GPU-based and Streaming-enabled Implementation of Pre-processing Flow towards Enhancing Optical Character Recognition Accuracy and Efficiency

Gener Serhan, Dattilo Parker, Gajaria Dhruv, Fusco Alexander and Akoglu Ali

Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, 85721, AZ, USA.

*Corresponding author(s). E-mail(s): gener@arizona.edu; parkerdattilo@arizona.edu; dhruvgajaria@arizona.edu; afusco1@arizona.edu; akoglu@arizona.edu;

Abstract

Research has demonstrated that digital images can be pre-processed through operations such as scaling, rotation, and blurring to enhance the accuracy of optical character recognition (OCR) by emphasizing important features within the image. Our study employed the open-source Tesseract OCR and found that accuracy can be improved through pre-processing techniques including thresholding, rotation, rescaling, erosion, dilation, and noise removal, based on a dataset of 560 phone screen images. However, our CPU-based implementation of this process resulted in an average latency of 48.32 ms per image, which can hinder the processing of millions of images using OCR. To address this challenge, we parallelized the pre-processing flow on the Nvidia P100 GPU and executed it through a streaming approach, which reduced the latency to 0.825 ms and achieved a speedup factor of 58.6x compared to the serial execution. This implementation enables the use of a GPU-based OCR engine to handle multiple sources of data streams with large-scale workloads.

Keywords: Optical Character Recognition (OCR), Tesseract, Leptonica, Image Processing, CUDA, GPU

1 Introduction

The conversion of images containing text into text-only outputs is a common use of Optical Character Recognition (OCR). OCR systems rely heavily on feature extraction and classification based on patterns. OCR has numerous applications, including the digitization of handwritten and typewritten documents, the recognition of license plates, and the imaging of invoices [1]. Certain use cases of OCR prioritize real-time processing and high-speed execution for streams of images. For instance, postal services rely on OCR to swiftly scan over 17,000 packages per hour [2].

Tesseract, an extensively utilized open-source OCR engine with multilingual support [3], employs a neural network-based recognition system [4]. This engine takes images as input and generates corresponding data text lines as output. Although Tesseract's accuracy varies across different datasets, the accuracy of the OCR engine can be significantly improved through image pre-processing techniques [5–7]. For instance, studies have shown that Tesseract OCR achieves an F1 score of 0.163 on the Brno Mobile OCR Dataset [8], but through pre-processing, the F1 score can increase up to 0.729 [9]. To evaluate the impact of pre-processing on Tesseract's accuracy, we conducted a preliminary analysis

using 560 images of phone screen menus captured in an indoor setup with a mounted camera. Without pre-processing, Tesseract's average accuracy was found to be 81.98%, which improved to 84.61% after applying techniques such as binarization, noise prediction, image sharpening, contrast management, and rotation. However, it is worth noting that the introduction of pre-processing incurs execution time overhead. Leveraging Leptonica library implementations of these kernels, we observed a total execution overhead of 48.32 ms per 1920x1080 image on an Intel(R) Xeon(R) processor at 2.40GHz. Considering the throughput of 17,000 packages per hour in postal services, the pre-processing overhead of 48.32 ms per image results in a 22.82% increase in execution time for each package, reducing the total throughput to 13,842 packages per hour. Recognizing the data parallel nature of the pre-processing kernels, which makes them suitable for GPU implementation, we aimed to address the pre-processing overhead in a use case where an OCR engine receives images from multiple cameras for streaming processing. We implemented the entire pre-processing chain on an Nvidia P100 GPU to minimize its overhead and established a streaming-based execution methodology.

We initiate the process by converting the pre-processing algorithms from the Leptonica libraries into simplified C++ code to achieve this. Subsequently, we transform these algorithms into CUDA using a global memory implementation. We then delve into CUDA optimization, exploring techniques such as shared memory and constant memory. We use our knowledge of Tesseract's pre-processing to combine kernels and reuse data in shared memory for more efficient use. Additionally, we investigate optimization with streaming, leveraging the combined kernels to enhance data reuse within the kernels. Furthermore, we examine the interplay between structuring elements, specifically the mask sizes for the convolution kernel, and their impact on these CUDA optimizations [1]. Through experimentation, we determined that the optimal number of supported streams on the P100 GPU is 39, resulting in a reduced pre-processing time of 0.825 ms per image for our dataset. Considering the throughput requirement of postal services at 17,000 packages per hour, the introduction of pre-processing incurs a 0.41% increase in execution time per package, allowing for a throughput of 16,932 packages per hour while improving accuracy from 81.98% to 84.61%.

A preliminary version of this work appeared in the *"International Conference on Computer Systems and Applications (AICCSA 2022)"* [10], where we presented a baseline GPU-based pre-processing implementation. In this paper, we expand the preliminary work with the following contributions:

- We designed and implemented a streaming-enabled execution for the pre-processing flow that improves the GPU resource utilization.
- We expose the performance constraints imposed on streaming-based execution, and address them by coupling batching technique with streaming where multiple streams of input image batches are processed concurrently and each batch is composed of multiple images.
- We perform design space exploration to identify the optimal batch size and stream size, and improve the throughput from 512 images/sec to 1212 images/sec with the streaming enabled execution.
- We expand the literature review on character recognition with a focus on the need for pre-processing in the trade space of execution time performance and recognition accuracy.

The structure of this paper is organized as follows: In Section 2 we discuss character recognition techniques. In Section 3, we provide an algorithmic perspective of the pre-processing flow, explain the rationale behind the chosen pre-processing methods, and discuss our parallelization strategies. Next, we present our experimental setup and analyze the obtained results in Section 4 and Section 5, respectively. Finally, we conclude and outline potential future work in Section 6.

