TENCON 2023 - 2023 IEEE Region 10 Conference (TENCON)
31 Oct - 3 Nov 2023. Chiang Mai, Thailand

Fast Data Augmentation for Scene Text Recognition using CUDA

David Angelo Piscasio
Electrical and Electronics Engineering Institute
University of the Philippines
Quezon City, Philippines
david.piscasio@eee.upd.edu.ph

Rowel Atienza
Electrical and Electronics Engineering Institute
University of the Philippines
Quezon City, Philippines
rowel@eee.upd.edu.ph

***Abstract*—Scene Text Recognition (STR) is a task in computer vision that is used to read texts in natural scene images. STR currently suffers from data distribution shift due to the lack of large real datasets for training. Data augmentation is a method that has been used in multiple studies to address this issue. However, performing augmentation also introduces computational overhead during training. In this paper, we propose FastSTRAug, a CUDA-based library of 36 augmentation functions specifically designed for STR. When executed through varying image sizes, FastSTRAug is observed to be significantly faster over its serial counterpart in most functions, reaching up to 380× speedup on larger images.**

***Index Terms*—scene text recognition, data augmentation, CUDA**

I. Introduction

Texts are present everywhere. They are seen in documents, logos, street signs, product labels, clothing, electronic devices, etc. These provide useful information to aid us in our daily activities. It helps us know what we are buying from the market, which lane we should stay on the road, or where our favorite restaurant is located. Because of this, scene text recognition (STR), or the task of reading texts from scene images, has gained a lot of interest in computer vision.

STR is a challenging problem since texts from scene images are naturally irregular and inconsistent [2]. Moreover, STR lacks large real datasets, leading to the trend of training STR models on large synthetic datasets such as MJSynth [8] and SynthText [9] instead. This is not ideal since synthetic datasets are not as diverse as real scene text data, which leads to models trained on synthetic data performing poorly when tested on real images that are more complex and varied [5].

To improve STR model performance, multiple studies attempt to include data augmentation as part of training in order for synthetic data to better resemble irregularities found in real scene text data [1], [6], [7]. While proven effective in improving accuracy, augmentation also requires various computations to be performed. Thus, computational overhead is incurred and the time to train STR models becomes longer.

While multiple studies have focused on improving text recognition accuracy through data augmentation, there are none that focus on improving the speed of STR data augmentation. Thus, this paper proposes the development of a GPU-accelerated STR data augmentation library through Compute Unified Device Architecture (CUDA) Python libraries.

FastSTRAug aims to implement 36 augmentation functions based on STRAug [1], which is an earlier study that focuses on the augmentation of STR data, but is designed to work on the CPU. Each function performs the augmentation based on the input parameters `img`, `mag`, and `prob`, which dictates the input image, the magnitude of transformation, and the probability of transformation respectively. Similar to STRAug, the 36 functions are distributed into 8 logical groups: 1) *Warp*, 2) *Geometry*, 3) *Noise*, 4) *Blur*, 5) *Weather*, 6) *Camera*, 7) *Pattern*, and 8) *Process*, based on the functions' nature, origin, and impact. Results show that FastSTRAug provides significant speedup on most data augmentation functions especially with larger images, compared to its serial counterpart in STRAug.

II. Related Work

Scene text recognition is an active research field in computer vision that aims to recognize texts from natural scene images. Recent studies have been mostly centered in improving STR model architecture, with the development of models such as PARSeq [10], MATRN [11], S-GTR [12], CDistNet [13], and DPAN [14].

However, aside from fine-tuning model architecture, STR also faces the problem of lacking publicly available large real datasets for training. This scarcity is attributed to high labeling costs incurred in generating real datasets [3]. As an alternative, efforts have been made to generate large and synthetic datasets for model training instead [4]. However, their synthetic nature causes it to have less diversity and irregularity than what is present in real data, a manifestation of data distribution or domain shift.

In addressing distribution shift, data augmentation has been used by various STR studies. Data augmentation is a technique of performing transformations and manipulations on input data to better mimic irregularities and diversities in real scene text data [1]. Through this, the gap between synthetic training data and real test data becomes more narrow and the recognition accuracy of STR models increases.

