*OpenCV Performance Evaluation*

Brian Lam, Don Huynh, Jorge Luis Suarez, Benjamin Hong

{lam, donhuynh, bjhong, jorgeluiss}@cpp.edu

ECE-department, College of Engineering

California State Polytechnic University, Pomona

*Abstract*—This paper will investigate performance metrics and analyze benchmark results from tests on the OpenCV library using different programming languages on the Raspberry Pi 4, Model B (8gb). Using scripts to collect the performance results of several programming languages for each function, the best language to use with OpenCV can be determined.

Keywords—Performance metrics, benchmark, OpenCV, Raspberry Pi

# Introduction (*Heading 1*)

This research focuses on benchmarking the execution speed, CPU and memory usage, and power consumption of five specific OpenCV image processing functions using four different languages on the Raspberry Pi 4, Model B. The five functions are image cropping, image resizing, image sharpening, blue to gray color conversion, and histograms. Each of these OpenCV functions included four different programs to perform the image processing – A C++ program, a Python program, a Golang program, and a Rust program. C++’s low-level optimizations, Python’s high-level scripting, Golang’s concurrency support, and Rust’s memory safety provides us with four solid language platforms for each of the test functions on our testbed, the RPi. The results of these tests will provide indication as to which language is best for specific functions as well as a conclusion as to which language is best for OpenCV overall.

# Function set

The chosen functions for evaluation were specifically tailored to minimize external influences, aiming to gauge the raw performance of library implementations across various programming languages. Acknowledging the variability inherent in computer vision tasks, the selection of fundamental functions—image cropping, resizing, sharpening, color conversion, and histograms—strives to provide a focused assessment.

## Blue to Gray Color Conversion

The blue to gray color conversion function within OpenCV was the first function we were set to implement and serves as an example of a popular use case for the library. This operation involves transforming an input image, assumed to be in the Blue-Green-Red (BGR) color space, into a grayscale representation.

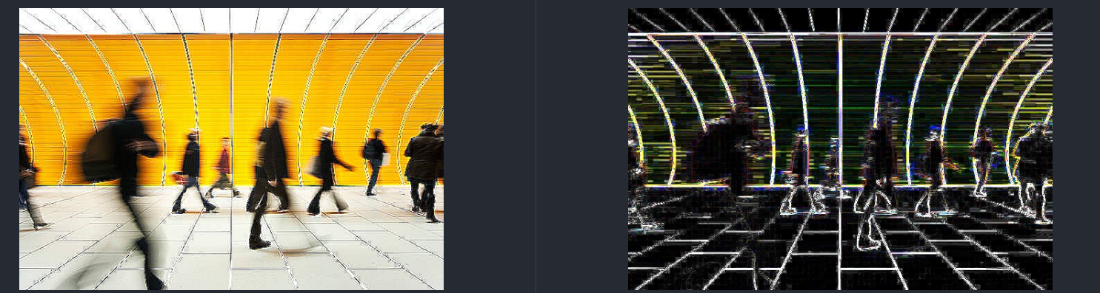
In the BGR color model, images are typically represented as a combination of three color channels—blue, green, and red—forming the fundamental basis for color representation in many digital images. The blue to gray conversion primarily entails extracting the blue channel from the input BGR image and transforming it into a grayscale channel, discarding color information while retaining luminance data.

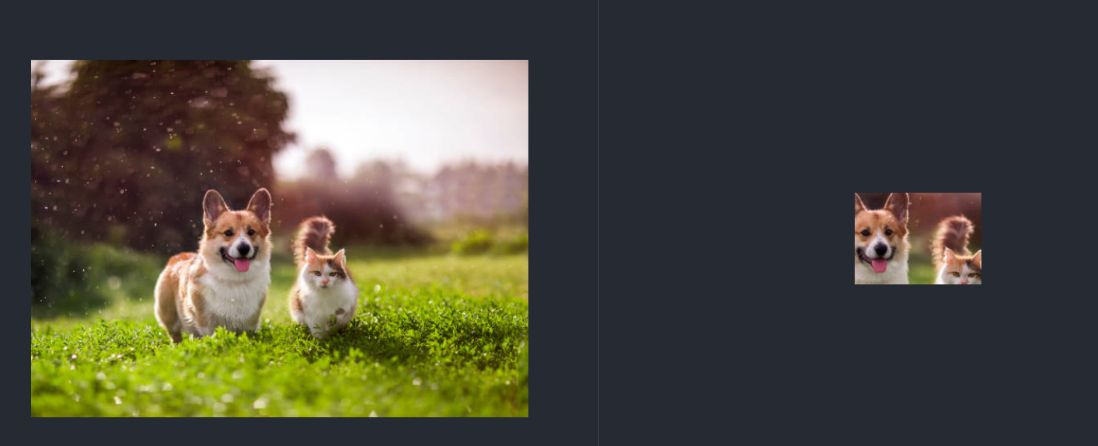
Grayscale representations often serve as the preferred choice for extracting descriptors rather than operating directly on color images, primarily due to the simplified algorithms and reduced computational demands. The rationale behind this practice lies in the notion that color, while rich in information, might offer limited advantages in numerous applications. In fact, the inclusion of color data could potentially necessitate increased training data for optimal performance, introducing unnecessary complexities into the computational process.