2 Related Work

The process of a fully functional OCR engine is thoroughly explained in both [11] and [12]. They divide this process into six main steps: image acquisition, pre-processing, segmentation, feature extraction, classification, and post-processing. Focusing the second step, which is pre-processing [11], mentions five crucial operations,

[1] https://github.com/YSerhanGener/GPGPU-OCR-Pre-processing

binarization, spatial image filtering operations, thresholding, noise removal, and skew correction detailing how pre-processing improves the pictures' quality and, in return, the accuracy of the OCR engines.

Numerous studies have focused on enhancing the accuracy of various character recognition engines. Some works concentrate on image processing operations [13–17], while others employ machine learning (ML)-based approaches [9, 16–18] to improve accuracy. In image processing-based approaches, normalization, binarization, and equalization techniques are utilized in [14], while methods such as greyscale conversion, contrast alteration, unsharpening, and binarization are employed in [13]. Additionally, contrast-altering techniques for background removal are utilized in [15]. ML-based methods include the adaptive convolution-based pre-processing approach in [9] and the usage of exact string matching and approximate string matching techniques in [18]. Furthermore, approaches such as [16] combine ML and image pre-processing techniques, employing operations like resizing, sharpening, and blurring, coupled with k-means clustering, while [17] determines the most suitable pre-processing operation through CNNs and utilizes algorithms like binarization, noise reduction, and sharpening. All of these approaches have demonstrated improvements in the accuracy of different character recognition engines, including TesseractOCR [3], NNOCR [19], and Google Cloud Vision [20], showing the importance of pre-processing in every OCR engine.

3 Overview of Pre-processing Implementation

Tesseract currently incorporates limited support for utilizing OpenCL in its character recognition neural network [21]. However, it lacks GPU support for the pre-processing algorithms that enhance recognition quality. For the recommended pre-processing algorithms, Tesseract suggests the C-based Leptonica library.

Our research focuses on optimizing the pre-processing algorithms commonly employed by Tesseract using CUDA. Figure 1 illustrates the stages involved in the pre-processing flow examined in this study, while Figure 2 showcases the outputs from each stage. In the subsequent paragraphs, we provide a concise description and the GPU implementation approach adopted for each stage of the pre-processing flow.

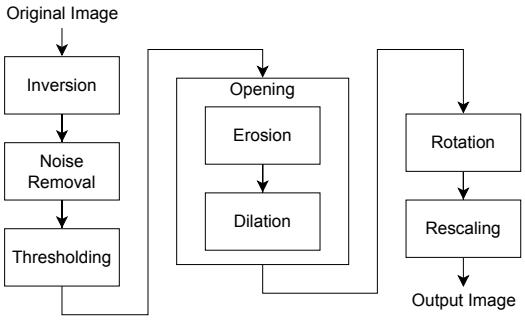


Fig. 1: The pre-processing flow used for Tesseract OCR.

3.1 Pre-processing Kernels

3.1.1 Grayscale and Inversion

The Grayscale operation transforms an image into black and white by converting the RGB values of each pixel into a single value ranging from 0 to 255. Following that, an inversion process is executed, flipping the value of each pixel in the image. In a grayscale image where pixel values range from 0 to 255, each pixel's value is substituted with 255 minus its current value after the inversion step is applied [22]. This kernel aligns well with the fine-grained programming model of CUDA, where each thread is assigned a single pixel.

3.1.2 Noise Removal

The majority of digital images typically contain noise that arises during the capture or transmission process. To eliminate undesired details that might interfere with character recognition, a linear algorithm is applied for noise removal [23]. This algorithm employs convolution to blur the image. The GPU utilizes constant memory to store the mask array to facilitate this process. Additionally, a tiling method enhances data access speed and workload distribution among threads within a block, improving memory bandwidth utilization.

3.1.3 Thresholding

Thresholding is a technique used for binarization, where the pixel colors are assigned either black or

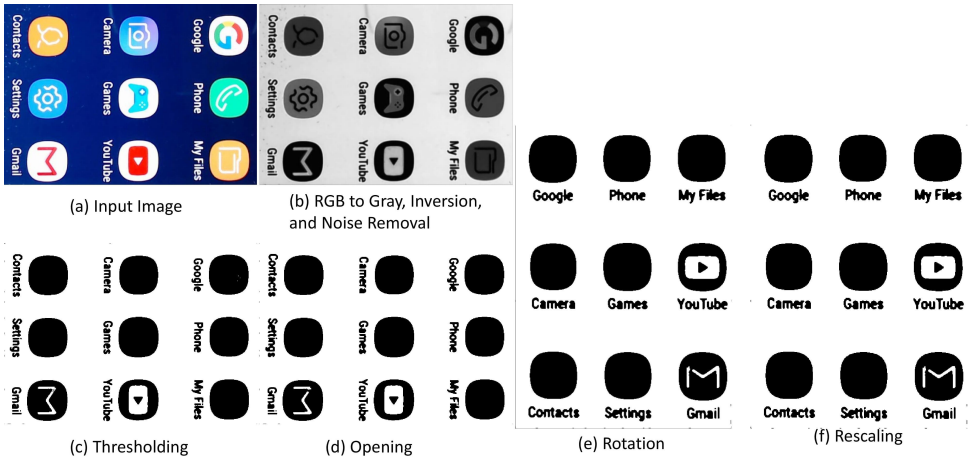


Fig. 2: Sample image captured by the OCR testbed and corresponding outputs of the individual stages of the pre-processing flow shows masking of non-text portions of the image for improved accuracy.

white values. This process is commonly employed to distinguish foreground from background in an image. In the context of OCR, it specifically aids in separating text from non-text elements, thereby enhancing recognition accuracy [21]. Leptonica supports both Otsu's method [24] and Sauvola's method [25] of thresholding. However, for our implementation, we opted to utilize Otsu's method [24] due to its successful parallelization on GPUs in a previous study [26]. Moreover, Otsu's method exhibits a lower algorithmic complexity as a global thresholding technique compared to Sauvola's local thresholding approach [27]. The pixel-level operations based on histogram analysis align well with the data parallel execution model of CUDA, utilizing shared memory.

