PC-2023/24 Course Project Template

Elia Matteini 7148467
E-mail address
elia.matteini@edu.unifi.it

Filippo Zaccari 7148301
E-mail address
filippo.zaccari@edu.unifi.it

Abstract

Image augmentation is the process of creating new training examples by using already existing ones. Image wise it'll create slight changes by modifying their technical attributes. The topic of this paper will be to analyze two of the different ways an algorithm can implement image augmentation: parallel and sequential. We used python and Joblib to make a parallel version of the algorithm which dataset's is going to be a set of images of fish and the sea bed in which they live. We'll discuss the results and compare the differences in speedup obtained with the parallelization at the end of the paper.

Future Distribution Permission

1. Introduction

This paper contains an overview of two implementation of an Image augmentation algorithm [2], a sequential version and a parallel version. Both versions are implemented with Python using Albumentations [1] to execute the augmentation on the images.
Given a dataset of images the two programs will produces a larger dataset composed of a prefixed number of images per image.
To make a parallel version of the algorithm we used Joblib [3]. In the end of this report we will see the advantages of the parallelization and the speedup gained.

2. Image augmentation algorithm

Image augmentation is the process of creating a new pool of images by using already existing ones, these images will differ from the original ones in brightness, orientation, saturation etc.
This technique is widely used due to the larger amount of samples it creates for deep neural networks, and because it allows the user to generate a vast dataset starting from a small amount of images, as shown in Figure 1, that can be used for training AI.

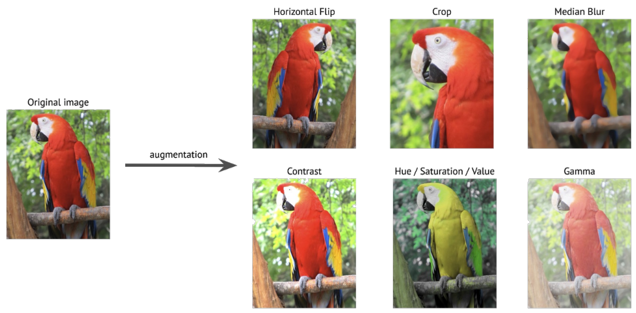

Figure 1: Example of imagine augmentation

As we can see, from just one image the algorithm generates 6 more images, each one different from the original and all the other ones. In this way it's very easy to obtain a large dataset to train, for example, an object detection AI.

3. Workflow

The very beginning of this project has been an accurate research about Image Augmentation. After we acquired sufficient knowledge on

the subject we tried to write an algorithm in Python that reads all the images inside a specific folder, from each one generates a certain number of augmented images and then saved them in another folder.
The code is basically divided into three parts:

- load the images;
- define the set of transformations;
- generate the augmented images.

In particular, with the line below the algorithm load all the images using the OpenCV library (cv2) [5].

```
images_from_folder = os.listdir(folder_in)
images = [cv2.imread(folder_in + image) for
    image in images_from_folder]
```

Then using Albumentations library we created a pool of transformations:

```
transform = A.Compose([
    A.HorizontalFlip(p=1),
    A.VerticalFlip(p=1),
    A.Rotate(limit=360, p=1),
    A.RandomBrightnessContrast(p=1),
    A.GaussianBlur(blur_limit=(3, 9), p=1),
    A.ColorJitter(brightness=0.6, contrast
        =0.6, saturation=0.6, hue=0.6, p=0.5),
    A.RGBShift(r_shift_limit=40, g_shift_limit
        =40, b_shift_limit=40, p=0.5),
    A.ChannelShuffle(p=1),
    A.RandomGamma(p=1),
    A.Blur(p=1),
    A.ToGray(p=0.5),
])
```

finally, for each image we applied a prefixed number of transformation from the pool defined above, and as done before we used cv2 to save them into a different folder:

```
j = 0
for image in images:
    i = 0
    for i in range(num_augmentations):
        augmented_image = transform(image=
            image)['image']
        cv2.imwrite('./' + folder_out + '/
            augmented_image' + str(j) + '_' +
            str(i) + '.jpg', augmented_image)
    j = j + 1
```

