Optimization of Gabor Filters by Employing NVIDIA GPUs in Python

Conner McInnes and Shadi Alawneh
Oakland University, Rochester Hills, United States
Email: {cmcinnes, shadialawneh}@oakland.edu

***Abstract*—In this paper, a through rigorous testing and benchmarks the efficacy of the utilization of CUDA's GPU accelerated libraries for a Gabor filter was examined. Following a series of benchmarks, the change in computational time between a program that applied a set of Gabor kernels to images using CuPy and SciPy was recorded. The benchmark's results provided statistical evidence in favor of future utilization of CuPy's GPU accelerated libraries in such a program. With this data in hand, further work can be carried out that leverages a GPU to be incorporated in a compression algorithm using the Gabor transform. This will offer a fast compression technique that allows the fine tuning of the compression ratio of a target image.**

***Index Terms*—GPU, CUDA, Gabor filter, image filtering, Python**

I. Introduction

The Gabor filter is often used in image processing as a means to perform edge detection, however it's uses can stretch much further beyond. The Gabor filter is derived from the convolution of a fast Fourier transformed Gaussian function and a fast Fourier transformed sinusoidal wave. These functions allow the Gabor filter to be able to be fine-tuned to a precise degree allowing for the manipulation of: wavelength, orientation, phase offset, standard deviation, and ellipticity.

While Gabor filters and the Gabor transform are somewhat related, the terms are not interchangeable. To perform a Gabor transform a Gabor filter is utilized, however, the Gabor transform is used specifically to analyze the relationship between an image's spatial and frequency domain [1].

The Gabor transform in turn can be utilized in an image compression algorithm as way to isolate high frequency noise [2]. This allows for the fine tuning of the compression ratio providing an immense amount of control over the degree to which the image's quality and file size is reduced.

II. Related Works

Numerous other authors have undertaken a multitude of projects that have made use of the Gabor transform predominantly in the fields of Artificial Intelligence (AI), signal analysis, and image processing. Wiesmeyr, Holighaus, and Søndergaard sought to identify any possible alternative method for performing the Gabor transform [3]. The work performed for this thesis seeks to accomplish a similar goal. This goal is to deduce if an alternative method using a Graphics Processing Unit (GPU) as opposed to the standard Central Processing Unit (CPU) based approach for computing the Gabor transform would be more efficient. This is an approach that the referenced paper had not considered.

The topic of the Gabor transform has been explored in depth by Baxter [4]. While Baxter's research is not extremely current it has served as an excellent foundation laying out the numerous steps for the implementation of a compression system using a discrete wavelet transform, in this case the Gabor transform [4]. The use of discrete wavelet transforms is still a great modern solution to the compression of images. Baxter's work was able to break down the various techniques and steps for performing the Gabor transformation to filter an image [4]. By employing similar tactics into a program, the work done for this project would attempt to optimize this operation further by expediting the computational time of such an operation.

In addition, Wang and Shi, seem to have already expanded into this topic area. However, their research is antiquated using old GPUs and a method of filtering that differs from this project [5]. By researching into the practicality of such an approach with modern GPUs and a simpler algorithm, the work done during this project could identify a massive computational leap in the process of compressing and filtering images. Images greatly benefit from advances in discrete wavelet transform compression algorithms because they allow for the fine tuning of a compression ratio. This is ideal since typical lossy compression methods used for JPEGs can result in too much detail being lost from said images rendering them much blurrier than desired.

While these sources are a few years old this research is by no means invalid or still not an issue in modern times as shown by the research teams of Zhao, Tao, Li, and Wang as well as He *et al.* [6], [7]. By devising a more efficient algorithm for compressing and filtering images, the cost associated in the storage of a large volume of images can be alleviated. This is on top of a much faster processing time for said images. In addition, the steps laid out for this program would be provided in verbose detail along with providing the open-source code. This in itself

Manuscript received May 31, 2021; revised September 2, 2021.

is distinctive as little open-source code exists in this topic area.