3.1.4 Erosion and Dilation

Erosion eliminates irrelevant details from a binary image [28] by applying convolution with a structuring element as the mask. On the other hand, dilation is a convolution-based kernel that fills in gaps that may arise after the erosion phase, effectively repairing these gaps within the image [28]. In Figure 1, the combined operation of erosion followed by dilation is referred to as Opening. Both of these kernels contribute to sharpening the image, resulting in clearer characters. In our implementation of the erosion and dilation kernels, we leverage constant memory and employ a tiling approach similar to the one used for the Noise Removal kernel, which allows for efficient memory utilization and facilitates faster data access and workload sharing among threads within a block.

3.1.5 Rotation

To properly process text, Tesseract assumes it is aligned in the standard left-to-right reading orientation. To ensure correct alignment, a method called *gather* [29] is used, which involves rotating the images by 90 degrees. This rotation is achieved through a transformation matrix [30] in the CUDA implementation, with each thread assigned to a specific destination pixel. The matrix maps this destination pixel to its corresponding source pixel.

3.1.6 Rescaling

For optimal character recognition performance, Tesseract's neural network operates best when the image is resized to a resolution of at least 300 pixels per inch [21]. When adding additional pixels to the image to achieve this scale, interpolation techniques are employed to determine the values of these extra pixels based on neighboring pixels. In a data parallel fashion on the GPU, each thread is assigned an output pixel and utilizes bilinear interpolation [31–33] to upsample the image.

3.2 Kernel merging for improving data reuse

The pre-processing stages exhibit a tight coupling of data dependencies, resulting in a serial execution flow. In cases where a kernel utilizes shared memory if its preceding or succeeding kernel also utilizes shared memory, the shared memory can serve as a communication medium instead of relying on global memory. Moreover, merging kernels helps reduce the overhead associated with launching multiple kernels.

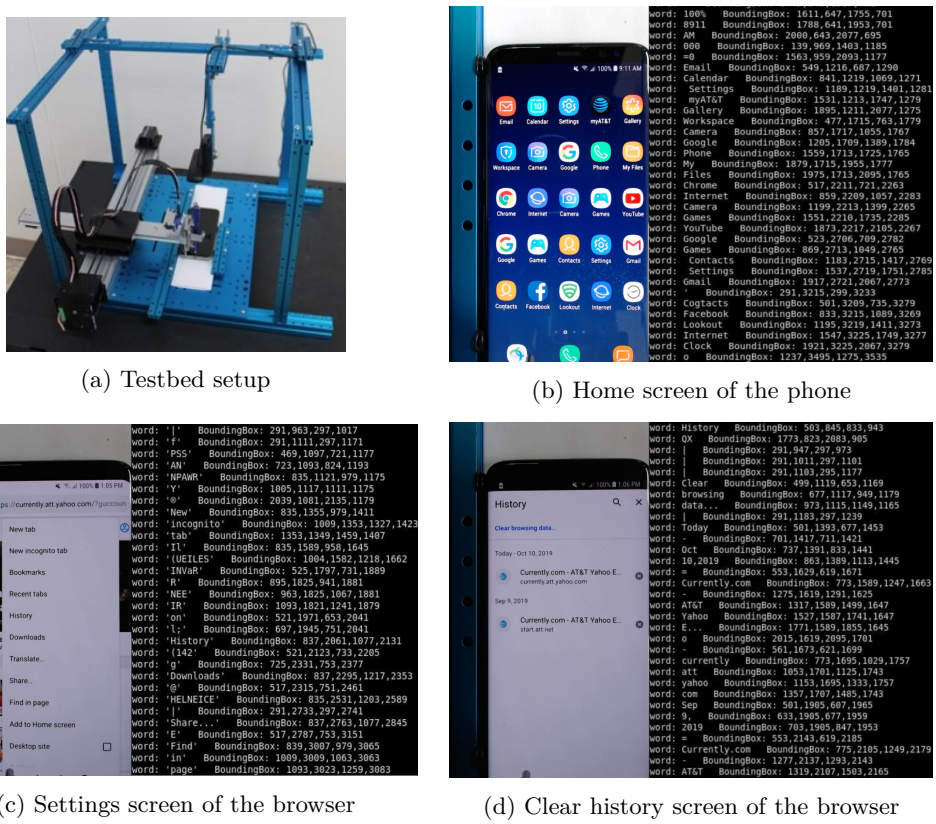
To optimize data transfers between global and shared memory, we combine the inversion, noise removal, and histogram collection steps of Otsu's method. Keeping the data in shared memory and reusing it prevents frequent transfers between global and shared memory. Since the computation for the global threshold primarily operates on the 256-element histogram rather than the full-sized 1920x1080 image, separating these steps into different thread block sizes is more efficient. Multiple kernel calls are made for each step of the threshold calculation process within Otsu's thresholding kernel. We modify the approach presented in [26] by combining some of the kernel calls. After collecting the initial histogram, we generate zero-order and first-order cumulative histograms to compute the inter-class variance for each threshold value. Subsequently, we select the threshold value that maximizes the variance, which is then applied to the input image in a final kernel call. All computations rely on the same histogram data throughout these threshold computation steps. Furthermore, we merge the erosion and dilation kernels since both involve convolution and utilize shared memory. Combining these two kernels allows for the reuse of shared memory, similar to merging the inversion and noise removal kernels.

It is essential to consider that merging kernels can result in an increase in the register footprint, potentially leading to a decrease in the number of threads per block. This reduction in thread count would significantly impact the total number of threads launched at the grid level. To mitigate this issue, we adopt an approach in our work where we use 8-bit data for our local variables, preventing register spilling and helping maintain efficient register usage. Our analysis demonstrates that the merging approach employed in this study is, at most, a register usage of 18 registers. Thus, it ensures that the register constraint does not hinder the launch of thread blocks on the NVIDIA P100 GPU.