Meng et al. [7] developed a sample-aware data augmentor for STR that aims to balance the under-diversity caused by affine transformations and the over-diversity caused by elastic transformations, providing accuracy gains of up to 4.1% when tested on real data. Luo et al.'s *Learn to Augment* [6] is a data augmentation method for text recognition that focuses on spatial transformations. This method leads to accuracy gains of up to 4.5% on irregular text datasets. While these methods offer significant accuracy gains on STR models, they both introduce an additional augmentation network integrated into the recognition model, leading to significantly longer training time. Atienza's STRAug [1] is a library of diverse functions such as image blurring, weather conditions, camera sensor variations, and other augmentations that cater a wide range of image corruptions observed in natural scenes. Moreover, STRAug provides significant accuracy gains of up to 2.10% even without the introduction of additional network parameters to the recognition model.

In performing data augmentation, various numerical computations are performed, leading to computational overhead and added training time. Recent developments have been made to accelerate scientific computing through GPU programming, since GPUs are capable of high performance computing due to its highly parallel nature and increasing computational power [15]. With the development of NVIDIA Compute Unified Device Architecture (CUDA), GPU acceleration has become more accessible for developers, making it more ideal for high performance computing [16], [17].

Various applications in image processing make use of GPU programming and CUDA to develop parallel programs that achieve significant speedups over their serial counterparts.

Fung and Mann [23] presented a comparison on CPU and CUDA implementations of the Lucy-Richardson Deconvolution, which is commonly used in restoring blurred images. The GPU implementation achieved speedups of 9.8× without damping and 21× with damping. Focusing on data augmentation, Vila-Blanco et al. [24] developed IDALib, a Python library for efficient general image augmentation on GPU. It was observed that IDALib achieves up to 18.03× speedup compared to CPU-implemented data augmentation.

To the best of our knowledge, there is a lack of study in GPU utilization for performing STR data augmentation. While GPU-accelerated data augmentation libraries such as IDALib [24] are available, they are designed for general image augmentation. Using these libraries for STR may cause certain elements in text images to be heavily distorted, causing poor model performance due to loss of information. Thus, this study aims to address this gap by developing an accelerated STR data augmentation library using CUDA Python, and analyzing its effects on improving the speed of data augmentation for STR.

III. Methodology

In making data augmentation more time-efficient, we propose FastSTRAug, a CUDA Python alternative of STRAug, which is a data augmentation library designed for STR. STRAug [1] is chosen as reference for FastSTRAug since it caters a diverse set of augmentation methods, while allowing great flexibility and reproducibility for reimplementation and comparison. The 8 logical groups and 36 augmentation functions of FastSTRAug is discussed further in this section. In general, the algorithm used for performing augmentations in FastSTRAug is adopted from Atienza [1]. However, we use alternative CUDA Python libraries instead in order to accelerate numerical computations and image processing for the augmentation functions. Other differences in the algorithm and implementation between our method and STRAug [1] are also further discussed in this section.

For reference, the input image used in visualizing the FastSTRAug data augmentations is displayed in Fig. 1.

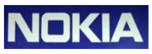

Fig. 1. Source image for data augmentations

A. Warp

The *Warp* logical group contains the *Curve*, *Distort*, and *Stretch* augmentation functions. These augmentation functions mimic elastic deformations found in clothing, street signs, product labels, logos, etc. The thin plate spline (TPS) [25] algorithm is used to perform warping through pixel movement according to source and destination control points. The amount of warping performed is dictated by the magnitude of augmentation set by the user. These augmentations are visualized using FastSTRAug as seen in Fig. 2.

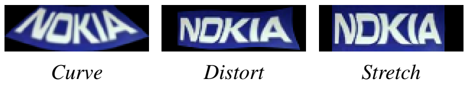

Fig. 2. *Warp* group augmentations

Kornia [21] and PyTorch [18] are used as the primary libraries in performing CUDA-acceleration of the *Warp* group. PyTorch is used for data handling as images are converted into Torch CUDA tensors, while Kornia is used for performing TPS transformation through CUDA.

B. Geometry

The *Geometry* group is designed to perform perspective and affine transformations with the *Perspective*, *Rotation*, and *Shrink* functions. Perspective transformation is used for the *Perspective* augmentation, while TPS is used for the *Shrink* function. The *Rotation* function performs image rotation with the angle uniformly sampled from the range θ_{min} to θ_{max}, depending on the augmentation magnitude. Fig. 3 visualizes the *Geometry* group augmentations.