III. Objective of Research

The objective of this project is to ascertain the benefits, if they do indeed exist, of a Compute Unified Device Architecture (CUDA) accelerated program using the Gabor transform created for the purpose of compressing and filtering images. This was accomplished by specifically targeting a computational speed up during the creation of the Gabor filters used in the Gabor transform. A program that is able to apply a Gabor filter to an image would then be required. In addition, a separate algorithm that incorporates a method to calculate the filtered image faster would also be required. This would allow for a direct comparison to be performed between the two implementation that would provide statistical evidence in the form of a benchmark pertaining to the relative speed of each algorithm.

The programming language picked for this research was Python. Python is a scripting language that is often used alongside MATLAB for image processing due to the numerous libraries that are available for use and the relative ease of implementation of said libraries. Work was performed on creating an implementation using C++ and CUDA, however numerous difficulties with the application of the filter so development was shifted to Python.

The libraries chosen for the implementation of this program were SciPy and CuPy. SciPy is an open-source Python library with a multitude of functions [8]. Specifically for the purpose of this research the convolution function was of importance as this would apply the filter to the target image. This library also employs very similar methods to CuPy making them direct comparisons when comparing the computational speed.

Likewise, CuPy is an open-source Python library. Unlike SciPy, CuPy is able to incorporate CUDA's GPU accelerated libraries such as cuBLAS, cuDNN, cuFFT, and more to provide Python with an expansive suite of GPU integrated functions [9]. CuPy is actively under development and supports NVIDIA's most recent versions of the CUDA toolkit, but has not completely integrated all of CUDA's functions from its numerous libraries. This is one of the few CUDA libraries that is actively being supported and is recommended by now defunct and out of date CUDA libraries due to the wide range of functions that have been incorporated. The library is also being supported by Nvidia, the creators of CUDA.

IV. Programs

Two programs were created in Python with a discrete purpose. One program was solely responsible for creating the Gabor filter banks that would be utilized by the other program which would apply the filter bank to an input image and benchmark the results.

A. *Creation of Gabor Kernels*

This program allows the user to adjust the many aspects that make up a Gabor filter. This includes the resolution of the filter, the standard deviation of the root gaussian function, the angle of the sinusoid, the wavelength of sinusoid, the ellipticity of the sinusoid, and the phase offset of the sinusoid. The Python library OpenCV is used to create the Gabor filter according to the input parameters. The program creates a filter folder in the directory of the program if it does not already exist, and gives it a unique name in the form of "gaborFilter#" where the number sign starts at zero and is incremented by one until a unique value is found. The file saved is in the format of npy using a function that saves NumPy arrays in CuPy. In addition, a csv file is created if it does not already exist or is appended to if it does exist. The data wrote to this file records what filter has what parameters so the filter bank can more easily be examined. Fig. 1 and Fig. 2 show two example filters created using this program and shown using the library Matplotlib.

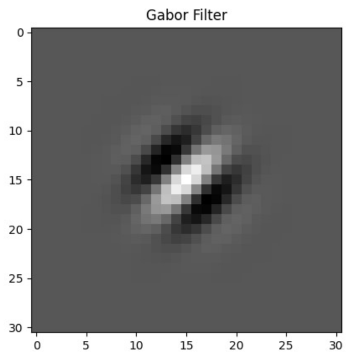

Figure 1. Gabor filter with parameters: Resolution = 30, Standard Deviation = 3, Angle = π/4, Wavelength = π/4, Ellipticity = 1.0, Phase Offset = 0.

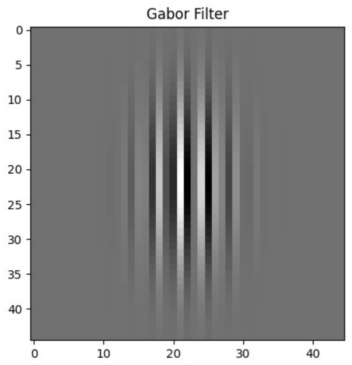

Figure 2. Gabor filter with parameters: Resolution = 45, Standard Deviation = 4, Angle = 0, Wavelength = π/2, Ellipticity = 0.5, Phase Offset = 15.