4 Experimental Setup

Our experimental testbed, depicted in Figure 3a, consists of a robotic arm equipped with a stylus, a top-mounted camera, and the phone being tested. The camera captures the phone's screen image, which is then sent to an OCR engine to identify the target position. The resulting coordinates are transmitted to the robotic arm, which moves the stylus to the desired location and presses it accordingly. To illustrate the process of clearing the browsing history on the phone, we present Figures 3b- 3d. This test sequence begins at the phone's home screen and involves three OCR operations. Starting from the home screen, we capture the image, detect the "Chrome" keyword, determine the position of the "Chrome" icon in the 2D grid, maneuver the robotic arm to the target location, and finally, the stylus presses the "Chrome" icon. This process is repeated to identify the "History" keyword in the second step and the "Clear Browsing Data" keyword in the third step, completing the testing procedure. Figures 3b- 3d display the phone screens and the corresponding OCR text output at each stage of the test sequence. These testing sequences are crucial for certifying information technology products. It is crucial to incorporate automation and achieve high throughput execution to conduct large-scale verifications promptly.

To handle the arrival and processing of images from multiple testbeds running various testing sequences on multiple devices concurrently, we employ an Nvidia P100 GPU server hosted on an Intel(R) Xeon(R) processor operating at 2.40GHz, coupled with 192GB of RAM. This server is the central unit for receiving and processing the captured images from the testbeds. It then communicates the necessary coordinates to the respective testbeds for stylus movement. We have implemented a streaming approach for the image preprocessing kernels to cater to the requirement of simultaneous image processing. This implementation significantly enhances throughput, enabling efficient processing of input images in parallel.



(a) Testbed setup

(b) Home screen of the phone

(c) Settings screen of the browser

(d) Clear history screen of the browser

Fig. 3: Camera mounted over the phone under test captures the phone screen image, sends image to OCR engine to locate Internet app and launch it by positioning stylus attached to the robotic arm over the Internet icon. A sequence of images are shown for realizing a test sequence for clearing browsing history with phone screen image coupled with the extracted words through OCR.

We conduct pixel-wise comparisons between the outputs and their corresponding CPU implementations to ensure the functional verification of each step in the pre-processing flow illustrated in Figure 1. To verify our work, we utilize the CUDA-optimized pre-processing output as the input for Tesseract 5 [21]. Tesseract, which offers a C/C++-based API, is integrated with our proposed work. This API call expects an image as input and provides the resulting text as a string output. In order to utilize Tesseract in our work, we first initialize it using the English dataset. To assess the accuracy of the optical character recognition (OCR), we compare the percentage of correctly recognized characters between the input image and the output text string. When the Tesseract API call is employed without pre-processing, it produces random characters for the original input image. To ensure a fair comparison, we rotate the original image by 90 degrees, if necessary, before making the Tesseract API call.

The dataset for cell phone input comprises 560 phone screen images taken indoors using multiple testbeds with a similar layout featuring a top-mounted camera, as depicted in Figure 3a. These images were captured under diverse lighting and shading conditions. Figure 4 presents a subset of this dataset, showcasing variations in contrast, noise, and glare levels that challenge the accuracy of optical character recognition (OCR). It is worth noting that the Tesseract OCR used in this study is not specifically trained with this particular cell phone image dataset. Instead, we utilized the generic trained model of Tesseract designed for the English dictionary.

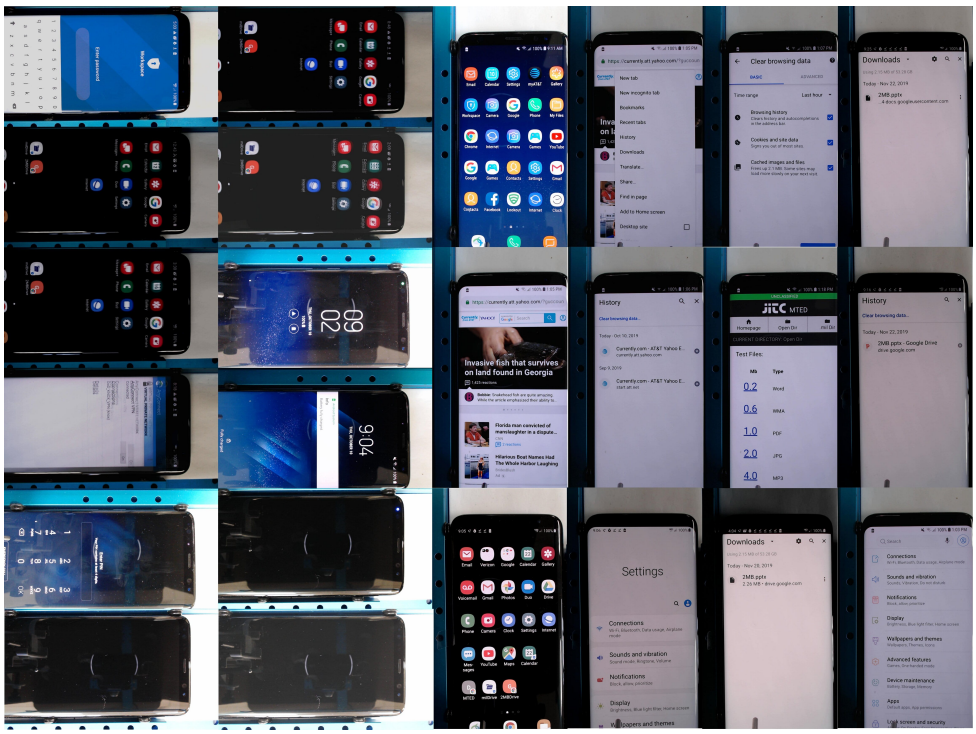


Fig. 4: 24 sample phone screen images from the dataset obtained from the multiple Testbed setups showing various artifacts such as glare, noise, and contrast stressing the OCR accuracy.

