice, enabling the parallelization of processes to leverage multi-core processing architectures, thus significantly reducing execution times and increasing throughput.

KEYWORDS

image binarization, adaptive thresholding, algorithm engineering, binarization, document image

1 INTRODUCTION

The quintessence of image binarization lies in converting a color or grayscale image, loaded with textual content, into a binary image delineated solely by black and white pixels. This dichotomy engenders a pronounced contrast, pivotal for the facile classification of pixels in machine learning algorithms, thereby streamlining the processing of textual data, whether printed or handwritten. Moreover, binarization plays a crucial role in the preservation of historical documents, despite the challenges posed by variable lighting conditions, disparate camera qualities, and the deterioration of paper.[4]

Complicating factors extend beyond the variability of source images to encompass the exigencies of algorithmic implementation and computational efficiency. A paramount consideration is the optimization of the binarization process to manage the extensive pixel analysis required during preprocessing. This is critical not only for the expeditious handling of individual images but also for scaling to the preprocessing needs of machine learning applications, which may involve tens of thousands of images.

2 THEORY OF IMAGE BINARIZATION

Image binarization is the process of converting a grayscale or color image into a binary image, where each pixel is classified as either foreground (object) or background. This process plays a crucial role in various image processing tasks, such as document analysis, object recognition, and computer vision. In this section, we discuss the theoretical foundations and methods commonly employed in image binarization.

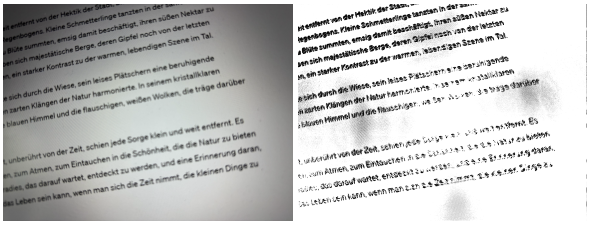

Figure 1: On the left, an example image is presented. On the right, the image after binarization is shown.

2.1 Thresholding

Thresholding is the most fundamental technique in image binarization, where a threshold value is chosen to classify pixels as foreground or background based on their intensity. Mathematically, given an input grayscale image $I(x, y)$ and a threshold value T, the binary image $B(x, y)$ is obtained as:

$$B(x, y) = \begin{cases} 1 & \text{if } I(x, y) \geq T \\ 0 & \text{otherwise} \end{cases}$$

where $B(x, y)$ represents the binary output, and $I(x, y)$ denotes the intensity of the pixel at coordinates (x, y).

2.2 Global Thresholding

Global thresholding methods utilize a single threshold value across the entire image. Common techniques include Otsu's method, which maximizes the between-class variance, and the minimum error thresholding method, which minimizes classification errors.[5]

2.3 Local Thresholding

Local thresholding methods adaptively select threshold values for different regions of the image, considering local variations in intensity. Techniques such as Niblack's method and Sauvola's method are widely used for local thresholding.[1, 5]

2.4 Adaptive Thresholding

Adaptive thresholding techniques dynamically adjust threshold values based on local image characteristics, such as mean or median

intensity. Adaptive methods are effective in handling illumination variations and uneven backgrounds.[1, 2]

2.5 Other Methods

In addition to the thresholding techniques explored in our study, there exist other sophisticated methods, such as gradient-based thresholding and entropy-based thresholding. These approaches offer unique advantages in processing and analyzing images by leveraging the gradients of image intensity and the entropy within image regions, respectively. However, due to the specific assignment guidelines provided by our professor, our focus in this program will be primarily on the previously mentioned methods.[3, 5, 6]

3 ALGORITHM IMPLEMENTATION

In this section, we will delve into the intricacies of the code developed by the authors, beginning with a detailed examination of the main function and the critical steps executed within it. Following this, our attention will shift to the thresholding methods, where we will elucidate our approach to implementing these techniques.

3.1 main

The program is structured as a main function that accepts command-line arguments, processes an input image, applies several thresholding methods, and measures the performance of these operations.

```
if (argc < 2) {
    cerr << "Usage: " << argv[0] << " <
        image_path>" << endl;
    return -1;
}
```

The program checks if it has been provided with at least one command-line argument besides the executable name. If not, it prints a usage message and exits. This is essential for ensuring the program operates with the necessary input and provides basic user guidance.

```
const char* imagePath = argv[1];
unsigned char* img = loadImage(imagePath,
    width, height, channels);
```

The program then uses the provided image path to load the image into memory. The `loadImage` function, presumably a custom or library function not shown in the snippet, initializes the `img` pointer to the loaded image's data and sets the image's dimensions and number of channels (color depth).

```
gray_channels = channels == 4 ? 2 : 1;
gray_img_size = width * height *
    gray_channels;
auto* gray_img = new uint8_t[gray_img_size];
```