B. *Method for Applying Gabor Kernels*

This program applies the filter banks already created by searching for the "Filters" folder in the same directory of the Python file. This is the program that is benchmarked and the one that this research is primarily concerned with, although the other program allows for the creation of filter banks to be much easier.

The program requires the user to provide an input image and to enter its name as a string. The program will then search the "Images" folder in the same directory and quit out if the image is not found. If it is the program will continue operating. The image is read using OpenCV and converted to grayscale if it is not already. The image is then converted into a CuPy array. This differs from the standard NumPy array because the GPU requires the allocation of variable and constants separately from the CPU. A function is then called that counts the total number of filters in the "Filters" folder, this is the filter bank. Following this the benchmark for the GPU begins.

The program goes through each filter one-by-one loading it in as a CuPy array. The array is then convolved with the CuPy array of the image and the resulting array is converted into a NumPy array. These steps are also individually benchmarked to see how long each filter individually takes to apply and to convert the result back to the NumPy format. This is repeated for each filter until each of them has created a convolved result with the image. Once this is done the benchmark is ended and the total runtime is calculated by taking the difference of the end and start times.

Next, the CPU benchmark would begin and follow a very similar set of steps as the GPU benchmark. The arrays would be loaded in as NumPy arrays. Following this they would be convolved with the image; however, there is no need to convert the result into the NumPy format. This provides a slight edge to the CPU as the GPU will need to convert the resulting image into a NumPy array for further operation to be performed on it. Nevertheless, these convolutions are also individually benchmarked and recorded so an analysis on how long each filter took to apply can be performed. Once all filters have been applied, the benchmark will be ended and the total runtime will be calculated.

The results for both the GPU and CPU are saved to a csv file for later analysis. This includes the total runtime for the entire filter bank and runtime for each filter application for both the GPU and CPU. For more details about the flowcharts of the programs please check Appendix A.

V. Benchmark Results

Fig. 3, Fig. 4, Fig. 5, and Fig. 6 depict the images used for testing sourced from pixabay. A variety of resolutions were used for the benchmarking to ensure accurate data could be gathered across a variety of input images. These images have a variety of angles and patterns making them well suited for testing Gabor filters in realistic scenarios.

Fig. 7 and Fig. 8 illustrate the results gathered from benchmarking. A filter bank of 36 filters with a constant resolution of 30 was used for benchmarking. From testing the only parameter that caused a major difference in the runtime of the functions was when an assortment of different resolutions was used. A constant resolution was used as this can be achieved by scaling all filters up to the resolution of the largest filter's size.

Fig. 7 highlights the total runtimes for each image at a specific resolution to have all 36 filters applied to the image. As stated earlier, the GPU runtime includes the time it takes to convert the resulting filtered image into a NumPy array so that it may be processed further by the CPU as a necessary step.

Fig. 8 focuses on what percent faster the GPU ran in comparison to the CPU. That is to say an entry of 0% would mean that the programs ran at the exact same speed, whereas 100% faster would mean the GPU was twice as fast or ran in half of the time. Equation 1 shows how these values were calculated.

Figure 3. Test image referred to as Pantheon.

Figure 4. Test image referred to as Tent.

Figure 5. Test image referred to as Castle.

Figure 6. Test image referred to as Lab.

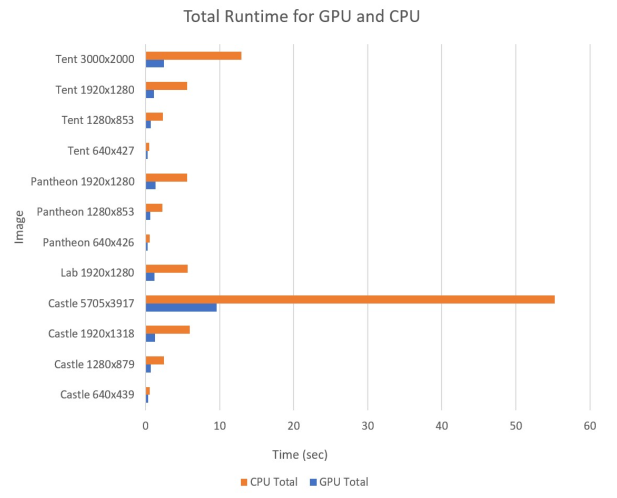

Figure 7. Chart comparing the total runtime in seconds for the GPU and CPU to apply all filters to the test image.

