PC-2024/2025 Performance evaluation on a random image agumentation pipeline

Edoardo Branchi
E-mail address
edoardo.branchi@edu.unifi.it

Abstract

How many times have you lost precious time with scarse image datasets?
How many times did you wish for a simple and straightforward way to expand the number of images at your disposal? Say no more.
This report presents Albumentations Roulette, a simple way to apply a random number of transformations to an image, in both a sequential and parallel approach using the Albumentation library. We examine the performance differences between these approaches, we present a comparative analysis of sequential and parallel implementations with a focus on execution time and scalability across different batch sizes, providing insights into the benefits and trade-offs of parallel processing in image augmentation workflows.

1. Introduction

Image augmentation is an important technique in computer vision and machine learning, used to increase dataset diversity and size to improve model robustness. While working on a single-image transformation is relatively straightforward, generating multiple augmented variants of images can become computationally intensive, making it an ideal case study for parallel processing optimization.

1.1. Key Characteristics

The key aspect of image agumentation is to generate new images based on a single one. This new generated images are transformations of the base image, transformations that include operations like cropping, applying various types of blurring, color corrections, compressions and so on. While this kind of operations are simple and lightweight for small images and a small number of generated images they can become very computationally heavy on RAM, Disks and CPU.

1.2. General implementation

A common pipeline for image agumentation is to build a flow of transformation, with various scales that are applied sequentially to an image. This process is usually applied to lot of images sequentially, loading the base image, transform it and than save it to the disk. While this process is running the image is saved in RAM until it is saved to the disk, this problem and how to circumvent it is explained better in the following sections.

2. My Implementation Overview

My implementation to tackle the augmentation problem consist mainly in an array of the most common transformations that can be applied to an image, this array ideally can be expanded to all the transformations provided by the library. When calling the main function giving it an image and the number of desired output images, the function applies between 1 and 5 transformation to each image at random picking it from the array and producing the wanted number of transformed output images. One important things to notice is that we are trying to measure performance calculating the transformations, so to exclude RAM dimensions and I/O delays the transformation is done and the isn't returned, so sent directly to the shadow realm of the garbage collector.

2.1. Sequential Albumentation Roulette

The sequential implementation of "generate_transfomed_images" relies on a for loop that runs a number of times equal to the number of image that the user wants to generate. This sequential approach should be slower than the parallel one especially for a large number of images.

2.2. Parallelized Albumentation roulette

The parallelized instead spawn a number of **Process** equal to the number of CPU Cores available. Each process create a number of images equal to the total images divided by the number of process apart from Core 1 that

also will process the remaining images in case the number of images divided by the core number is not a perfect division. In this way the load is distributed across the CPU granting us the possibility to use all the compute capacity available. I expect this to be much faster than the sequential implementation.

3. Experimental Setup

The tests are conducted on a test setup with the following components. The processor has 4 cores and 8 threads .

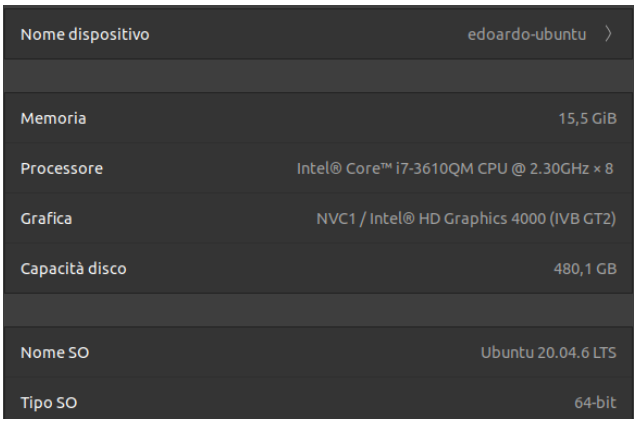

Figure 1. Specs of the test environment

The algorithm is applied to three images of various dimensions, for simplicity I will refer to it as

- **Small** image: 620x320 pixels.
- **Medium** image: 1280x853 pixels.
- **High** image: 2600x1498 pixels.

An example can be seen in Figure 2.

Figure 2. One of the images used for testing

To produce results the execution time is measured only on the computational part of the program leaving out the load and the display of the result and the color space conversions using "time.time()".

[1]https://www.intel.com/content/www/us/en/products/sku/64899/intel-core-i73610qm-processor-6m-cache-up-to-3-30-ghz/specifications.html

4. Results and Discussion

The program was tested using all three image sizes for producing 1000, 5000 or 10000 output images in both the sequential and the parallel version. Also a speedup is calculated afterwards to understand the impact of the modification done. In the following table we report the result obtained.

