A Comparative Study of the Speedup of Edge Detection in Python Numba

Carlos Alberto Lopez[1] and Jose Fernandez[1]

[1] *Columbia University in the City of New York, New York, United States of America*

May 2, 2024

Abstract

Edge detection is paramount in image processing for robotic applications since it enables feature extraction, pattern recognition, and image morphology. This project is a comparative study of edge detection speedup using Python-based implementations of edge detection that work in tandem with the Numba library designed for GPU manipulation. In essence, the precepts are images that were used as a training set for image stitching, but we only desire to do parallelization of edge detection as an initial step for feature and pattern extraction. For all intents and purposes, this study will perform Canny edge detection, Laplacian edge detection, Prewitt edge detection, Scharr edge detection, and Sobel edge detection using the Python Language and the Python Numba library. Ultimately, Numba implementations had a substantial speedup in comparison to pure Python and more simple CUDA implementations within Python, especially because images are larges complex vectors. The results are satisfactory and show that the most granular algorithms can be successfully parallelized with Numba such that more complex procedures such as image stitching can be performed in parallel after the most granular procedures are parallelized.

1 Introduction

Edge detection has been one of the most popular research areas since the early days of computer vision Vincent and Folorunso, 2009, serving as a fundamental a fundamental procedure to extract features and image morphology. More precisely, an **edge** is a point in an image with coordinates [i,j] at the location of a significant local intensity change in the image. An **edge detector** is an algorithm that produces a set of edges from an image. The relevance of this work is to build up on the implementation of edge detection algorithms that use first and second derivatives in a parallelized framework known as the Python Numba library. Image processing is data intensive endeavor that requires large amounts of computational power since images have a myriad of pixels or data points. Therefore, to enhance computer vision edge detectors and the accessibility to code-bases that work with GPUs, this study re-implements the Canny, Laplacian, Prewitt, Scharr, and Sobel edge detection algorithms within the Numba Python library that manipulates an NVIDIA GTX 4060 GPU. Overall, the Numba and Simple CUDA-Numba implementations surpass by far the pure Python implementation in time metrics. Consequently, this work will hopefully establish the basis for new developments in parallel computing for computing vision under the Python framework, which currently cannot be completely parallel for Python computer vision libraries sch as CV2.

2 Related Work

At GTC 2017, Anaconda et al., 2012-2020,(the main sponsor of Numba development) in collaboration with H2O, MapD, BlazingDB, Graphistry, and Gunrock announced the creation of the GPU Open Analytics Initiative (GOAI). Thus far, Numba has taken huge strides toward enhancing GPU data exchange between applications and libraries as cmputer vision workloads increasingly require the combination of multiple tools. GPU computing has become ubiquitous, so GPU manipulation as a hidden implementation detail. Additionally, Lam, Pitrou, and Seibert, 2015, have noted the importance of dynamic, interpreted languages such as Python for domain experts in computer vision who wish to scale to larger data sets. Hence, the ubiquity of Numba or Pyhton frameworks that enhance GPU ma-

nipulation should be further implemented in computer vision.

3 Background

Numba

As Oliphant, 2007, noted, Numba is a function-at-a-time Just-in-Time (JIT) compiler for CPython. Numba is implemented as a library that can be loaded by programs running in the CPython interpreter; It does not replace the CPython interpreter itself. Numba targets a Python subset that heavily uses ndarrays and numeric scalars in loops so that users no longer need to rewrite their Python code in low-level languages for better performance. Accordingly, Numba supports GPGPU backends by directly exposing the parallel execution model of the hardware similarly to CUDA-C and OpenCL. Consequently, any Python function can be decorated as a GPU kernel or a device function. This design enables simplicity in the implementation since the GPGPU backends can share most of the code generation logic with the CPU backend. However, the Numba implementation alters the language semantic to fit the parallel execution model of the GPGPUs, which may confuse regular Python programmers with no GPGPU experience. Numba supports CUDA GPU usage through both automatic and manual memory management between the GPU device memory and the CPU host memory. When a ndarray is used as an argument, the memory is automatically transferred to the device and the copied back when the kernel is completed. The transfer is clearly and transparent but may perform unnecessary transfer since Numba cannot determine whether a ndarray is never modified. It is recommended to explicitly request memory transfer using the to device and copy to host functions.

Edge Detectors

Now, edge filters are implemented as 3x3 matrices in this study an rely on the mathematical principles of first and second derivatives to calculates changes in light and direction in images. The following definitions are important to comprehend why parallelizing is essential in the filtering process:

- **Edge**: point in an image with coordinates [i,j] at the location of a significant local intensity change in the image
- **Edge fragment**: the i and j coordinates of an edge and the edge orientation e, which may be the gradient angle
- **Edge detector**: an algorithm that produces a set of edges applying the same filter to a subset of a myriad of points.
- Contour: a list of edges or the mathematical curve that models a list of edges.