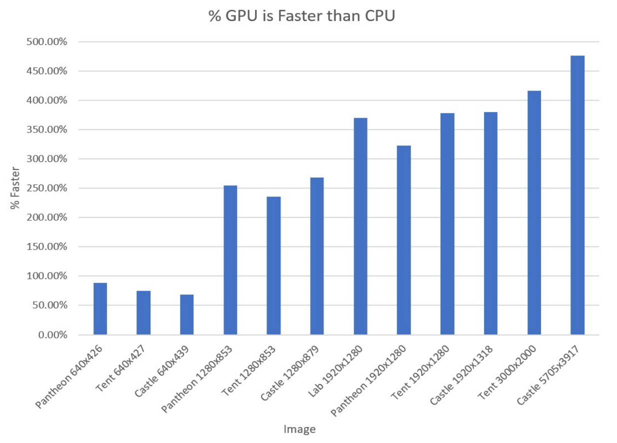

Figure 8. Chart showing the percent increase in computational speed ordered by resolution size.

TABLE I. GPU VS CPU COMPUTATIONAL PERFORMANCE

Image	Performance		
	% Increase	***GPU Runtime (sec)***	***CPU Runtime (sec)***
Castle 640×439	68.23%	0.34568	0.581535
Castle 1280×879	268.17%	0.682639	2.513258
Castle 1920×1318	379.39%	1.251794	6.000939
Castle 5705×3917	475.71%	9.604088	55.29144
Lab 1920×1280	370.10%	1.214026	5.707164
Pantheon 640×426	88.36%	0.292269	0.550507
Pantheon 1280×853	254.62%	0.649599	2.303611
Pantheon 1920×1280	322.80%	1.335696	5.647336
Tent 640×427	74.69%	0.301277	0.526294
Tent 1280×853	235.88%	0.694818	2.333745
Tent 1920×1280	377.60%	1.169873	5.587256
Tent 3000×2000	416.27%	2.503815	12.92643

Table I provides all of the data shown in Fig. 7 and Fig. 8 in a tabular format for the exact data points for both the total run time, as well as, the percent faster.

$$\% \ Increase = (CPU_{time} - GPU_{time})/\ GPU_{time} \quad (1)$$

VI. CONCLUSION

It can be seen from the graphs provided that the GPU is able to much more effectively perform the calculations necessary for filtering the image through convolution in comparison to the CPU. CuPy is able to leverage the GPU's ability to parallel process to a degree in which not even the lowest resolution images are able to be computed on the CPU faster. This shows the computational speed is able to overcome the offset of having to convert the image back to a NumPy formatted array for the CPU to be able to read. Notably, as the image's resolution increases, so too does the speed up provided from using CuPy.

CuPy and SciPy interestingly seemed to speed up as it applied more and more filters. While the first initial filter would take about as long as the SciPy took to filter, CuPy would accelerate much faster as it applied more and more from the filter bank. Within two to three filters being applied in succession CuPy would hit its fastest speeds.

The use of CUDA's libraries to assist in the filtering of images with Gabor filters certainly seems to be a realistic choice that is able to provide consistently faster results when compared to its CPU based approach in SciPy. The use of a filter bank exasperates this and ensures that the GPU will have enough time to ramp up its computational speed. One thing that does cause issues to arise is when the filters have different resolutions to the previous one used. This causes problems in both the GPU and CPU approach and seems to reset the speed up in computational speed that occurs when applying filter in succession. This can be remedied by choosing to utilize a filter bank with a constant resolution.