Before converting the image to grayscale, the program calculates the required size for the grayscale image buffer. If the original image includes an alpha channel (`channels == 4`), the grayscale image will have two channels (presumably grayscale and alpha); otherwise, it will be a single-channel image. Memory is then dynamically allocated for the grayscale image.

```
for (unsigned char *p = img, *pg = gray_img;
     p != img + img_size; p += channels, pg
    += gray_channels) {
    *pg = (uint8_t)((*p + *(p + 1) + *(p +
        2)) / 3.0);
    if (channels == 4) {
        *(pg + 1) = *(p + 3);
    }
}
```

This loop iterates over each pixel in the original image, computes the average of the red, green, and blue components to obtain the grayscale value, and assigns this value to the corresponding pixel in the grayscale image buffer. If the original image includes an alpha channel, that value is copied directly.

```
start_time = omp_get_wtime();
// A thresholding function is called
end_time = omp_get_wtime();
cout << "Time for ...: " << end_time -
    start_time << " seconds." << endl;
```

The program uses OpenMP's `omp_get_wtime` function to measure the execution time of various thresholding operations, demonstrating a focus on performance optimization. These operations include basic grayscale conversion, primitive thresholding, adaptive thresholding, a proposed thresholding method utilizing global mean and standard deviation calculations, and local min-max thresholding.

```
stbi_image_free(img);
delete[] gray_img;
delete[] prim_MM_img;
delete[] adaptiv_MM_img;
delete[] prop_MM_img;
```

The program concludes with releasing dynamically allocated memory for the image buffers, ensuring there are no memory leaks. The original image data loaded into `img` is freed using a specific function, provided by the same library used for image loading.

3.2 convertToGrayscale

The function `convertToGrayscale` takes as input an image stored in a linear array `img`, its width `width`, height `height`, and the number of channels `channels`. It outputs a grayscale image to the array `gray_img`. The algorithm works as follows:

The function begins by announcing the start of the conversion process. It then calculates the total size of the image array (`img_size`) and determines the number of channels in the grayscale image (`gray_channels`), which is 2 if the original image has an alpha channel and 1 otherwise.

The core of the function is a loop that iterates over each pixel of the input image, averaging the red, green, and blue components to compute the grayscale value. This operation is parallelized using OpenMP, as indicated by the `#pragma omp parallel for` directive. This allows the loop to be executed simultaneously by multiple threads, each processing a portion of the image, thereby accelerating the conversion process.

If the original image includes an alpha channel (i.e., `channels` equals 4), the function preserves this channel by copying it directly to the grayscale image array. Finally, the function announces the completion of the conversion process.

3.3 primitiveThreshold

The `primitiveThreshold` function is designed to apply a simple thresholding operation on a grayscale image.

The function takes five parameters: pointers to the input grayscale image (‘gray_img‘) and the output binary image (‘MM_img‘), the dimensions of the image (‘width‘ and ‘height‘), and the number of channels in the grayscale image (‘gray_channels‘).

The function begins by announcing the start of the thresholding process. It calculates the total size of the image array (‘img_size‘) based on the provided dimensions and number of channels. A parallel for loop, facilitated by OpenMP through the `#pragma omp parallel for‘` directive, iterates over each pixel (or pixel group if ‘gray_channels‘ is 2). For each pixel, if its value exceeds 128, it is set to 255 in the output image, representing white. Otherwise, it is set to 0, representing black. If the input image includes an alpha channel (‘gray_channels‘ equals 2), this alpha value is copied unchanged to the output image. Finally, the function signals the completion of the thresholding process.

3.4 adaptiveThreshold

The ‘adaptiveThreshold‘ function applies an adaptive thresholding technique to a grayscale image. The algorithm, provided below, demonstrates a sophisticated approach that leverages local statistics to determine the optimal threshold for each pixel.

The function’s core logic involves iterating over each pixel in the input grayscale image and determining whether it should be classified as foreground (white) or background (black) in the output binary image. It employs a sliding window approach to analyze local areas around each pixel, calculating local statistics such as the mean and standard deviation within the window. The decision to classify a pixel is then based on these statistics, allowing for a dynamic adjustment of the threshold based on local image characteristics.

A notable aspect of this algorithm is its use of local maximum and minimum values, the average intensity within the window, and a standard deviation-based method for cases with significant contrast variation within the window. This approach ensures that the threshold adapts to both global and local contrast variations, significantly improving the binarization quality in diverse lighting conditions.