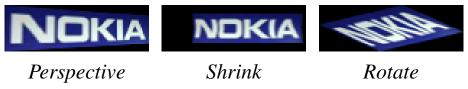

Fig. 3. *Geometry* group augmentations

To enable CUDA-acceleration in the *Geometry* group, both Kornia [21] and PyTorch [18] are used. Similar to the *Warp* group implementation, Kornia is used for estimating and performing both Perspective and TPS transformations. On the other hand, PyTorch is used for data handling in CUDA and for performing *Rotation* along the earlier mentioned range.

C. Pattern

In order to mimic patterns on scene texts caused by multi-line LED displays, dot-matrix printed documents, or texts overlapped by certain patterns, the *Pattern* group is implemented. The group includes 5 grid patterns: *Grid*, *VGrid*, *HGrid*, *RectGrid*, and *EllipseGrid*. The data augmentation magnitude will determine the number of lines drawn over the image. The augmentation group is visualized using FastSTRAug in Fig. 4.

Line drawing on images is performed by changing the pixel value of the line coordinates to the desired color. Thus, *Pattern* augmentations on the CPU is already highly time-efficient and performing it in CUDA-enabled libraries such as Kornia [21] would not make it faster. Because of this, it is decided to adopt the STRAug implementation for these functions and transfer the image tensor over to the GPU memory once the grid patterns have been drawn.

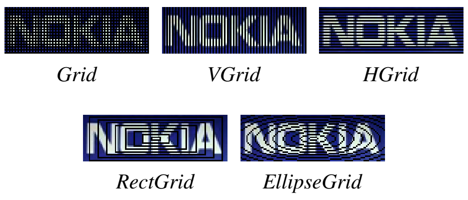

Fig. 4. *Pattern* group augmentations

D. Noise

Since STR data comes from natural scenes, it is common for images to be corrupted due to noise. Thus, the *Noise* group is designed to implement four types of noise corruption on input images, namely *GaussianNoise*, *ShotNoise*, *ImpulseNoise*, and *SpeckleNoise*. The amount of noise injected to the input image is affected by the data augmentation magnitude. Images augmented through the *Noise* group is displayed in Fig. 5.

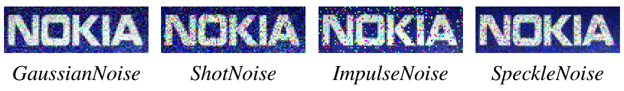

Fig. 5. *Noise* group augmentations

As STRAug [1] uses NumPy to implement the algorithms for all functions in the *Noise* group, FastSTRAug utilizes its CUDA-counterpart, CuPy [19], to accelerate these computations. In addition, the cuCIM [20] library of the RAPIDS suite is used to accelerate the scikit-image functions used to implement the *ImpulseNoise* function.

E. Blur

Due to irregularities in imaging, blurring is commonly seen in natural scene text images. In order to mimic this in synthetic data, the *Blur* group contains *GaussianBlur*, *DefocusBlur*, *MotionBlur*, *GlassBlur*, and *ZoomBlur* functions. The amount of blurring on the images is also affected by the user-defined augmentation magnitude parameter. A simulation of the *Blur* augmentations are seen in Fig. 6.

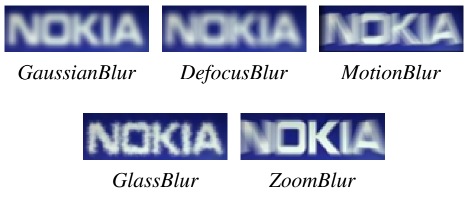

Fig. 6. *Blur* group augmentations

GaussianBlur and *ZoomBlur* is accelerated by performing CUDA tensor operations in PyTorch [18]. *DefocusBlur* uses CuPy [19] and cuSignal [20] for performing computations in CUDA. The `motion_blur` filter of Kornia [21] is utilized by *MotionBlur* to accelerate the STRAug implementation. The *GlassBlur* implementation is adopting the STRAug implementation on the CPU. This is because the implemented algorithm performs pixel swapping through a sequential nested for-loop, which is observed to be multiple times slower on the GPU than on the CPU. Moreover, since the algorithm is inherently sequential, parallelizing it through a custom kernel will not result to the desired effect similar to the STRAug implementation.

F. Weather

The *Weather* group is designed to augment input images to simulate different weather conditions, including *Fog*, *Snow*, *Frost*, *Rain*, and *Shadow*. The data augmentation magnitude determines the amount of effect that the weather condition simulation has on the input image. Sample scene text images augmented with the *Weather* group are shown in Fig. 7.