## IMG Sharp/Crop/Resize

In the OpenCV library, image processing is done through a variety of functions, including image sharpening, image cropping, and image resizing. These functions are one of the most standard image processing techniques, which is why they are chosen to be included in the tests performed on the RPi. The image sharpening function enhances the edges and details of an image, while the image cropping and image resizing focuses on a selected region of the image and discarding the rest. Between image cropping and image resizing, the former selects a specific region of interest and discards the rest, while the latter adjusts the dimensions of the image and either scales it up or down.

These three functions are similar to each other in terms of the structure of their code, thus they are perfect candidates for the tests performed in this project.





## Histogram

Histogram function analyzes an image as a graph or plot by relating the frequency of pixels in the specific color channels like in grayscale or red, green, blue into a graphical representation. It represents the distribution of pixel intensities in an image. The function’s arguments is “cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])”. The function’s parameters are the image source, channel index, optional masking region, histSize refers to a range in which the pixel intensity values are binned (BIN count), the range of pixel values for each channel. The generated histogram allows for valuable insights in composition and characteristics of images,

# Language Performance

The set of coding languages used in this project include C++, Python, Golang, and Rust. All four of these are versatile programming languages, each with their own distinct characteristics.

With C++, a powerful and high-performance language known for its efficiency, Python, a language known for its simplicity and readability, Golang, a language made by Google that is known for its simplicity, concurrency support, and scalability, and Rust, a language known for its low-level control and memory safety, this project has a well diverse set of languages to be able to draw the conclusion of which language is best compatible with OpenCV, based on performance results.

## C++

For installing OpenCV in C++, the installation process followed the official documentation’s OpenCV C++ quick start guide. The versions used were OpenCV 4.5.4 and g++ 11.4.0. Originally, OpenCV was developed in C and extended to C++, so most of the functions used for testing was available and documented like the getTickCount(), file operations, and the specific functions. When compiling C++ code, the code used lopencv\_core, lopencv\_highgui, lopencv\_imgcodecs, lopencv\_imgproc libraries from the OpenCV header. An example of compiling the resize function is “g++ resize.cpp -I/usr/include/opencv4 -lopencv\_core -lopencv\_highgui -lopencv\_imgcodecs -lopencv\_imgproc -o resize”.

## Python

For OpenCV on Python, the installation was straightforward with a “pip install” and was the easiest to run. Python was the most convenient option. For OpenCV, many users prefer Python because of its ease of use, rapid prototyping, and robust community support. The abundant resources in the documentation and community posts expedited the setup and testing process for using OpenCV in Python.

## Golang

Implementing OpenCV functionalities in Golang using the gocv package was facilitated by its straightforward installation process. Despite encountering significant processing delays during the object linking process on the Raspberry Pi, we proceeded with the evaluation, emphasizing the code's intrinsic execution efficiency once initiated.

The gocv package provided a relatively uncomplicated integration of OpenCV functionalities into Golang code. Extensive documentation significantly aided the implementation process, allowing us to get up and running quickly. Golang's interpreted nature proved advantageous, facilitating debugging through detailed runtime error messages in the terminal, particularly in identifying missing parameters by datatype.

An intriguing part of Golang is its robust concurrency support, offering potential scalability benefits in image processing tasks. Although not directly utilized in our evaluation, Golang's concurrency model could potentially enhance performance in scenarios requiring parallel processing of multiple images or intensive computational tasks. Leveraging Golang's goroutines and channels, developers can explore parallelism to scale output and optimize performance in image processing applications.

An interesting observation was the discernible startup delay for Golang scripts, which was observed in the Raspberry Pi environment. However, in our evaluation, we chose to prioritize the code's performance once initiated, consciously excluding external factors such as startup delays from our execution time measurements. This approach enabled us to focus solely on evaluating the code's computational efficiency and execution speed, providing insights into the raw performance of the implemented OpenCV functions in Golang.

## Rust

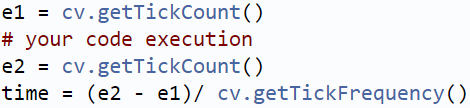
For the Rust implementations of the five OpenCV functions we used, the coding experience was simple and straight forward. Using OpenCV with Rust is as simple as using Cargo to add the package to the project [6], and with the C++ implementation already developed, coding the Rust implementation was nothing more than a translation of the C++ language, which was the case for the other two languages.