5 Results

Within this section, we introduce our step-by-step enhancements and assess their effect on the duration of execution, alongside the advancements in accuracy resulting from the pre-processing workflow compared to Tesseract's OCR. Results are collected based on the total execution times including the data transfer overhead using the Nvidia P100 GPU.

5.1 Impact of thread block configuration and mask size

We examine the influence of thread block organization on the execution time, using configurations of 8x8, 16x16, and 32x32 threads per block, as depicted in Figure 5. Our findings indicate that, except for the thresholding kernel, a block size of 16x16 yields the most favorable execution time. For the thresholding kernel, the 32x32 configuration exhibits the fastest execution due to smaller block sizes necessitating a higher number of thread blocks to cover the image dimensions, resulting in an increased number of

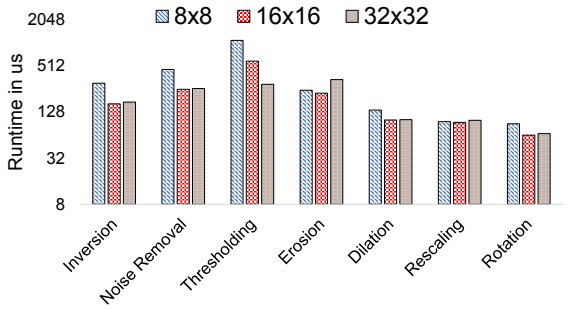


Fig. 5: Block size sweeping experiments of kernels in the pre-processing flow shown with a log scale.

sequential writes to global memory. When multiple threads write to the same histogram bin in shared memory, the penalty for atomic operations is comparatively less costly and less frequent than sequentially accumulating each shared memory histogram bin from each thread block into global memory. We also observe that the thresholding kernel exhibits the longest runtime among all kernels when using both 8x8 and 16x16 block

Table 1: Execution time analysis with respect to various mask sizes on Xeon 2.4GHz CPU and Nvidia P100 GPU of relevant kernels.

		Mask Size		
		3x3	5x5	7x7
CPU (ms)	Noise Removal	121.536	328.828	635.371
	Erosion	158.414	466.971	935.686
	Dilation	184.096	417.135	862.09
GPU (ms)	Noise Removal	0.26	0.474	0.795
	Erosion	0.22	0.34	0.64
	Dilation	0.22	0.33	0.64

sizes. This consideration is considered during the kernel merging process described in Section 5.2.

The noise removal, erosion, and dilation kernels can utilize constant memory through 2D-convolution based execution. Consequently, we investigate the execution time using mask sizes 3x3, 5x5, and 7x7, as outlined in Table 1. In the case of the noise removal kernel, smaller mask sizes are particularly effective for OCR applications as they effectively eliminate noise at the edges of characters. Conversely, when applied to larger images, the effects of erosion and dilation kernels are less noticeable with smaller mask sizes. Given that our main focus in this work is execution time and considering that we are working with 1920x1080 pixel-sized images, we select the 3x3 mask size for all relevant kernels during kernel merging. However, it is worth noting that larger sizes for erosion and dilation may yield improved accuracy at the cost of speed.

5.2 Kernel Merging

In this experiment, we combine two sets of kernels: the first set includes inversion, noise removal, and a portion of thresholding, while the second set comprises erosion and dilation (opening). We compare the runtimes of the merged kernels with the sum of the individual kernel runtimes for these two sets, as illustrated in Figure 6, using the mask size chosen in Section 5.1 for the relevant kernels. These results align with the trends observed in Figure 5 for the individual kernels. However, it should be noted that only the creation of the histogram time is considered for the thresholding kernel in Figure 6. The merged kernels all benefit from the larger block size, as the increased number of instructions within a single kernel call enables

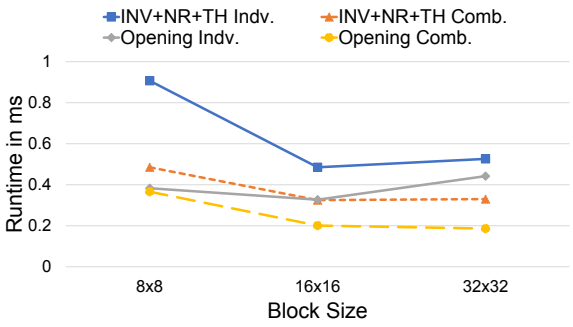


Fig. 6: Merged kernel runtime compared to the sum of individual kernels using mask size of 3 where Comb label indicates merged kernels, and Indv indicates the individual based execution for Inversion (INV), Noise Removal (NR), and part of thresholding (TH) kernels.

the GPU scheduler to make more efficient scheduling decisions. Consequently, we utilize the 32x32 block size for the merged kernel implementations to implement streaming.

Furthermore, we merged the remaining steps of the thresholding kernel that were unrelated to histogram creation. We observed that the individual thresholding kernel steps took 0.44 ms to execute, while the merged thresholding kernel steps ran in 0.14 ms, resulting in a speedup of 3.14x. Combining the kernels, we eliminated the need to store the zero-order and first-order histograms required for Otsu thresholding [24] in global memory between their creation and the subsequent scan and interclass variance calculations. This enhancement improved memory efficiency and reduced kernel initialization overhead, as we reduced the number of kernel calls by a factor of 4.