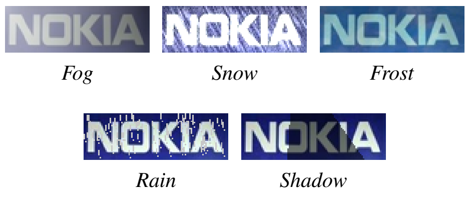

Fig. 7. *Weather* group augmentations

The *Fog* implementation uses CuPy [19] to accelerate NumPy operations. *Snow* and *Frost* use PyTorch [18] to perform intermediary CUDA tensor operations, while *Snow* also uses Kornia [21] to perform motion blurring in CUDA. Kornia's `RandomRain` function is used to implement *Rain* in CUDA. The *Shadow* function is performed by drawing a shadow overlay over the image. As discussed in the *Pattern* group, drawing over the image is already efficient on the CPU and thus, *Shadow* is also adopting the STRAug implementation to keep all FastSTRAug functions time-efficient.

G. Camera

Functions in the *Camera* group focus on augmentations caused by camera effects and image manipulations. This group supports *Contrast*, *Brightness*, *JpegCompression*, and *Pixelate* functions. Similar to other groups, the amount of augmentation done is determined by the user-defined magnitude parameter. *Camera* effects as implemented by FastSTRAug are simulated as shown in Fig. 8.

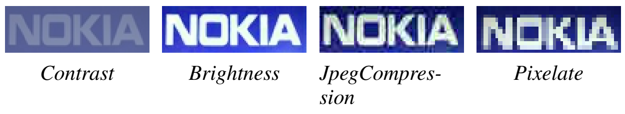

Fig. 8. *Camera* group augmentations

Contrast and *Pixelate* use PyTorch [18] to accelerate tensor operations. *Brightness* uses CuPy [19] to accelerate numerical computations and cuCIM [20] to accelerate scikit-image functions. Finally, *JpegCompression* uses the Python implementation of nvJPEG [26], [27] to perform CUDA-accelerated JPEG operations for image compression.

H. Process

Other image processing augmentations that are not applicable to be included in previously mentioned groups but are still suitable for STR compose the *Process* group. This group includes *Posterize*, *Solarize*, *Invert*, *Equalize*, *AutoContrast*, *Sharpness*, and *Color*. While all other functions in the group have the augmentation magnitude defined by the user, the *Invert*, *AutoContrast*, and *Equalize* functions only have one augmentation level. The *Process* group augmentations are shown in Fig. 9.

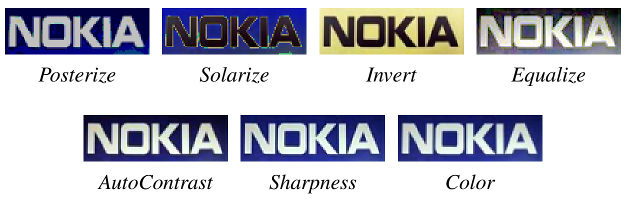

Fig. 9. *Process* group augmentations

Conveniently, PyTorch [18] supports CUDA-accelerated functions for all augmentations through its torchvision library, thus this is used for implementing the logical group.

IV. Results, Analysis, and Discussion

FastSTRAug is evaluated through an adoption of Zhang et al.'s framework [22]. In this evaluation, we compare our method to its serial counterpart, STRAug [1], to see if FastSTRAug does perform faster data augmentation operations.

A. Data Augmentation Process

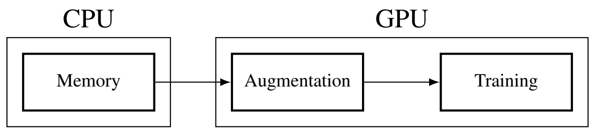

Fig. 10. Data flow for optimizing FastSTRAug

Since FastSTRAug is designed for STR model training, we adopted the data flow by Vila-Blanco et al. [24] as seen in Fig. 10 to further optimize augmentation. We prevent memory transfer overhead by keeping the data in the GPU once it has been augmented so that it can be accessed by the network model when performing GPU-accelerated training.