VII. FUTURE WORK

CuPy is a library that is still under active development and is in fact missing a feature that would assist this program in having the GPU based approach become even faster. One of the features SciPy has is that for N-dimensional arrays it can choose the best method to apply the filter to the image through convolution. From the testing with the images used it consistently appeared that using the fast Fourier transform was the most effective way to apply the filter to the image. CuPy does not yet support this feature for N-dimensional arrays. What currently is implemented is only able to perform a direct convolution for any array that is greater than 1-dimensional. Should this feature be implemented, it is very possible that CuPy could see even greater speed in the process of applying filters to images.

Furthermore, this work serves as the basis to which a Gabor transform can be implemented with the intent to compress images. With this algorithm that is able to compute the convolution of an image and a Gabor filter faster than the CPU based approach, the Gabor transform for compressing images can be made quicker. This will also require the creation of a bit allocation and quantization system that Baxter outlined [4].

APPENDIX FLOWCHARTS

Fig. 9 and Fig. 10 describe the flowcharts of the programs used in this paper.

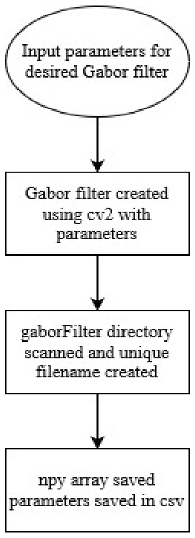
Input parameters for desired Gabor filter
Gabor filter created using cv2 with parameters
gaborFilter directory scanned and unique filename created
npy array saved parameters saved in csv

Figure 9. Flowchart for Gabor filter creation program.

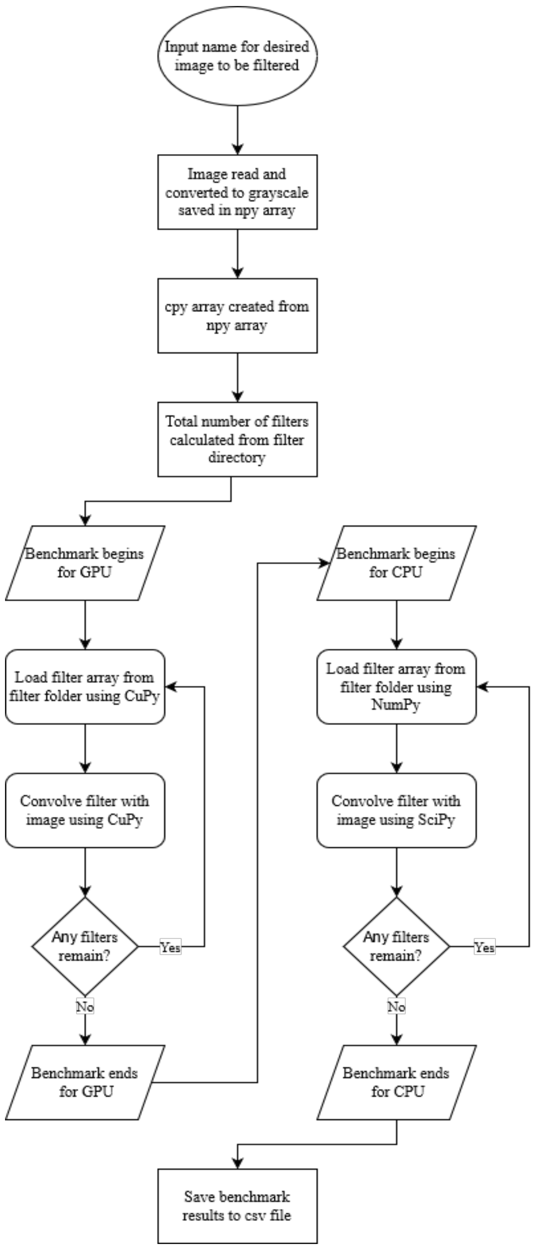
Input name for desired image to be filtered
Image read and converted to grayscale saved in npy array
cpy array created from npy array
Total number of filters calculated from filter directory
Benchmark begins for GPU
Load filter array from filter folder using CuPy
Convolve filter with image using CuPy
Any filters remain?
Yes
No
Benchmark ends for GPU
Benchmark begins for CPU
Load filter array from filter folder using NumPy
Convolve filter with image using SciPy
Any filters remain?
Yes
No
Benchmark ends for CPU
Save benchmark results to csv file

Figure 10. Flowchart for Gabor application program.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

Conner McInnes conducted the research, analyzed the data, wrote the paper; Shadi Alawneh supervised all steps; all authors had approved the final version.