3.5 proposedThreshold

The proposed method calculates a unique threshold for each pixel by considering its local mean, local standard deviation, and the relationship between these local statistics and global image statistics. The algorithm is described as follows:

The algorithm begins by calculating the range of global standard deviation within the local windows across the image, establishing a baseline for local variations. It employs a nested loop structure, parallelized using OpenMP’s ‘#pragma omp parallel for collapse(2)‘, to iterate over each pixel in the image, ensuring efficient processing by utilizing multiple threads. For each pixel, the method calculates the local mean and standard deviation within a defined window size around the pixel. These statistics are then used to compute a local adaptive threshold. The thresholding decision for each pixel is based on a formula that dynamically adjusts according to the pixel’s local mean, local standard deviation, and their relationship with the global statistics. This adaptive approach allows the algorithm to effectively handle images with varying lighting conditions, enhancing the binarization quality.

3.6 localMinMax

The function first calculates the contrast for each pixel in the grayscale image, storing the results in a floating-point array. This step involves assessing the intensity variation between a pixel and its surrounding pixels, highlighting areas of the image with significant changes in brightness.

Using a predefined threshold (in this case, 0.5), the algorithm identifies pixels with contrast values above this threshold, marking them in a binary mask. These high-contrast pixels typically correspond to edges or features within the image, which are critical for determining appropriate binarization thresholds.

With the high-contrast mask and the original grayscale image, the function then estimates local thresholds for each pixel or region. It uses these thresholds to binarize the image, setting pixels to either black or white based on their intensity relative to the local threshold. This step is crucial for adapting to varying lighting conditions and preserving important features in the binarized image.

Finally, dynamically allocated memory for the contrast image and high-contrast mask is released, concluding the function.

3.7 Performance Evaluation and Optimization

In the development of each algorithm, our primary focus was on maximizing the utilization of OpenMP to enhance parallel processing capabilities, while simultaneously ensuring that the integrity of the results remained uncompromised. To prevent any potential conflicts that might affect the outcomes, we implemented rigorous testing protocols to guarantee the reliability of the results. Additionally, we aimed to optimize the computational efficiency of our algorithms, striving to minimize the number of operations required. This approach not only improved the execution speed but also contributed to the overall robustness and effectiveness of our image processing solutions.

4 RESULTS

The primitive thresholding technique proves effective for images with limited variation and inherently strong contrast; otherwise, the outcomes are typically unsatisfactory.

Conversely, the local thresholding technique yielded a significantly more balanced image, thanks to its analysis of the contextual information around each pixel. However, it struggled to produce distinct contrasts.

The adaptive threshold method was distinguished as better than the previous ones, thanks to its integration of a global factor. This key addition empowers the algorithm to adjust the thresholding criteria flexibly, considering both local and overarching brightness variations, and thus achieves a superior balance in image detail and clarity.

The proposed algorithm, while exceeding the adaptive method in performance, still managed to significantly outperform the adaptive threshold. By employing more sophisticated techniques, it adeptly adapts to the specific attributes of the image, securing a strong second place in our assessment.

However, a critical examination of the processing times reveals a significant disparity. The adaptive techniques, despite their superior image quality outcomes, necessitate considerably longer computation times due to their more complex threshold calculations.

- Grayscale conversion: **0.00202378 seconds**
- Primitive thresholding: **0.0135227 seconds**
- Local Min-Max thresholding: **0.100658 seconds**
- Proposed method: **1.56501 seconds**
- Adaptive thresholding: **17.5788 seconds**

This data underscores a trade-off between computational efficiency and the quality of the thresholding result, highlighting the complexities involved in choosing the optimal approach for different applications.

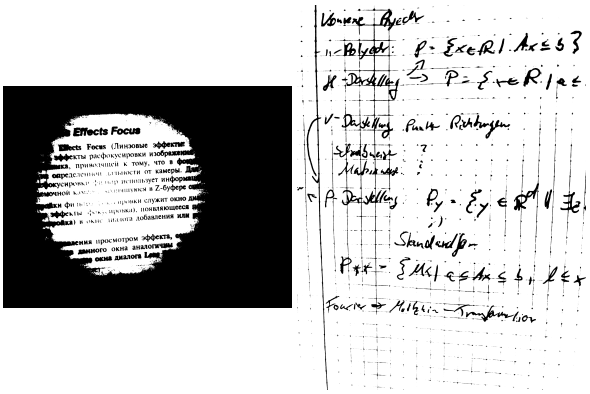

Figure 2: Result of primitive threshold method.