Size	# of output	Sequential	Parallel	Speedup
small	1000	2.294	0.932	2.46
small	5000	11.15	3.346	3.33
small	10000	21.714	6.262	3.46
medium	1000	8.394	3.138	2.67
medium	5000	41.506	13.836	2.99
medium	10000	75.554	27.496	2.74
big	1000	23.478	9.452	2.48
big	5000	116.424	47.7	2.44
big	10000	234.314	96.048	2.43

Table 1. Data for Sequential, Parallel, and Speedup performance across different input sizes, with dashed lines separating inputs.

4.1. Performance Comparison

In this section the results from the table seen in section 4 are plotted and we can see interesting patterns. As expected, in Figure 3 we can see that the image dimensions plays an important role in the total time of execution. As the image size increase the execution time almost quadruple for both sequential and parallel execution.

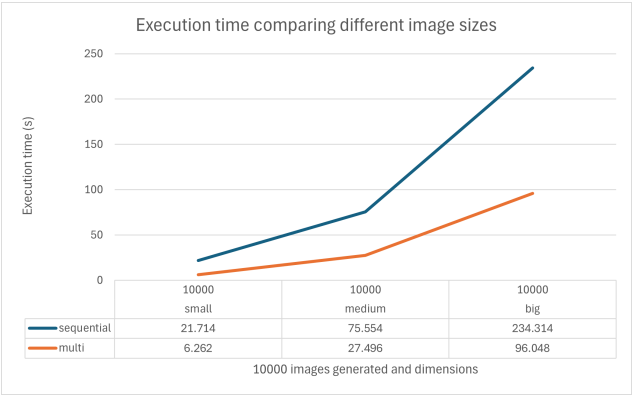

Figure 3. Results varying the image dimensions

Also we can perform a test using the same image size but varying the number of image generated producing an interesting result as can be seen in Figure 4. Between 1000 and 5000 generation we measured a 5 time increase in execution time that is linear to the five time increase in the number of image generated, this pattern also is valid when comparing

5000 to 10000 to doubling in the number of image corresponds a doubling in execution time.

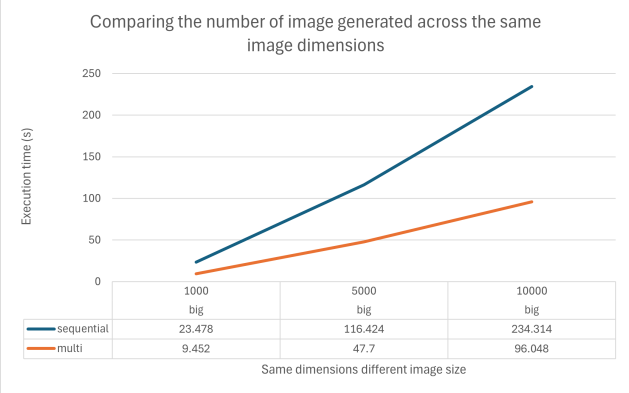

Figure 4. Results varying the number of generated image

In figure 5 we plotted a round up of all tests performed for both sequential and parallel execution.

Figure 5. All tests performed

In 6 we calculated and plotted the speedup achieved between the sequential and parallel versions. With a small image we can see some "noise" in the plot, this could be due to the small overall execution time so the allocation of processes and accesses to the memory could overcome the actual execution time, taking into consideration that this "noise" is less marked in the medium size image and almost gone in the big image measurement.

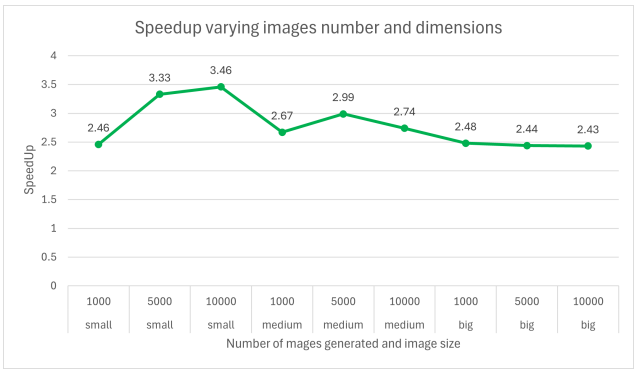

Figure 6. Speedup obtained

5. Conclusion

The parallel version is very simple to implement bringing and average of 2x the speed in execution time at minimum, so, we can affirm the parallel version is worth to create. Worth nothing also that between the two version there isn't an increase in readability difficult that for example we can find in a CUDA version for a program.

6. GitHub repository

Albumentation Roulette sequential
Albumentation Roulette parallel