Any Numba code implementation of a filter must follow most of the processes below:

- **Filtering**: the process that reduces noise for image filtering, which may in turn reduce edge strength (a trade-off)
- **Localization**: finding the i and j coordinates of an edge with sub-pixel resolution
- **Enhancement**: the process that emphasizes pixels in which here is a significant change in local intensity.
- **Detection**: thresholding for points with strong edge content which is required because an image may have non-zero gradients for many edges.

4 Algorithms and Implementation in ”numba” and simple ”numba.cuda”

Canny Edge Detector

Succinctly, the Canny edge detector, developed by John F Canny, is an operator that uses a multistage algorithm involving: noise reduction, gradient calculation, non-maximum suppression, double threshold, and edge tracking by hysteresis (Milan S and B, 2010)(R., 2009)(Raman and Aggarwal, 2009).

Simple "numba" for the Canny Filter

```
@jit(uint8[:, :](float32[:, :]), nopython=
    True, parallel=True)
def canny_operator(input_image):
    # Apply Gaussian blur to the image
    blurred_image = gaussian_blur(input_image
    )
    # Initialize the gradient images
    filter_x = np.zeros_like(blurred_image)
    filter_y = np.zeros_like(blurred_image)
    # canny operator kernels
    Gx = np.array([[-1, 0, 1], [-2, 0, 2],
    [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1], [0, 0, 0],
    [1, 2, 1]])
    # Initialize the gradient images
    filter_x = np.zeros_like(input_image)
    filter_y = np.zeros_like(input_image)
    # Apply the filter
    # Parallelize the outer loop
    for i in prange(1, input_image.shape[0] -
     1):
        for j in prange(1, input_image.shape
    [1] - 1):
            filter_x[i, j] = np.sum(Gx *
    input_image[i-1:i+2, j-1:j+2])
            filter_y[i, j] = np.sum(Gy *
    input_image[i-1:i+2, j-1:j+2])
    magnitude = np.hypot(filter_x, filter_y)
    magnitude = np.clip(magnitude, 0, 255) #
    Clip to the range 0-255
    return magnitude.astype(np.uint8) #
    Convert to uint8 if necessary
def canny_operator_helper():
        nonlocal filtered_image
        filtered_image = canny_operator(
    input_image)
```

Simple "numba.cuda" for the Canny Filter

```
# canny operator kernels
    CANNY_X = np.array([[-1, 0, 1], [-2, 0,
    2], [-1, 0, 1]])
```

```
    CANNY_Y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
def Canny_filter_helper():
        nonlocal filtered_image, blurred_image
        gfilter[blockspergrid, threadsperblock](input_image, filtered_image, Gausskernel)
        cfilter[blockspergrid, threadsperblock](input_image, filtered_image, CANNY_X, CANNY_Y, Gausskernel, blurred_image)
        cuda.synchronize()
```

Laplacian Edge Detector

Succintly, the Laplacian filter is a linear high pass filter. This detection operator used to compute the second derivative of an image (concavity or convexity) (Milan S and B, 2010)(R., 2009)(Raman and Aggarwal, 2009). **Simple "numba" for the Laplacian Filter**

```
@jit(uint8[:, :](float32[:, :]), nopython=True, parallel=True )
def laplace_operator(input_image):
    kernel = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
    blurred_image = gaussian_blur(input_image)
    output_image = np.zeros_like(blurred_image)
    for i in prange(1, input_image.shape[0] - 1):
        for j in prange(1, input_image.shape[1] - 1):
            output_image[i, j] = np.sum(kernel * input_image[i-1:i+2, j-1:j+2])
    magnitude = np.abs(output_image)
    magnitude = np.clip(magnitude, 0, 255)  # Clip to the range 0-255
    return magnitude.astype(np.uint8)
def laplace_operator_helper():
        nonlocal filtered_image
        filtered_image = laplace_operator(input_image)
```

Simple "numba.cuda" for the Laplace Filter

```
#Laplace Kernel
    Laplacekernel = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]], dtype=np.int32)
    def Laplace_filter_helper():
        nonlocal filtered_image, blurred_image
        gfilter[blockspergrid, threadsperblock](input_image, filtered_image, Gausskernel)
        opfilter[blockspergrid, threadsperblock](input_image, filtered_image, Laplacekernel, Gausskernel, blurred_image)
        cuda.synchronize()
```