B. Experimental Setup

In evaluating the performance of FastSTRAug in terms of speed, we compared the execution times of each FastSTRAug function with its direct STRAug counterpart at `mag=2` over varying image sizes, getting the speedup ratio for each. The execution time with the standard deviation was recorded and averaged over 10 iterations. This was performed over three varying image sizes: (1) 168×50, (2) 393×216, and (3) 778×336. In recording the runtimes, the data transfer time from CPU to GPU was included, but the transfer time from GPU back to the CPU was excluded following the data flow in Fig. 10. All tests are run on the following hardware, CPU: *Intel Xeon @ 2.20 GHz 12GB RAM* and GPU: *NVIDIA Tesla T4 16GB GDDR6 VRAM*.

C. Speedup Evaluation

The results and analysis of FastSTRAug's speedup evaluation is done according to each logical group.

TABLE I
Warp Group Evaluation

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
Curve			
168×50	14.22±0.76	3.445±0.051	4.129
393×216	16.39±0.48	3.360±0.13	4.878
778×336	21.36±0.60	4.032±0.11	5.299
Distort			
168×50	1.951±0.056	1.885±0.066	1.035
393×216	16.40±1.4	3.458±0.051	4.743
778×336	48.76±1.8	9.675±0.065	5.039
Stretch			
168×50	2.346±0.13	2.268±0.30	1.034
393×216	31.02±0.56	4.796±0.11	6.468
778×336	72.13±19	7.494±0.13	9.626

Table I presents the results of the speedup evaluation for the *Warp* functions. The speed gains for the *Curve* function is more consistent across varying image sizes, with 4.13× speedup even on the 168×50 image. On the other hand, even though very little speedup is observed on smaller images for *Distort* and *Stretch*, they show significant speedups of up to 9.626× on larger images (778×336).

TABLE II
Geometry Group Evaluation

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
Rotate			
168×50	8.201±0.70	1.096±0.024	7.486
393×216	10.2±0.37	1.262±0.018	8.08
778×336	16.19±1.0	1.826±0.029	8.869
Perspective			
168×50	0.5375±0.044	0.9488±0.047	0.5665
393×216	2.132±0.066	1.537±0.061	1.387
778×336	6.019±0.91	2.931±0.10	2.054
Shrink			
168×50	2.024±0.082	3.473±0.58	0.5826
393×216	16.32±1.9	6.107±0.068	2.672
778×336	50.57±6.5	11.08±1.8	4.563

As seen in Table II, the *Rotate* function leverages the use of CUDA the most among the functions in the *Geometry* group as it is observed to have speedups of at least 7.486× (168×50). On the contrary, the performance of both *Perspective* and *Rotate* is observed to be poorer in smaller images (168×50) with FastSTRAug taking almost twice the time to execute compared to STRAug. However, it is seen to improve to up to 4.563× faster as the input image size becomes larger.

TABLE III
Pattern Group Evaluation

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
Grid			
168×50	0.8840±0.020	0.8508±0.028	1.039
393×216	1.816±0.071	2.107±0.37	0.8618
778×336	5.713±0.15	5.806±0.30	0.9840
VGrid			
168×50	0.7504±0.027	1.423±1.4	0.5271
393×216	1.533±0.051	1.502±0.035	1.021
778×336	5.015±0.14	5.009±0.26	1.001
HGrid			
168×50	0.5614±0.044	0.6198±0.18	0.9058
393×216	1.323±0.043	1.309±0.025	1.011
778×336	4.742±1.2	3.929±0.089	1.207
RectGrid			
168×50	0.4579±0.017	0.4693±0.019	0.9758
393×216	1.149±0.037	1.157±0.040	0.9933
778×336	3.699±0.064	3.759±0.17	0.9841
EllipseGrid			
168×50	0.5504±0.015	0.5379±0.018	1.023
393×216	1.461±0.049	1.694±0.31	0.8627
778×336	5.844±0.64	5.640±0.94	1.036

The observed execution times and speedup for the *Pattern* group is displayed in Table III. As expected, the speedup ratios observed are all close to 1, since both use the same implementation as discussed.

The results of the *Noise* group, as seen in Table IV, shows significant speedup for most of the group's functions. *GaussianNoise*, *ShotNoise*, and *SpeckleNoise* all exhibit speedups over STRAug across all image sizes, ranging from 2.895× to as high as 46.64×. *ImpulseNoise*, on the other hand, is observed to be slower when performed on the 168×50 image, but improves steadily with increasing image size. This is believed to be caused by cuCIM's `random_noise` function, which is used in *ImpulseNoise*, being slower in smaller images.