5.3 Streaming

The sweeping experiments conducted for streaming were designed such that each input image has a dedicated stream for its operations on the GPU. Figure 7 illustrates the results of these sweeping experiments, indicating that the execution time per image reaches saturation after 20 streams. Taking into account the average run-time of memory copy and kernel compute operations, which includes 0.56 ms for the host-to-device memory copy, 2.8 ms for the completion of the pre-processing flow, and 0.18 ms for the

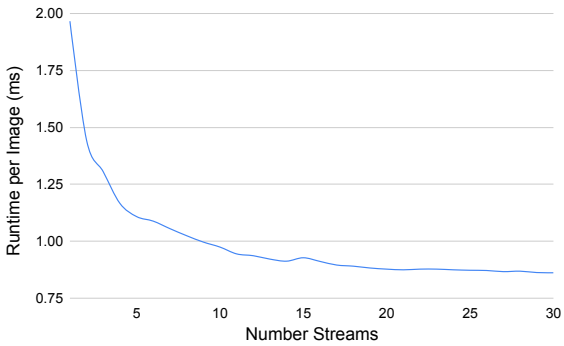


Fig. 7: Average execution time of the pre-processing flow with respect to the number of CUDA streams over the cell-phone image data set.

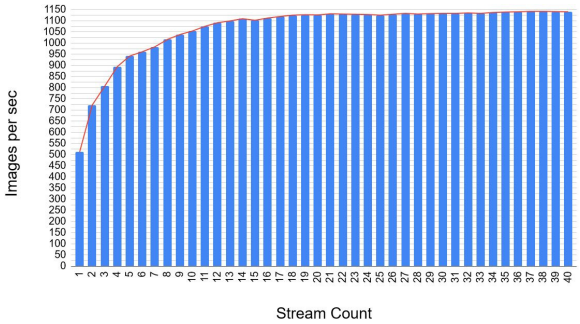


Fig. 8: Image throughput based on the number of streams using one image per stream.

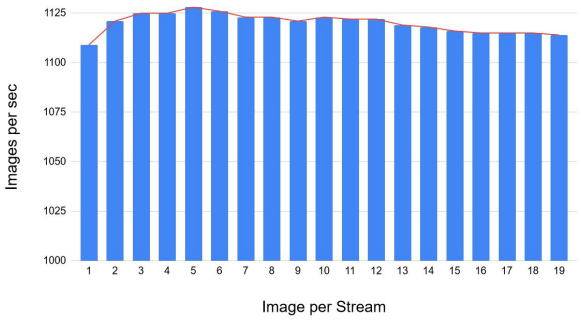


Fig. 9: Image throughput with varying number of images per stream with 18 streams

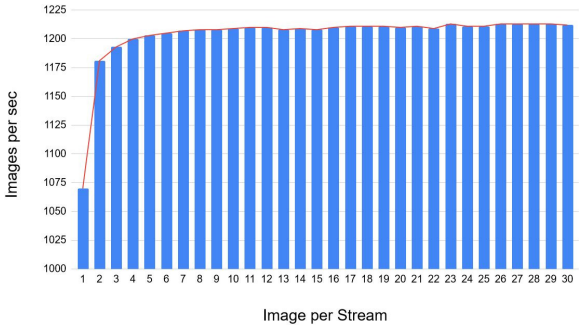


Fig. 10: Image throughput with varying number of images per stream with 18 streams in batch mode

final device-to-host memory copy, we anticipate a 3.7x speedup by utilizing streams and overlapping compute and memory operations. Depending on the device's capabilities, it may be possible to achieve additional speedup by running multiple streams concurrently. In our experiments, we observed a speedup of 4.18x by implementing the combined previous optimizations and using 29 streams. Considering the use case described in Section 4 with 30 testbeds, we can pre-process each image in 0.847 ms when employing the optimal block and mask size configurations, kernel merging, and streaming.

To utilize most of the resources available on the GPU, we conducted experiments with varying numbers of streams and images per stream. Utilizing the streaming feature, we analyzed the image throughput as we increased the number of streams from 1 to 40. Our results, presented in Figure 8, indicate that as we increased the number of streams, the throughput also increased. However, after 18 streams, the throughput saturated at 1125 images per second, with a single stream achieving 512 images per second. Increasing the number of streams to 18 resulted in a 2.236x increase in throughput. However, further increases in the number of streams did not yield significant improvements due to the GPU resources reaching their maximum utilization achievable by increasing the number of streams. Following this observation, we performed an experiment changing the number of images processed by a single stream. Initially, increasing the number of images per stream increased throughput to 1128 images per second at 5 images per stream. However, we observed a gradual decrease in throughput as we increased the number of images per stream since, with the increased number of streams and images per stream, the kernel launch overheads become the bottleneck for the execution time. The results of this experiment are shown in Figure 9. To accommodate this, we modified the image per

Table 2: Execution time analysis for each step of the pre-processing flow including the data transfer overhead to and from the GPU over the baseline serial CPU-based implementation, OpenCV based GPU implementation, initial (baseline) GPU implementation, and optimized GPU implementation.

Kernel	CPU (ms)	OpenCV GPU (ms)	Baseline GPU (ms)	Optimized GPU (ms)	Baseline Speedup over CPU	Optimized Speedup over CPU	Optimized Speedup over OpenCV
Inversion	7.6	0.11	0.17	0.17	44.7x	44.7x	0.65x
Noise Removal	19.61	0.37	0.39	0.26	50.28x	75.42x	1.42x
Thresholding	8.88	1.17	1.3	0.14	6.83x	63.43x	8.35x
Erosion	1.2	0.33	0.24	0.22	5x	5.45x	1.5x
Dilation	1.2	0.11	0.24	0.1	5x	12.00x	1.1x
Rotation	1.66	0.06	0.06	0.06	27.66x	27.66x	1x
Rescaling	8.14	0.35	0.1	0.1	81.4x	81.4x	3.5x
Total (inc. memcpy)	48.32	6.52	4.99	3.54	9.68x	13.65x	1.84x

stream implementation to work with batches of inputs meaning each kernel launch will have multiple images stitched together as an input and process the multiple images in a single kernel call. With this modification, we see a greater increase in the throughput with 1210 images per second and a stable saturation after 11 images per stream, as shown in Figure 10. This resulted in a total of 2.363x increase in the throughput. Finally, based on these results, we conducted another round of number of stream sweeps, using 11 images per stream this time. As shown in Figure 11, from this experiment, we achieved a throughput of 1212 images per second using 11 images per stream at 39 streams. This final experiment resulted in a 2.367x increase in throughput compared to the 1-stream approach.