Prewitt Edge Detector

Succinctly, the Prewitt filter is a non-linear highpass edge detection operator that computes an approximation of the image intensity function (eg increases from light to dark or vice versa) (Milan S and B, 2010)(R., 2009)(Raman and Aggarwal, 2009). **Simple "numba" for Prewitt Filter**

```
@jit(uint8[:, :](float32[:, :]), nopython=True, parallel=True )
def prewitt_operator(input_image):
    Gx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    filter_x = np.zeros_like(input_image)
    filter_y = np.zeros_like(input_image)
    for i in prange(1, input_image.shape[0] - 1):
        for j in prange(1, input_image.shape[1] - 1):
            filter_x[i, j] = np.sum(Gx * input_image[i-1:i+2, j-1:j+2])
            filter_y[i, j] = np.sum(Gy * input_image[i-1:i+2, j-1:j+2])
    magnitude = np.hypot(filter_x, filter_y)
    magnitude = np.clip(magnitude, 0, 255)  # Clip to the range 0-255
    return magnitude.astype(np.uint8)  # Convert to uint8 if necessary
def prewitt_operator_helper():
        nonlocal filtered_image
        filtered_image = prewitt_operator(input_image)
```

Simple "numba.cuda" for Prewitt Filter

```
# Define Prewitt kernels
    PREWITT_X = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.int32)
    PREWITT_Y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.int32)
    def prewitt_filter_helper():
        nonlocal filtered_image
        dfilter[blockspergrid, threadsperblock](input_image, filtered_image, PREWITT_X, PREWITT_Y)
        cuda.synchronize()
```

Scharr Edge Detector

Succinctly, the Scharr filter is a nonlinear edge detection operator used identify the edges using the first derivative; It has better isotropy than most (uniformity in all directions) (Milan S and B, 2010)(R., 2009)(Raman and Aggarwal, 2009)

Simple "numba" for the Scharr Filter

```

@jit(uint8[:, :](float32[:, :]), nopython=True, parallel=True )
def scharr_operator(input_image):
    """
    Apply a Sobel filter to the image.
    """
    Gx = np.array([[-3, 0, 3], [-10, 0, 10], [-3, 0, 3]])
    Gy = np.array([[3, 10, 3], [0, 0, 0], [-3, -10, -3]])

    # Initialize the gradient images
    filter_x = np.zeros_like(input_image)
    filter_y = np.zeros_like(input_image)

    # Apply the filter
    # Parallelize the outer loop
    for i in prange(1, input_image.shape[0] - 1):
        for j in prange(1, input_image.shape[1] - 1):
            filter_x[i, j] = np.sum(Gx * input_image[i-1:i+2, j-1:j+2])
```

```
            filter_y[i, j] = np.sum(Gy *
    input_image[i-1:i+2, j-1:j+2])

    magnitude = np.hypot(filter_x, filter_y)
    magnitude = np.clip(magnitude, 0, 255)  #
     Clip to the range 0-255
    return magnitude.astype(np.uint8)  #
    Convert to uint8 if necessary
def scharr_operator_helper():
        nonlocal filtered_image
        filtered_image = scharr_operator(
    input_image)
```

Simple "numba.cuda" for the Scharr Filter

```
 # Define Scharr kernels
    SCHARR_X = np.array([[-3, 0, 3], [-10, 0,
     10], [-3, 0, 3]], dtype=np.int32)
    SCHARR_Y = np.array([[3, 10, 3], [0, 0,
    0], [-3, -10, -3]], dtype=np.int32)
def scharr_filter_helper():
        nonlocal filtered_image
        dfilter[blockspergrid,
    threadsperblock](input_image,
    filtered_image, SCHARR_X, SCHARR_Y)
        cuda.synchronize()
```

Sobel Edge Detector

Succinctly, the Sobel filter is a nonlinear-highpass edge detection operator that calculates 1st derivative and emphasizes pixels that are close to the center of the mask (Milan S and B, 2010)(R., 2009)(Raman and Aggarwal, 2009).