Out of all the function groups, the *Blur* group is observed to have the highest speedup increase especially in larger images. As seen in Table V, speedups of up to 380.9× is observed with the *GaussianBlur* function, with slightly lower but still significant speedups observed in *DefocusBlur*, *MotionBlur*, and *ZoomBlur*. As expected, the *GlassBlur* function is observed to have little-to-no difference with STRAug since they both have the same implementation.

TABLE IV
NOISE GROUP EVALUATION

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
	GaussianNoise		
168×50	1.063±0.16	0.3674±0.017	2.895
393×216	6.369±0.49	0.5431±0.022	11.73
778×336	19.88±1.5	0.8563±0.029	23.22
	ShotNoise		
168×50	2.139±0.14	0.426±0.021	5.022
393×216	15.4±1.3	0.7617±0.23	20.21
778×336	44.49±2.5	0.9538±0.059	46.64
	ImpulseNoise		
168×50	2.495±0.020	4.494±0.010	0.5552
393×216	10.21±0.24	4.826±0.010	2.115
778×336	29.01±0.41	6.891±0.050	4.210
	SpeckleNoise		
168×50	0.9882±0.032	0.3693±0.012	2.676
393×216	6.848±0.76	0.5199±0.014	13.17
778×336	19.08±1.2	0.8339±0.017	22.88

As displayed in Table VI, the *Snow* function is observed to gain the highest speedups of at most 43.28× over the STRAug implementation. On the other hand, moderate speedups were observed in *Frost* and *Rain*. The *Fog* function is slower in the 168×50 image, but is seen to be faster in larger images (393×216 and 778×336). As expected, execution times between STRAug and FastSTRAug's *Shadow* are relatively close since they have the same implementation.

TABLE V
Blur GROUP EVALUATION

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
	GaussianBlur		
168×50	1.86±0.12	0.6833±0.091	2.722
393×216	1966±1.3	6.012±0.025	327.0
778×336	7117±24	18.69±0.24	380.9
	DefocusBlur		
168×50	1.64±0.062	0.9709±0.066	1.689
393×216	14.76±0.60	1.678±0.044	8.794
778×336	30.24±1.4	1.591±0.38	19.01
	MotionBlur		
168×50	29.65±6.7	2.882±0.16	10.29
393×216	178.6±6.2	2.872±0.15	62.17
778×336	437.7±23	4.117±0.15	106.3
	GlassBlur		
168×50	213.5±19	213.3±18	1.001
393×216	2351±28	2451±38	0.959
778×336	7319±60	7304±56	1.002
	ZoomBlur		
168×50	4.452±0.15	1.122±0.066	3.968
393×216	33.24±0.94	1.736±0.052	19.15
778×336	153.8±4.8	4.524±0.26	33.99

Table VII shows that functions of the *Camera* group exhibit speedups across all image sizes. *Brightness* has the largest speedup at 110.6×. Other functions, *Contrast*, *JpegCompression*, and *Pixelate*, while gaining smaller speedups than *Brightness*, is still faster over its STRAug counterparts.

Finally, the execution times and speedups of the *Process* group are shown in Table VIII. As observed, performing these operations on the CPU is already time-efficient especially on smaller images. Thus, it is expected that FastSTRAug will not perform significantly better on these smaller data, which is evident in the *Equalize*, *AutoContrast*, *Sharpness*, and *Color* functions. However, these functions still exhibit speedups over STRAug on larger-sized images.

TABLE VI
Weather GROUP EVALUATION

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
	Fog		
168×50	6.793±0.20	11.82±0.26	0.5749
393×216	35.08±1.7	18.38±0.52	1.909
778×336	156.4±19	15.38±1.4	10.17
	Snow		
168×50	24.57±2.6	3.355±1.2	7.323
393×216	123.9±5.8	4.289±0.15	28.88
778×336	337.6±22	7.800±0.26	43.28
	Frost		
168×50	25.45±12	13.89±6.8	1.832
393×216	22.56±8.5	13.94±6.9	1.618
778×336	48.26±12	14.11±7.3	3.418
	Rain		
168×50	4.369±0.33	3.107±0.31	1.406
393×216	3.474±0.18	2.823±0.09	1.231
778×336	7.388±0.42	6.008±1.7	1.230
	Shadow		
168×50	0.637±0.080	0.6152±0.026	1.035
393×216	1.524±0.078	1.521±0.072	1.002
778×336	6.047±0.69	6.596±1.7	0.9167

Overall, it is clear that FastSTRAug outperforms STRAug in most augmentation functions. This is highly evident in more complex augmentation methods such as *GaussianBlur*, *MotionBlur*, and *Brightness* where the STRAug implementation takes hundreds to thousands of milliseconds to perform.