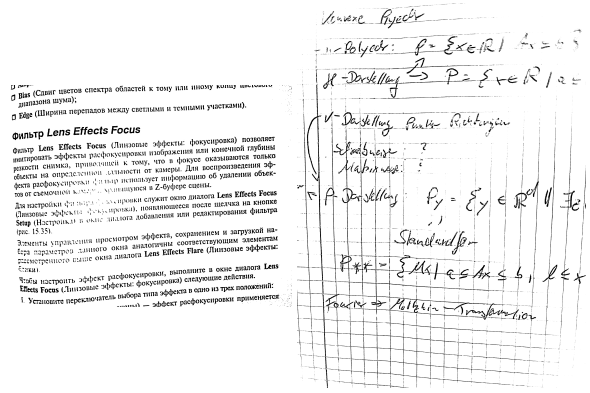

Figure 3: Result of proposed threshold method.

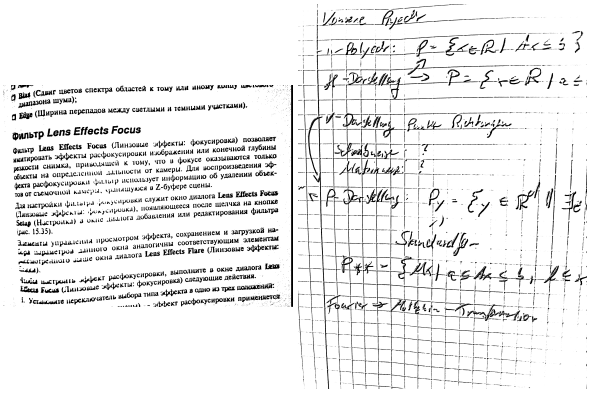

Figure 4: Result of adaptive threshold method.

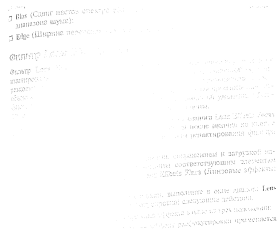

Figure 5: Result of LMM threshold method.

5 CRITICAL EVALUATION

It's important to mention that the evaluation of the analyzed methods was conducted using a small collection of images. As a result, the findings may be overly specific to this particular set of examples. This detail might also help explain why the method suggested in the literature showed a slight improvement; it's designed specifically for handling documents with historical complexities.

Additionally, there's room for improvement in both the algorithm and the code structure, possibly by employing techniques that streamline processing and make better use of computer memory. The initial investigation primarily focused on utilizing OpenMP, indicating that the exploration of other strategies could potentially improve the efficiency of each method.

6 CONCLUSION

The exploration and development of thresholding algorithms presented us with an engaging and enriching experience, forming a highlight of our coursework. This exercise, embarked upon within the framework of our lecture series, offered us a hands-on opportunity to delve into the intricacies of image processing and the optimization of computational algorithms.

Through this exercise, we were introduced to the complexities involved in discerning and enhancing the details within images, a task that is both challenging and fascinating. We navigated through various methods of thresholding, learning how each approach affects the outcome differently and discovering the importance of selecting the appropriate technique based on the image's characteristics and the desired result.

One of the most valuable lessons derived from this experience was the art of code optimization. We experimented with different strategies to improve the efficiency of our algorithms, from refining logic and reducing computational overhead to leveraging advanced programming constructs. These endeavors not only improved the performance of our image processing tasks but also deepened our understanding of how software efficiency impacts real-world applications.

REFERENCES

[1] Bilal Bataineh, Siti N. H. S. Abdullah, K. Omar, and M. Faidzul. 2011. *Adaptive Thresholding Methods for Documents Image Binarization.* Springer Berlin Heidelberg, 230–239. https://doi.org/10.1007/978-3-642-21587-2_25

[2] Derek Bradley and Gerhard Roth. 2007. Adaptive Thresholding using the Integral Image. *Journal of Graphics Tools* 12, 2 (Jan. 2007), 13–21. https://doi.org/10.1080/2151237x.2007.10129236

[3] Chein-I Chang, Kebo Chen, Jianwei Wang, and Mark L.G. Althouse. 1994. A relative entropy-based approach to image thresholding. *Pattern Recognition* 27, 9 (Sept. 1994), 1275–1289. https://doi.org/10.1016/0031-3203(94)90011-6

[4] Bolan Su, Shijian Lu, and Chew Lim Tan. 2010. Binarization of historical document images using the local maximum and minimum. In *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems (DAS '10)*. ACM. https://doi.org/10.1145/1815330.1815351

[5] Wei Xiong, Lei Zhou, Ling Yue, Lirong Li, and Song Wang. 2021. An enhanced binarization framework for degraded historical document images. *EURASIP Journal on Image and Video Processing* 2021, 1 (May 2021). https://doi.org/10.1186/s13640-021-00556-4

[6] Haniza Yazid and Hamzah Arof. 2013. Gradient based adaptive thresholding. *Journal of Visual Communication and Image Representation* 24, 7 (Oct. 2013), 926–936. https://doi.org/10.1016/j.jvcir.2013.06.001