Simple "numba" for the Sobel Filter

```

@jit(uint8[:, :](float32[:, :]), nopython=
    True, parallel=True )
def sobel_filter(input_image):
    """
    Apply a Sobel filter to the image.
    """
    Gx = np.array([[-1, 0, 1], [-2, 0, 2],
    [-1, 0, 1]])
    Gy = np.array([[-1, -2, -1], [0, 0, 0],
    [1, 2, 1]])

    # Initialize the gradient images
    filter_x = np.zeros_like(input_image)
    filter_y = np.zeros_like(input_image)

    # Apply the filter
    # Parallelize the outer loop
    for i in prange(1, input_image.shape[0] -
     1):
        for j in prange(1, input_image.shape
    [1] - 1):
            filter_x[i, j] = np.sum(Gx *
    input_image[i-1:i+2, j-1:j+2])
            filter_y[i, j] = np.sum(Gy *
    input_image[i-1:i+2, j-1:j+2])

    magnitude = np.hypot(filter_x, filter_y)
    magnitude = np.clip(magnitude, 0, 255)  #
     Clip to the range 0-255
    return magnitude.astype(np.uint8)  #
    Convert to uint8 if necessary
    def sobel_filter_helper():
        nonlocal filtered_image
        filtered_image = sobel_filter(
    input_image)
```

Simple "numba.cuda" for Sobel Filter

```
SOBEL_X = np.array([[-1, 0, 1], [-2, 0, 2],
    [-1, 0, 1]], dtype=np.int32)
SOBEL_Y = np.array([[-1, -2, -1], [0, 0, 0],
    [1, 2, 1]], dtype=np.int32)
def sobel_filter_helper():
        nonlocal filtered_image
        dfilter[blockspergrid,
    threadsperblock](input_image,
    filtered_image, SOBEL_X, SOBEL_Y)
        cuda.synchronize()
```

Commentary on simple ”numba” implementation

Overall, the simple Numba implementation requires simple python code without library dependencies other than Numpy functions. That is, any Opencv code will result in an error. The Numba library simply uses as an indicator the "@jit" header to parallelize all operations in the function if possible. As shown, we set the parallel parameters to True. As demonstrated in the official Numba documentation, the user is required to make sure that the loop does not have cross iteration dependencies except for supported reductions.

Commentary on simple ”numba.cuda” Implementation

This CUDA implementation simply loads the kernels and the respective images to the device. We set the i and j index to a CUDA grid of size 2 in the intermediate helper functions that can be revised in the codebase (Lopez and Fernandez, 2024). Additionally, we use the @cuda.jit header which implies a less automatic parallelization; A tangible example of the exact CUDA manipulations are shown below: **Simple "numba.cuda" for Sobel Filter**

```
# Define block size and grid size this is
    just and out of place example
    threadsperblock = (16, 16)
    blockspergrid_x = int(np.ceil(input_image
    .shape[0] / threadsperblock[0]))
    blockspergrid_y = int(np.ceil(input_image
    .shape[1] / threadsperblock[1]))
    blockspergrid = (blockspergrid_x,
    blockspergrid_y)
#... ...
@cuda.jit(void(float32[:,:], uint8[:, :],
    int32[:, :], int32[:, :]))
def dfilter(input_image, output_image,
    kernel_x, kernel_y):
    i, j = cuda.grid(2)
    height, width = input_image.shape
    kernel_height, kernel_width = kernel_x.
    shape
    # Calculate the margins to avoid boundary
     issues
    margin_y, margin_x = kernel_height // 2,
    kernel_width // 2
    if (margin_y <= i < height - margin_y)
    and (margin_x <= j < width - margin_x):
        Gx, Gy = 0, 0
        # Apply operator dynamically based on
     kernel size
        for u in range(-margin_y, margin_y +
    1):
```

```
            for v in range(-margin_x, margin_x + 1):
                img_val = input_image[i + u, j + v]
                Gx += kernel_x[u + margin_y, v + margin_x] * img_val
                Gy += kernel_y[u + margin_y, v + margin_x] * img_val
        # The magnitude of the gradient
        output_image[i, j] = min(255, max(0, math.sqrt(Gx**2 + Gy**2)))
```

Above we just present the "dfilter" function we use as an intermediate step in the implementation of the Prewitt, Scharr, and Sobel filters. There are more intermediate simple "numba.cuda" functions. As in the "PyCUDA" case, the function call needs calling the kernel in a main loop and storing the snapshot arrays at some given moments and synchronization is overall simplified with a call to "cuda.synchronize()."

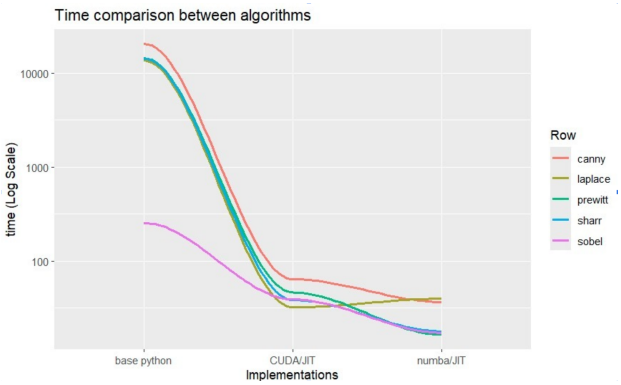

Figure 2: *Additionally, for this small subset of 20 runs, there is an evident scale reduction. This log-scale chart shows that "numba" and "numba.cuda" perform better by two to three magnitudes of scale in base 10.*

5 Results and Discussion

5.1 Box Charts of Filters

The results will not focus on a single case but rather on subsets of size 100, 1000, and 10000 due to time limitations. The mean and median case for each component used will only focus on the usage of simple "numba" and simple "numba.cuda" since it will be evident that for complex computations on large data, "numba" and "numba.cuda" will perform better by large magnitudes.