Acknowledgment

The authors wish to thank Michigan Space Grant Consortium. This work was supported by the Michigan Space Grant Consortium (Grant Number: NNX15AJ20H).

References

[1] H. Badgujar. (2013). Re: What’s the difference between Gabor filter and Gabor transform? [Online]. Available: https://www.researchgate.net/post/Whats_the_difference_between_Gabor_filter_and_Gabor_transform/51348cf4e24a465752000003/citation/download

[2] B. Deokate, P. Patil, and S. Majgaonkar, “Image compression using Gabor filter,” *International Journal of Emerging Trends in Electrical and Electronics*, vol. 2, no. 3, pp. 28-32, 2013.

[3] C. Wiesmeyr, N. Holighaus, and P. L. Søndergaard, “Efficient algorithms for discrete gabor transforms on a nonseparable lattice,” *IEEE Transactions on Signal Processing*, vol. 61, no. 20, pp. 5131-5142, Oct. 2013.

[4] R. A. Baxter, “SAR image compression with the Gabor transform,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 37, no. 1, pp. 574-588, Jan. 1999.

[5] X. Wang and B. E. Shi, “GPU implemention of fast Gabor filters,” in *Proc. IEEE International Symposium on Circuits and Systems*, Paris, 2010, pp. 373-376.

[6] Z. Zhao, R. Tao, G. Li, and Y. Wang, “Clustered fractional Gabor transform,” *Signal Processing*, vol. 166, p. 107240, Jan. 2020.

[7] L. He, C. Liu, J. Li, Y. Li, S. Li, and Z. Yu, “Hyperspectral image spectral-spatial-range Gabor filtering,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 58, no. 7, pp. 4818-4836, July 2020.

[8] “SciPy.org,” *SciPy.org - SciPy.org.* [Online]. Available: https://www.scipy.org/

[9] A NumPy-compatible array library accelerated by CUDA. *CuPy*. [Online]. Available: https://docs.cupy.dev/en/stable/

Conner McInnes was born in the United States in 1999. He received a B.S.E. in computer engineering from Oakland University, Rochester Hills, United States in 2021. His research interests include computer vision, robotics, signal processing, and machine learning.
He has worked for R.L Deppmann as a Research Intern during 2018 evaluating possible areas of expansion for products. In 2019 to 2020 he worked for Oakland University’s outreach program as an Outreach Support Member encouraging and teaching STEM topics to K-12 students. From 2020 to 2021 he worked at Oakland University for the GPU Computing Research Laboratory as a Research Student studying GPU acceleration of Gabor compression algorithms.
Mr. McInnes has received an award for best presentation at international conference VSIP 2020 for his research performed on Gabor Transforms.

Shadi Alawneh received the BEng degree in computer engineering from the Jordan University of Science and Technology, Irbid, Jordan in 2008, the MEng and PhD degrees in computer engineering from the Memorial University of Newfoundland, St. John’s, NL, Canada, in 2010 and 2014, respectively. His research interests include parallel and distributed computing, general purpose GPU computing, parallel processing architecture and applications, autonomous driving, numerical simulation and modeling, software design and optimization.

Then, he joined the Hardware Acceleration Lab at IBM Canada as a staff software developer from May 2014 through August 2014. After that, he joined C-CORE as a research engineer from 2014 until 2016 and became adjunct professor in the Department of Electrical and Computer Engineering at Memorial University of Newfoundland in 2016. Dr. Alawneh is currently an assistant professor in the department of Electrical and Computer Engineering at Oakland University.

Dr. Alawneh has authored or co-authored scientific publications (including international peer-reviewed journals and conferences). He is a senior member of the IEEE Computer Society.