TABLE VII
Camera GROUP EVALUATION

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
	Contrast		
168×50	0.945±0.042	0.3586±0.020	2.635
393×216	5.519±0.094	1.031±0.030	5.354
778×336	23.69±0.87	3.045±0.15	7.778
	Brightness		
168×50	4.482±0.69	0.5132±0.019	8.734
393×216	35.76±1.2	0.6826±0.027	52.38
778×336	111.4±2.8	1.007±0.033	110.6
	JpegCompression		
168×50	1.161±0.55	0.8184±0.023	1.419
393×216	4.748±0.85	1.837±0.041	2.585
778×336	10.54±0.30	4.417±0.093	2.387
	Pixelate		
168×50	0.4372±0.024	0.3774±0.02	1.159
393×216	2.068±0.060	1.029±0.043	2.011
778×336	5.748±0.24	3.575±1.4	1.608

On the other hand, there are also functions that do not benefit much from CUDA acceleration such as *Equalize* and *GlassBlur* since either the STRAug implementation is already highly time-efficient or the operations needed to perform the augmentation is inherently slower on the GPU.

The general trend observed is that the speedup increases with increasing input data size, except some functions such as *Rain* and *Pixelate*. CUDA-accelerated augmentations are more effective in speeding up the processing of larger data, which we believe is due to factors such as memory transfer having less impact on the overall execution times.

One important observation is that while significant speedup is obtained in most functions, differences in execution times between our method and STRAug [1] is within milliseconds only, which may seem insignificant when evaluated with individual images. However, we believe that our method becomes useful if we consider commonly used STR datasets, MJSynth [8] and SynthText [9]. Since these datasets contain millions of images, individual speedups obtained from each augmentation will accumulate, thus decreasing overall time needed to train models with augmentation.

TABLE VIII
Process GROUP EVALUATION

Image Size (px)	STRAug (ms)	FastSTRAug (ms)	Speedup
	Posterize		
168×50	0.3664±0.018	0.1947±0.012	1.881
393×216	1.211±0.066	0.4527±0.022	2.675
778×336	2.962±0.16	0.9201±0.023	3.220
	Solarize		
168×50	0.3738±0.021	0.243±0.0092	1.538
393×216	1.202±0.033	0.5506±0.19	2.184
778×336	3.105±0.34	1.001±0.025	3.103
	Invert		
168×50	0.3314±0.024	0.1735±0.0088	1.910
393×216	1.405±0.28	0.4442±0.021	3.164
778×336	2.861±0.16	0.911±0.0077	3.14
	Equalize		
168×50	0.4821±0.02	1.46±0.068	0.3303
393×216	1.566±0.075	1.84±0.14	0.8513
778×336	3.72±0.42	2.218±0.050	1.677
	AutoContrast		
168×50	0.4917±0.020	0.6394±0.28	0.7689
393×216	1.531±0.060	0.7206±0.028	2.124
778×336	3.628±0.17	1.162±0.034	3.122
	Sharpness		
168×50	0.5612±0.027	0.6088±0.033	0.9217
393×216	4.058±0.13	1.775±0.44	2.286
778×336	10.5±0.47	2.69±0.27	3.903
	Color		
168×50	0.3711±0.020	0.4207±0.12	0.8821
393×216	2.266±0.44	0.5774±0.011	3.925
778×336	4.854±0.34	1.081±0.045	4.489

V. CONCLUSION

In this paper, we develop a CUDA-based STR data augmentation library called FastSTRAug, with the aim of minimizing computational overhead when performing data augmentation on scene text images.

Results show that FastSTRAug is effective in speeding up augmentation in most functions. Moreover, FastSTRAug augmentation in larger images usually leads to better speedup. However, some functions do not benefit much from CUDA acceleration since there are operations that are still inherently slow on the GPU.

Future work should focus on analyzing the accuracy gains of FastSTRAug compared to STRAug and other data augmentation methods when trained with STR models, along with further optimization of the functions, especially those that were observed to be slower than their serial counterparts.

REFERENCES

[1] R. Atienza, 'Data augmentation for scene text recognition', in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 1561–1570.