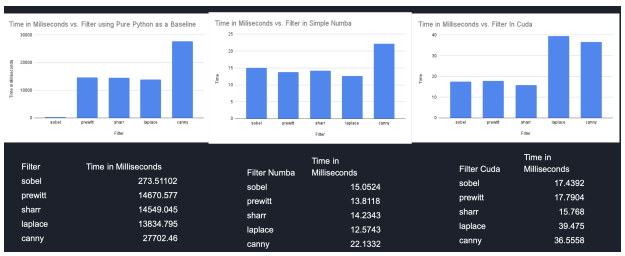

Figure 1: *This tangible example of the magnitude in difference of each run. The leftmost bar chart has values that go up to 14.8 seconds, which correspond to pure Python. By contrast "numba" and "numba.cuda" implementations did not go beyond 40 milliseconds for this particular subset.*

Further, the joint box plots for each filter are shown below an exhibit a great similarity between "numba" and "numba.cuda" for filters that require more computational steps: Prewitt, Scharr, and Sobel. By contrast, the more simple the computation is, the better numba performs overall.

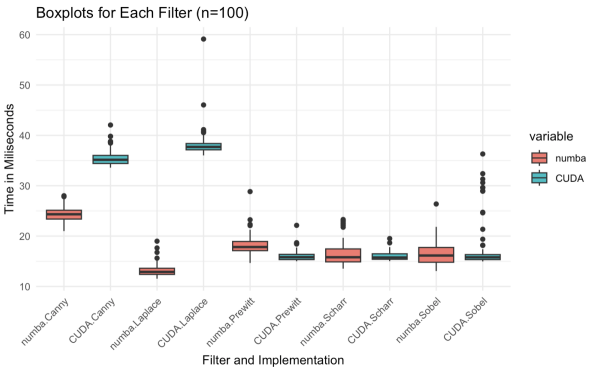

Figure 3: *Each filter exhibits a similar median in "numba" and "nuba.cuda" for Prewitt, Scharr, and Sobel filters since they have more complex procedures that require manipulating distinct arrays. By contrast, the simplicity of the Canny and Laplacian filters enables "numba" to perform better.*

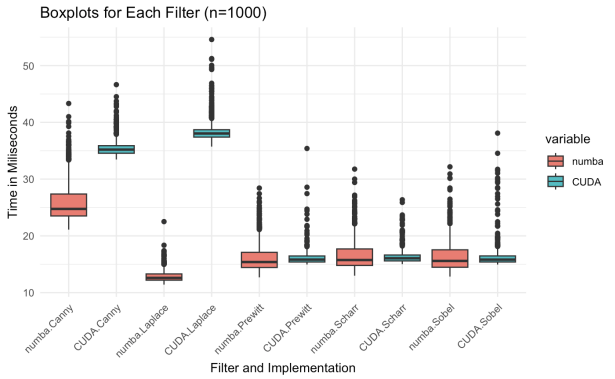

Figure 4: *The median similarity for Prewitt, Scharr, and Sobel filters remains. The discrepancy for the Laplacian an Canny filter remains. Additionally, the IQR of each box plot decreased significantly*

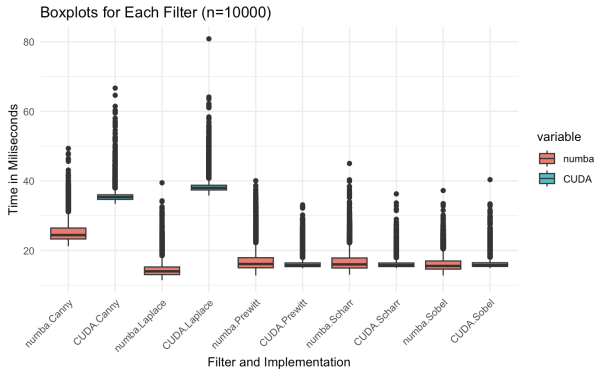

Figure 5: *The median similarity for Prewitt, Scharr, and Sobel filters remains. The discrepancy for the Laplacian an Canny filter remains. The IQR decreased further and the plotting of outliers also increased, which may require further investigation.*