The aim was to create several implementations of the same function as identical as possible. For Rust, this proved to be impossible for some of the functions in the function set. This was because of the differences between the OpenCV library in each of the languages. In the case of Rust, some functions had completely different names as well as a different number of parameters/arguments [5]. On top of the differences between the OpenCV library of each language, Rust’s base language also has its own limitations. Due to the restrictions that Rust’s compiler enforces to maintain the memory safety the language is known for, certain things in the C++ code had to be manipulated and structured differently to maintain the same output [6].

Although the Rust implementation had its complications, it proved to be the fastest in execution time compared to the other three languages. The Rust implementation for the OpenCV functions were up to approximately 2 times faster at most. However, the results were similar when comparing with the C++ implementation. Overall, the Rust implementation of the functions were a success despite being met with constant compiling errors, which were eventually fixed.

# Performance Gathering and Analysis

Since the only metric of interest in this performance evaluation is execution time, there are many ways to measure it. The method we chose to do so is provided by OpenCV’s library. From the OpenCV documentation, using python’s version the way to calculate execution time is as follows:



*Figure: Execution Time calculation [4]*

This allows us to measure the execution time of only the code pertaining to image processing, as well as I/O operations with things like input images.

1. *Benchmarking Methodology*

To collect data for the evaluation of 20 code segments across different languages, a bash script was developed to automate the benchmarking process. This script performed multiple tasks, including the deletion of any existing output files and execution of each script, capturing their respective execution times.

Two testing methodologies were employed:

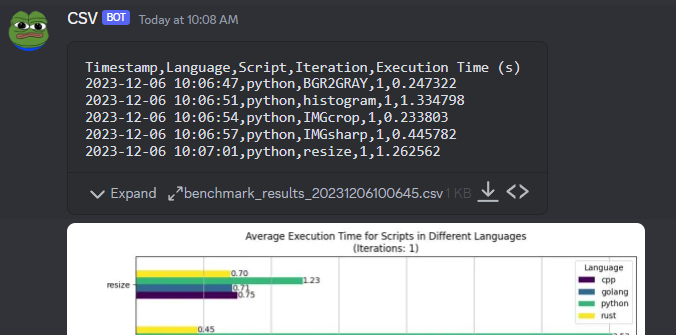
1. Single Run Method:

This method executed all 20 scripts once and recorded their individual execution times. The data was organized and saved into a CSV file for further analysis.

2. Iterative Method:

Offering greater control, the bash script allowed users to specify the number of iterations for each script. It executed the same script for the defined iterations and captured the execution times. These times were then averaged to derive more stable performance metrics, enhancing reliability in assessment.

Additionally, a Discord webhook integration streamlined access to test results. Upon completion of all tests, a Python script automatically generated a chart summarizing the collected data. Subsequently, both the CSV file and the chart image were transmitted via the Discord webhook, providing convenient access and visualization of the benchmarking outcomes.

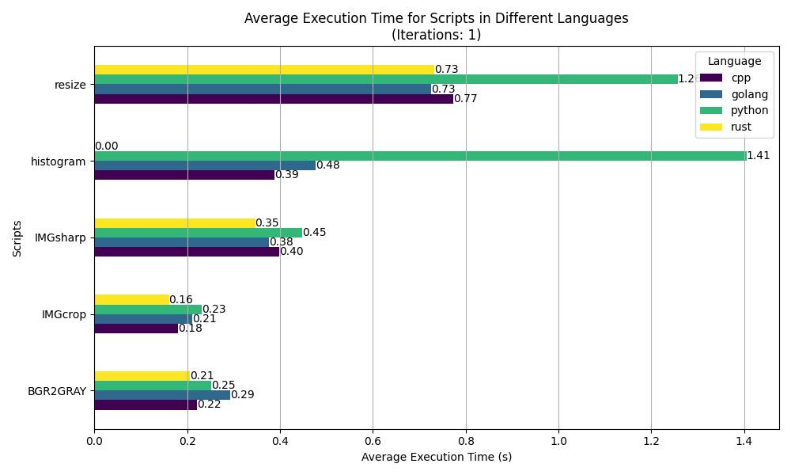


*Figure: Benchmark Output via Discord*

1. *Performance Analysis*

Using the bash script, we can start collecting and also representing data automatically.