[2] C. Chen, D.-H. Wang, and H. Wang, 'Scene character and text recognition: The state-of-the-art', in *Image and Graphics: 8th International Conference, ICIG 2015, Tianjin, China, August 13–16, 2015, Proceedings, Part III*, 2015, pp. 310–320.

[3] J. Baek et al., 'What is wrong with scene text recognition model comparisons? dataset and model analysis', in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 4715–4723.

[4] J. Baek, Y. Matsui, and K. Aizawa, 'What if we only use real datasets for scene text recognition? toward scene text recognition with fewer labels', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 3113–3122.

[5] V. Loginov, 'Why you should try the real data for the scene text recognition', *arXiv preprint arXiv:2107.* 13938, 2021.

[6] C. Luo, Y. Zhu, L. Jin, and Y. Wang, 'Learn to augment: Joint data augmentation and network optimization for text recognition', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13746–13755.

[7] G. Meng et al., 'Sample-aware Data Augmentor for Scene Text Recognition', in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 3978–3985.

[8] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman, 'Synthetic data and artificial neural networks for natural scene text recognition', *arXiv preprint arXiv:1406. 2227*, 2014.

[9] A. Gupta, A. Vedaldi, and A. Zisserman, 'Synthetic data for text localisation in natural images', in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2315–2324.

[10] D. Bautista and R. Atienza, 'Scene text recognition with permuted autoregressive sequence models', in *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXVIII*, 2022, pp. 178–196.

[11] B. Na, Y. Kim, and S. Park, 'Multi-modal text recognition networks: Interactive enhancements between visual and semantic features', in *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXVIII*, 2022, pp. 446–463.

[12] Y. He et al., 'Visual semantics allow for textual reasoning better in scene text recognition', in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022, vol. 36, pp. 888–896.

[13] T. Zheng, Z. Chen, S. Fang, H. Xie, and Y.-G. Jiang, 'CDistNet: Perceiving multi-domain character distance for robust text recognition', *arXiv preprint arXiv:2111. 11011*, 2021.

[14] Z. Fu, H. Xie, G. Jin, and J. Guo, 'Look back again: dual parallel attention network for accurate and robust scene text recognition', in *Proceedings of the 2021 International Conference on Multimedia Retrieval*, 2021, pp. 638–644.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, 'GPU computing', *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[16] N. Sunitha, K Raju, and N. N. Chiplunkar, "Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead", in 2017 International Conference on Inventive Communication and Computational Technologies (ICICCT), IEEE, 2017, pp. 211–215. Technologies (ICICCT), IEEE, 2017, pp. 211–215.

[17] D. Luebke, 'CUDA: Scalable parallel programming for high-performance scientific computing', in *2008 5th IEEE International Symposium on Biomedical Imaging: from nano to macro*, 2008, pp. 836–838.

[18] A. Paszke et al., 'PyTorch: An imperative style, high-performance deep learning library', *Advances in neural information processing systems*, vol. 32, 2019.

[19] R. Nishino and S. H. C. Loomis, 'CuPy: A NumPy-compatible library for NVIDIA GPU calculations', *31st Conference on Neural Information Processing Systems*, vol. 151, no. 7, 2017.

[20] RAPIDS API Docs. [Online]. Available: https://docs.rapids.ai/api.

[21] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski, 'Kornia: an open source differentiable computer vision library for PyTorch', in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, pp. 3674–3683.

[22] N. Zhang, Y.-S. Chen, and J.-L. Wang, 'Image parallel processing based on GPU', in *2010 2nd International Conference on Advanced Computer Control*, 2010, vol. 3, pp. 367–370.

[23] J. Fung and S. Mann, 'Using graphics devices in reverse: GPU-based image processing and computer vision', in *2008 IEEE international conference on multimedia and expo*, 2008, pp. 9–12.

[24] N. Vila-Blanco, R. R. Vilas, and M. J. Carreira, 'IDALib: a Python library for efficient image data augmentation', in *2022 10th European Workshop on Visual Information Processing (EUVIP)*, 2022, pp. 1–7.

[25] F. L. Bookstein, 'Principal warps: Thin-plate splines and the decomposition of deformations', *IEEE Transactions on pattern analysis and machine intelligence*, vol. 11, no. 6, pp. 567–585, 1989.

[26] nvJPEG Libraries. [Online]. Available: https://developer.nvidia.com/nvjpeg.

[27] Z. Qinghui and O. Batchelor, NvJpeg - Python, https://github.com/UsingNet/nvjpeg-python/, 2021.