P-value and T-statistics

Numba_mean <dbl>	CUDA_mean <dbl>	p.value <dbl>	Filter <chr>
16.33507	17.28710	5.332501e-02	Sobel
18.19367	16.05355	9.442998e-19	Prewitt
16.45828	15.99237	5.741981e-02	Scharr
13.15231	38.09033	8.457747e-127	Laplace
24.32520	35.48995	7.776411e-120	Canny

Figure 6: *For the case of 100 iterations, the means exhibit the same "numba" and "numba.cuda" differences shown for the medians in the box plots. The significance for each is well established by the fact all p-values are below the 0.01 threshold.*

Numba_mean <dbl>	CUDA_mean <dbl>	p.value <dbl>	Filter <chr>
16.36212	16.25700	0.319908216	Sobel
16.07326	16.07453	0.987923812	Prewitt
16.55282	16.26319	0.001072472	Scharr
12.86485	38.33937	0.000000000	Laplace
25.89532	35.39702	0.000000000	Canny

Figure 7: *For the case of 1000 iterations, the means exhibit the same "numba" and "numba.cuda" differences shown for the medians in the box plots. The significance of all p-values are below the 0.01 threshold except for Sobel and Prewitt filters which exhibit a decrease in the mean in comparison to the n=100 case.*

Numba_mean <dbl>	CUDA_mean <dbl>	p.value <dbl>	Filter <chr>
16.20619	16.13385	8.779736e-03	Sobel
17.02477	16.05694	1.137751e-179	Prewitt
16.88574	16.03489	8.656849e-155	Scharr
14.45314	38.48547	0.000000e+00	Laplace
25.29145	35.65721	0.000000e+00	Canny

Figure 8: *For the case of 10000 iterations, the means exhibit the same "numba" and "numba.cuda" differences shown for the medians. All p-values remain well below the 0.01 threshold. Additionally, the "numba" value for the Prewitt filter increased to 17 milliseconds and the "numba.cuda" implementation mean decreased by less than 0.02 milliseconds.*

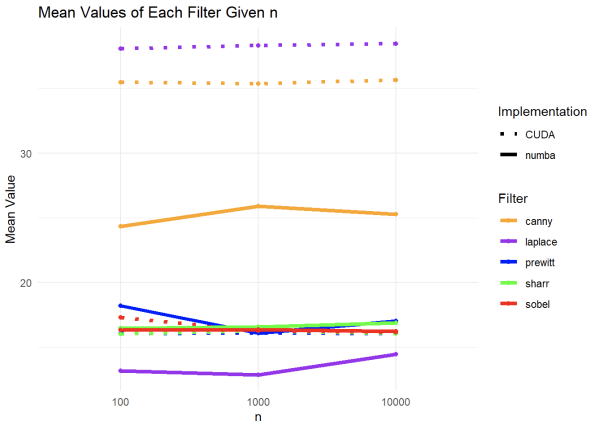

Figure 9: *Overall, means decrease more as the number of iterations moves from 100 to 1000. However, as the number of iterations moves from 1000 to 10000, the means increase slightly but the differences between the numba and cuda samples for all filters retains statistical significance at n=10000.*

Analysis of Variance

After running the algorithm multiple times, the Numba implementation seemed to outperform the CUDA implementation, so we decided to run a two-way ANOVA test, and a Barlett's test for homogeneity of variance. These procedures give more information on which implementation of the algorithm should be chosen when a project will use these filters at scale. Initially, there

would be no reason to assume that the variance of the algorithm would change as the number of iterations (n) increases. Additionally, if one implementation is better than another one we would expect this to remain the same as n increases as well. However, the tests show that this is not the case for some of the filters. In fact, despite the CUDA implementation taking longer to terminate in some cases, it has a more homogeneous variance overall when compared to the Numba implementation, which suggests that if a project demands more stability, the filter should be used at a large scale, a user might want to consider how the variability might affect the performance of their overall algorithm. Accordingly, we consider that taking the variance into consideration allows for a more granular understanding of when exactly it makes a difference to choose one algorithm over the other. For that reason, using the mean as the only point of comparison might lead to inaccurate decisions, despite rightfully claiming that one algorithm is on average faster than the other one.