1. Single Run Results

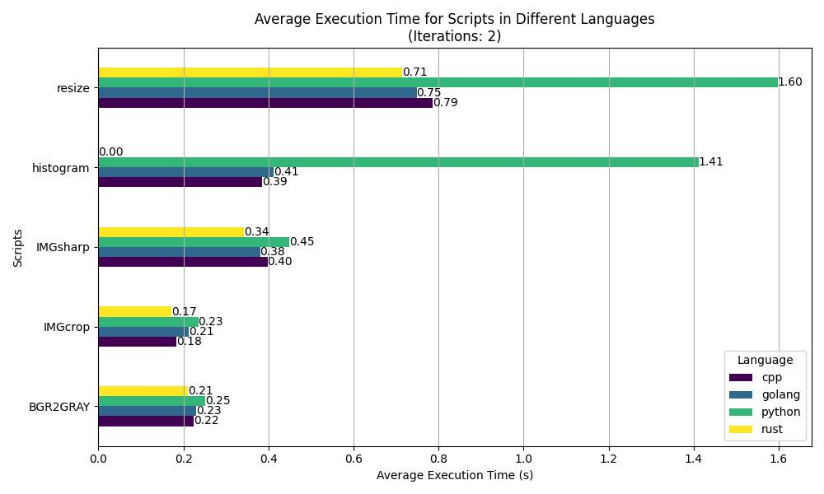


*Figure: Single Run Benchmark Results*

When analyzing the single run results, we can make initial assumptions as to which languages might be more suitable for image processing overall. We see that both Golang and Rust trade blows in every benchmark, which can be attributed to margin of error. C++ is not far behind with occasional victories as well. Python on the other hand seems to struggle on the resize and histogram implementations. The group member responsible for creating the python implementations mentioned a deprecation warning for the library functions regarding Histogram creation. On the subject of Histogram and as mentioned previously, the Rust implementation of the histogram creation was not working so it received a DNF.

This reinforces the importance of having iterative tests, as this result might not tell the whole story, especially for Python. Having the same benchmark ran back-to-back multiple times can give a more realistic view of the language’s performance. Initial assumptions are that the modern languages Rust and Go are solid choices for these tasks.

1. Iterative Run



*Figure: Iterative Run Benchmark Results*

In our iterative testing, we deliberately opted for just two consecutive runs to see if there would be subtle nuances that might not have appeared in a single run evaluation. This approach aimed to uncover any minor inconsistencies present in seemingly stable languages identified during single-run tests. Notably, this strategy shed light on intriguing observations, particularly evident in Python's performance.

Python, although proficient in various aspects, exhibited discrepancies, especially concerning image resizing and histogram operations. This anomaly may stem from potential inefficiencies in its underlying C++ libraries, signifying an area warranting further exploration and potential optimization.

Contrastingly, the consistency between C++, Rust, and Go remains striking. Their consistent performance reaffirms their robustness, demonstrating minimal variance between iterations. This reaffirmation of their neck-and-neck performance further solidifies their standing as formidable choices for OpenCV use. As the number of iterations increases, it holds the promise of offering more profound insights into the nuances and trends across multiple runs, potentially refining our understanding of language behavior and performance in image processing tasks.

# Conclusion

Overall, this project proved to be a success. The four languages used in this project successfully implemented each of the five OpenCV functions and generated its own performance metrics. For the most part, the performance of each language showed to be quite similar to each other, aside from the histogram and resizing functions. Ultimately, Rust came up on top, with performance as much as 77% faster than the weakest language. This could be due to Rust’s efficiency with low-level control and memory safety, since OpenCV is generally considered a low-level computer vision library.

##### References

1. C. Kanan and G. W. Cottrell, “Color-to-Grayscale: Does the Method Matter in Image Recognition?,” PLoS ONE, vol. 7, no. 1, p. e29740, Jan. 2012, doi: https://doi.org/10.1371/journal.pone.0029740.
2. J. Österberg, “Rust and OpenCV,” Medium, Feb. 14, 2022. <https://blog.devgenius.io/rust-and-opencv-bb0467bf35ff>
3. “OpenCV: Installation in Linux,” docs.opencv.org. https://docs.opencv.org/4.x/d7/d9f/tutorial\_linux\_install.html (accessed Dec. 06, 2023).
4. “OpenCV: Performance Measurement and Improvement Techniques,” docs.opencv.org. https://docs.opencv.org/3.4/dc/d71/tutorial\_py\_optimization.html
5. “OpenCV Python Tutorial,” GeeksforGeeks, Jan. 30, 2020. https://www.geeksforgeeks.org/opencv-python-tutorial (accessed Dec. 06, 2023).
6. “opencv - Rust,” docs.rs. https://docs.rs/opencv/latest/opencv/
7. [6]J. Österberg, “Rust and OpenCV,” Medium, Feb. 14, 2022. https://blog.devgenius.io/rust-and-opencv-bb0467bf35ff