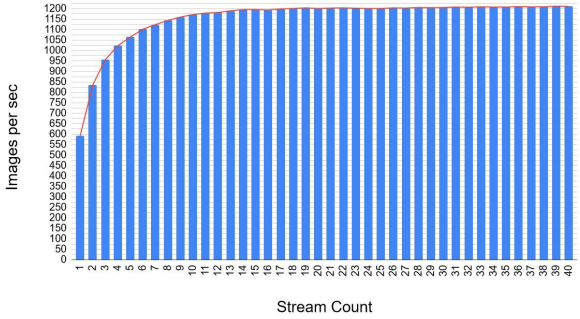


Fig. 11: Image throughput with different stream sizes using 11 images per stream in batch mode

5.4 Summary of Results and Accuracy

Table 2 presents the execution time for each stage of the pre-processing flow based on serial execution on the CPU, OpenCV-implementation executing on the GPU, baseline implementation using global memory-only approach executing on the GPU, and the optimized implementation running on the GPU after applying individual kernel-level optimizations such as tiling, constant memory, and shared memory as discussed in Section 3.1. In terms of overall execution time, the global memory-only approach achieves a speedup of 9.68 compared to the serial execution. After implementing the kernel-level optimizations, we observe an overall speedup of 13.65. Among all the kernels, the thresholding stage benefits the most from the kernel-level optimizations, achieving a speedup factor of 9.26 compared to its global memory-only implementation. Resolving the bottleneck caused by atomic-level operations through shared memory and privatization contributes to this speedup value. Finally, after incorporating kernel merging and implementing streaming, we achieve a total pre-processing flow execution time of 0.825 ms, resulting in a 58.57x speedup compared to the baseline CPU implementation. The reduction in kernel launch overhead and the hiding of data transfer latency through streaming are the critical factors in reducing the execution time from 3.54 ms, as observed with the individual kernel-level optimizations, to 0.825 ms.

We observe 1.30x and 1.84x speedups for the baseline and optimized versions, respectively, compared to the OpenCV implementation. For the OpenCV implementation, the execution time for each kernel except the Otsu's Thresholding algorithm are collected from the GPU. This algorithm for CUDA is not supported in OpenCV. Therefore, we report its runtime based on execution over the CPU including the necessary memory copy operations before and after the thresholding kernel. We observe that in the optimized implementation all kernels except Inversion and Rotation take less time than the OpenCV based implementation further validating the effectiveness of our implementation approach.

According to this study, using CPU and GPU-based pre-processing methods, then applying Tesseract OCR resulted in the same outcomes for the phone image dataset used in the research. The pre-processing flow proposed in this work, implemented using CUDA or the Leptonica library, generates human-readable text output with an accuracy of 84.61%. In comparison, using Tesseract OCR as a standalone tool without pre-processing yields an accuracy of 81.98%.

6 Conclusion

In this paper, we quantify the impact of pre-processing digital images on optical character recognition accuracy and expose the latency associated with it poses as a barrier for achieving high throughput OCR performance. We explore various CUDA optimization strategies through kernel merging, latency hiding with streaming based execution, tiling coupled with shared memory usage. We reduce the latency of the pre-processing from 48.32ms to 0.825ms scale, a factor of 58.57x, with the GPU on average compared to the serial execution. This reduction in latency resolves the barrier of deploying Tessaract OCR in large scale document processing and enables a GPU based OCR engine serve multiple documents captured from up to 39 different testbeds. Future work will involve further exploration of the design of cache-aware warp scheduling architectures to reuse the data available in the cache and further reduce the memory bandwidth requirements. Additionally, moving the implementation of the classification for Tesseract to CUDA instead of using OpenCL will eliminate the data transfer overhead from the start of the pre-processing until the end of the character recognition. The accuracy of OCR is highly dependent on the image characteristics such as the blurriness level and text color. The variation in accuracy observed over the phone image data set indicates that there is also need for exploring alternative pre-processing workflows towards enhancing the Tessarct OCR accuracy.

Acknowledgments. This work is partly supported by National Science Foundation (NSF) research projects NSF CNS-1624668. This material is based upon High Performance Computing (HPC) resources supported by the University of Arizona TRIF, UITS, and Research, Innovation, and Impact (RII) and maintained by the UArizona Research Technologies department.

References

[1] Singh, A., Bacchuwar, K. & Bhasin, A. A survey of OCR applications. *International Journal of Machine Learning and Computing* **2**, 314 (2012).

[2] Day, T. G. & Barranca, N. F. Guidelines for Optimizing Readability of Flat-Size Mail. Tech. Rep. 177, United States Postal Service (2003).

[3] Tesseract. URL https://github.com/tesseract-ocr/tesseract. Accessed: 2022-05-07.

[4] Tesseract-OCR. *How to Train Tesseract 4.00*. URL https://tesseract-ocr.github.io/tessdoc/tess4/TrainingTesseract-4.00.html. Accessed: 2023-06-20.

[5] Bieniecki, W., Grabowski, S. & Rozenberg, W. Image preprocessing for improving ocr accuracy (2007).

[6] Petrescu, R. *et al.* Combining tesseract and asprise results to improve ocr text detection accuracy. *Journal of Information Systems & Operations Management* **13**, 57–64 (2019).

[7] Lat, A. & Jawahar, C. Enhancing ocr accuracy with super resolution (2018).

[8] Kišš, M., Kohút, J., Beneš, K. & Hradiš, M. Importance of textlines in historical document classification (2022).

[9] Sporici, D., Cușnir, E. & Boiangiu, C.-A. Improving the accuracy of tesseract 4.0 ocr engine using

convolution-based preprocessing. *Symmetry* **12**, 715 (2020).