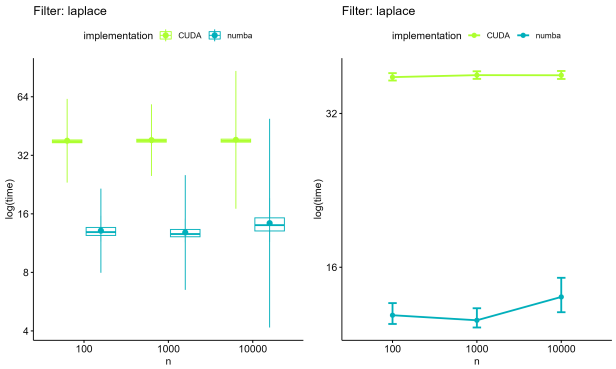

Figure 11: *This plot exemplifies how the Laplace filter behaves as n increases. For this algorithm the rate at which the variance of the time to halt increases in the Numba implementation is substantially different from the CUDA implementation. The results of the ANOVA test and the Bartlett test for homogeneity of variance are also highly significant being $Pr(> F) = 2E - 16$, and $p - value = 0.008808$ respectively. Therefore we are able to reject the null hypothesis that the variance remains constant across implementation. Nonetheless, the ANOVA test also remains significant meaning that the Numba implementation outperforms the CUDA implementation. Even though the test shows that the rate at which the variance increases is different, it is orders of magnitude less significant than it is for other filters. Therefore, it is safe to choose the Numba implementation over the CUDA irrespective of the scale at which the algorithm will be used.*

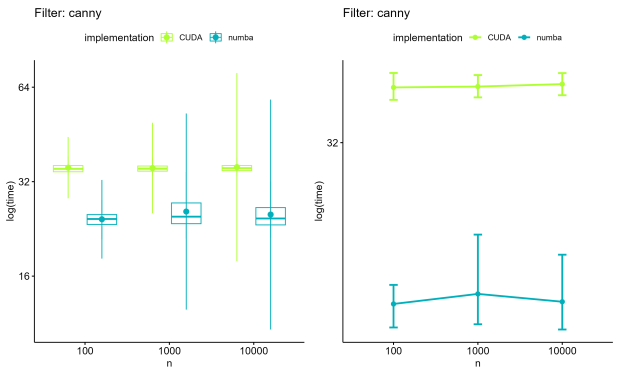

Figure 10: *The plot for the ANOVA test shows how the canny filter behaves as n increases. Overall, the difference in time is far superior in the Numba implementation for this algorithm vs the CUDA implementation. Therefore, this case conforms to the assumptions that one might initially make. However, the box plot on the left can help visualize how the variance increased more for the Numba implementation compared to the CUDA implementation. The results for this test are not as significant as with other filters. Therefore, choosing the Numba implementation remains the better choice.*

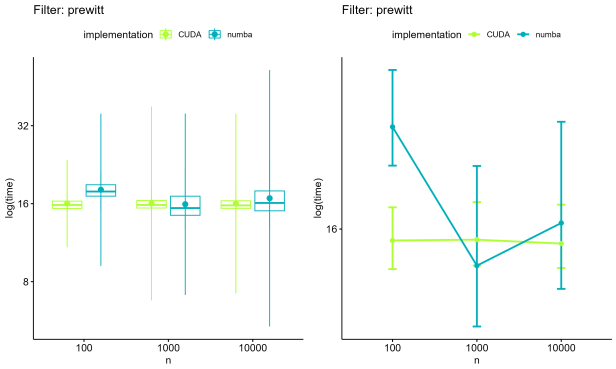

Figure 12: *This plot shows how the Prewitt filter behaves as n increases. From the plot, we can see that the CUDA outperforms Numba at some points. To choose CUDA over Numba a Bartlett test that is highly significant is necessary. Now, with an ANOVA $Pr(> F) = 2E - 16$ the variance between both implementations is clearly different, and a Bartlett test with a $p - value < 2E - 16$ indicates that although there is still a difference between algorithms the variance increases at a higher-rate overtime for the Numba. Although the ANOVA and Bartlett are significant there might be instances in which it is still better to choose Numba over CUDA, but CUDA is a much stronger choice when using the Prewitt filter.*

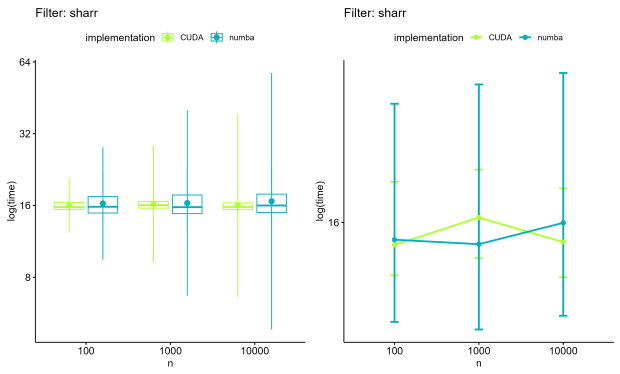

Figure 13: *The plot for the Scharr filter shows how the time difference between implementations is minimal. However, the variance for the Numba implementation increases so much that we can definitely conclude that CUDA is evidently a better implementation than Numba. The ANOVA test fails to reject the null at a* $Pr(> F) = 0.205$ *and the Bartlett Test exhibits a* $p - value < 2E - 16$ *such that we reject the null. In this case, the variance of the CUDA implementation remains virtually constant as n increases, whereas the Numba implementation's variance is greater and increases over time. Since the time complexity is not significantly different between both, then we can confidently choose the CUDA implementation for the Scharr filter. This is one case in which despite the I/O overhead that we incur with the CUDA implementation, the CUDA implementation outperforms in the long run.*

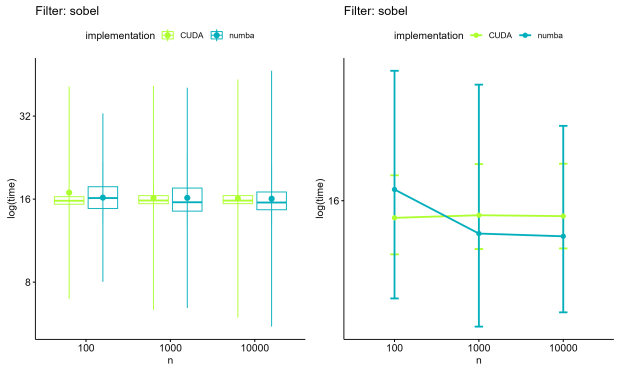

Figure 14: *The plot for the Sobel filter shows on average both algorithms perform similarly. However, due to the homogeneity in the variance of the CUDA filter, it is also sensible to choose that implementation for large-scale applications. The ANOVA test has* $Pr(> F) = 0.01437$ *which is at the edge of statistical significance, so we can infer there is little difference between the filters. However, the Bartlett test remains highly significant with* $p - value < 2.2E - 16$ *meaning that although on average for small n, both filters perform similarly, for large n there is an advantage to using the CUDA implementation.*

6 Conclusion and Future Work

Numba has a huge potential to ease GPU computing for edge detection as an initial step for feature extraction and pattern recognition methods. As shown, the speed increase is substantial compared to baseline metrics. Overall, simple Numba performs better with statistical significance. For some filters, Numba is uncontested. However, incorporating the variance into the equation allows for nuisance that would be lost if a user were to only consider the average times. The CUDA implementation has consistently more homogeneous variance across all algorithms, and this homogeneity allows it to overtake the Numba at large n. Incorporating the variance into decision-making exposes how some statistical tests that rely only on mean values have low power and are susceptible to type 2 errors, commonly known as false negatives. This Numba implementation of the filters offers deferred loop specialization, array expression rewrite, and multiple back-end support even when using an NVIDIA GTX 4060 GPU. Nonetheless, it is the deft API usage that makes it possible. It allows users to parallelize their algorithms using Python syntax that is vastly similar to the conventional one. However, the Numba implementation does not scale like the CUDA implementation does, as can be evidenced by the heterogeneity in the variance of time. The CUDA implementation we used also doesn't manage memory explicitly. However, Numba does allow users to manage memory allocation and interface it with CUDA, but with syntax that is far different from regular Python syntax, which allows Python users to come closer to C++ performance.

Bibliography

Anaconda, Inc. et al. (2012-2020). "Numba for CUDA GPUs". In: URL: http://numba.pydata.org/numba-doc/latest/cuda/index.html.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert (2015). "Numba: a LLVM-based Python JIT compiler". In: LLVM '15. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

Lopez, Carlos and Jose Fernandez (2024). "A Comparative Study of the Speedup of Edge Detection in Python Numba". In: URL: https://github.com/carloslomon/parallelfinal.

Milan S, Vaclav H and Roger B (2010). "Algorithms for Image processing and computer vision 2nd Edition". In.

Oliphant, Travis E. (2007). "Python for Scientific Computing". In: *Computing in Science and Engg.* 9.3. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.58. URL: https://doi.org/10.1109/MCSE.2007.58.

R., Parker J. (2009). "Image processing, analysis, and machine vision, second Edition". In.

Raman, Maini and Himanshu Aggarwal (Mar. 2009). "Study and Comparison of Various Image Edge Detection Techniques". In: *International Journal of Image Processing* 3.

Vincent, Olufunke and Olusegun Folorunso (Jan. 2009). "A Descriptive Algorithm for Sobel Image Edge Detection". In: DOI: 10.28945/3351.