[10] Gener, S., Dattilo, P., Gajaria, D., Fusco, A. & Akoglu, A. Gpgpu-based high throughput image pre-processing towards large-scale optical character recognition (2022).

[11] Mittal, R. & Garg, A. Text extraction using ocr: A systematic review (2020).

[12] Hamad, K. & Mehmet, K. A detailed analysis of optical character recognition technology. *International Journal of Applied Mathematics Electronics and Computers* 244–249 (2016).

[13] Harraj, A. E. & Raissouni, N. Ocr accuracy improvement on document images through a novel pre-processing approach. *arXiv preprint arXiv:1509.03456* (2015).

[14] Koistinen, M., Kettunen, K. & Kervinen, J. How to improve optical character recognition of historical finnish newspapers using open source tesseract ocr engine–final notes on development and evaluation (2020).

[15] Shen, M. & Lei, H. Improving ocr performance with background image elimination (2015).

[16] Brisinello, M., Grbić, R., Pul, M. & Anelić, T. Improving optical character recognition performance for low quality images (2017).

[17] Bui, Q. A., Mollard, D. & Tabbone, S. Selecting automatically pre-processing methods to improve ocr performances (2017).

[18] de Jager, C. & Nel, M. Business process automation: A workflow incorporating optical character recognition and approximate string and pattern matching for solving practical industry problems. *Applied System Innovation* **2**, 33 (2019).

[19] Graves, A. *et al.* A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence* **31**, 855–868 (2008).

[20] Google cloud vision ai. URL https://cloud.google.com/vision/. Accessed: 2023-06-20.

[21] Tesseract-OCR. *Tesseract User Manual.* URL https://tesseract-ocr.github.io/tessdoc/. Accessed: 2023-06-20.

[22] Gonzales, R. C. & Woods, R. E. Digital image processing second edition (2001).

[23] Szeliski, R. *Computer vision: algorithms and applications* (Springer Science & Business Media, 2010).

[24] Otsu, N. A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics* **9**, 62–66 (1979).

[25] Sauvola, J. & Pietikäinen, M. Adaptive document image binarization. *Pattern recognition* **33**, 225–236 (2000).

[26] Prahara, A., Pranolo, A., Anwar, N. & Mao, Y. Parallel approach of adaptive image thresholding algorithm on GPU. *Knowledge Engineering and Data Science* **4** (2022).

[27] Shafait, F., Keysers, D. & Breuel, T. M. Efficient implementation of local adaptive thresholding techniques using integral images (2008).

[28] Tambe, S. B., Kulhare, D., Nirmal, M. & Prajapati, G. Image processing (ip) through erosion and dilation methods. *International Journal of Emerging Technology and Advanced Engineering* **3**, 285–289 (2013).

[29] Gaster, B. R., Howes, L., Kaeli, D. R., Mistry, P. & Schaa, D. *Heterogeneous Computing with OpenCL (Second Edition)*, Ch. Chapter 4 - Basic OpenCL Examples, 65–83 (Morgan Kaufmann, 225 Wyman Street, Waltham, MA 02451, USA, 2013), 1.2 edn.

[30] Aldulaimi, F., Alshakargy, H. *et al.* Execution Speed up of Image Rotation Matrix Using Parallel Technique. *American Academic Scientific Research Journal for Engineering, Technology, and Sciences* **26**, 1–17 (2016).

[31] Sun, W., Lu, Y., Wu, F. & Li, S. Real-time screen image scaling and its GPU acceleration (2009).

[32] Di, C., Tian, X. & Yiying, S. Image scaling algorithm based on GPU parallel processing (2013).

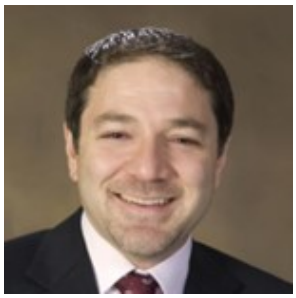
[33] Kraus, M., Eissele, M. & Strengert, M. Ersbøll, B. K. & Pedersen, K. S. (eds) *GPU-Based Edge-Directed Image Interpolation.* (eds Ersbøll, B. K. & Pedersen, K. S.) *Image Analysis*, 532–541 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007).

Serhan Gener is a Ph.D. student in the Electrical & Computer Engineering department at the University of Arizona. He received his BS and MS degrees in Computer Engineering from Yeditepe University, Istanbul, Turkey, in 2015 and 2017, respectively. His research interests include reconfigurable computing, embedded systems, heterogeneous computing, image processing, and software security.

Parker Dattilo is an M.S. student in the Electrical & Computer Engineering department at the University of Arizona. He also earned his B.S. degree in Electrical & Computer Engineering at the University of Arizona. His research interests are design, architectural exploration, and hardware emulation of neuromorphic and reconfigurable computing systems.

Dhruv Gajaria received his M.S. and Ph.D. in Electrical and Computer Engineering from the University of Arizona in 2019 and 2023, respectively, and his B.Eng in Electronics Engineering from the University of Mumbai, India, in 2017. He is a Post-Doctoral Research Associate with the High Performance Computing Group at Pacific Northwest National Lab, USA. His research interests include hardware-software co-design, computer architecture, simulation and performance analysis, and domain-specific architectures.

Alexander Fusco is an M.S. student at the University of Arizona in the Electrical & Computer Engineering department. He received his BS in Computer Engineering at the University of Wisconsin-Madison. His research interests include reconfigurable computing and heterogeneous scheduling.



Ali Akoglu received his Ph.D. degree in Computer Science from the Arizona State University in 2005. He is a Professor in the Department of Electrical & Computer Engineering and BIO5 Institute at the University of Arizona. He is the site-director of the NSF Industry-University Cooperative Research Center on Cloud and Autonomic Computing. His research focus is on high performance computing and non-traditional computing architectures.