3.1. Parallel version

Unlike Java and other programming languages, Python doesn't support multithreading. Not only does Python not natively support it, but also has what's called GIL (Global Interpreter Lock) [6] that prevents it.
Realize a parallel algorithm its not as easy as it seems after this consideration, but there is an escape which consists of using a library called Joblib that allows you to take advantages of multi processing.
Clearly this two technique are not the same in term of performance; we cannot expect the same speedup from multiprocessing compared to multi threading.
Joblib provide the class `Parallel` that can be combined with `delayed` to parallelize a function; the latter is a decorator that allows you to capture the arguments that will be passed to the function.
In this case the function that has been parallelized requires two arguments: `imagesPath` and `num_augmentations`; regarding `imagesPath` we decided to divide the images dataset into batches depending on the number of processes that are in going to be created.
So in first place we defined the batches dimension:

```
imageBatchSize = math.ceil(len(
    images_from_folder) / num_process)
```

and then created n batches where

$$n = number\_of\_image/batch\_size$$

and assigned all the images to the various batches:

```
batches_for_process = [images_from_folder[i:i
    + imageBatchSize] for i in range(0, len(
    images_from_folder), imageBatchSize)]
```

At this point all the needed elements for the parallelization are ready, we can proceed calling `Parallel` as follows:

```
Parallel(n_jobs=num_process)(delayed(
    imgAugmentation)(batch, num_augmentations)
     for batch in batches_for_process)
```

This line of code creates `num_thread` processes, each one running `imgAugmentation` function with the batches defined above, `imgAugmentation` do all the work described in the sequential version.
About OpenCV there is one more thing to talk about, and that is multithreading: cv2 library use multithreading in many of its algorithms and that could lead to overhead of the system, to avoid it we used `cv2.setNumThreads(0)` to ensure that the cv2 functions are executed sequentially.

4. Speedup

To test the program we used a Python script that run 10 times the sequential algorithm with a specific configuration, takes the execution time for each one and then calculate the average execution time of the sequential algorithm. The same thing has been done with the parallel algorithm using the same configuration starting from 1 process and increasing that number by the power of 2 since 32 processes; at each iteration it calculates the speedup of the parallel algorithm compared with the sequential one.
To calculate the speed up we used this formula:

$$S_p = t_s/t_p$$

where t_s is the sequential execution time and t_p the parallel one.
In conclusion the script:

```
1: calculate the sequential average execution time
2: for num_process ← 1 to 32 do
3:     calculate the parallel average execution time
4:     calculate the speed up
5:     write the speed up into a csv file
6: end for
```

Once we had collected all the measurements we were able to calculate the efficiency of the parallel algorithm

$$E_p = S_p/p$$

Where S_p represent the speedup calculated above and p is the number of cores. The achieved efficiency is 40%.

4.1. Hardware

Hardware specs of the machine where the code has been tested are:

Laptop System: ASUSTeK
Distro: Ubuntu 22.04.4 LTS
Kernel: 6.5.0-21-generic x86_64
CPU: quad core (8 thread) AMD Ryzen 7 3750H 2.3 Ghz
Memory: 16 GB

4.2. Measurements

To have a complete vision about the performance obtained with the parallelization, the program has been tested on two dataset with 100 and 200 images with the same resolution 1280 x 720.
With the script described before and the 2 dataset, we collected the measurements to calculate the speedup as shown in Figure 2 and Figure 3.

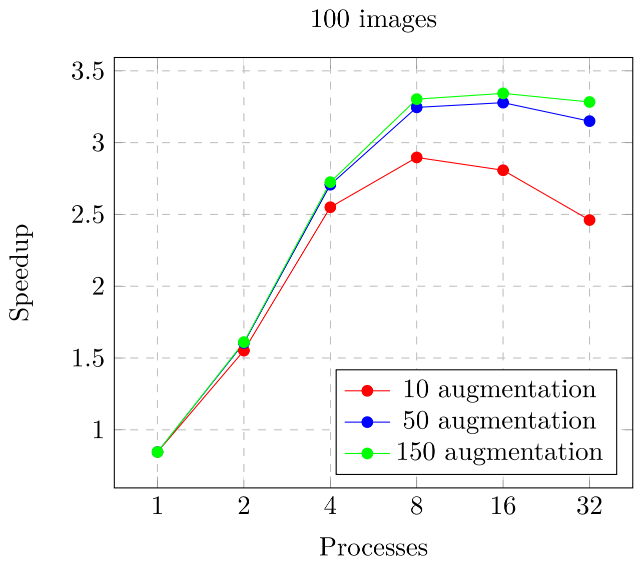

Figure 2: Speedup 100 images 1, 2, 4, ..., 32 processes

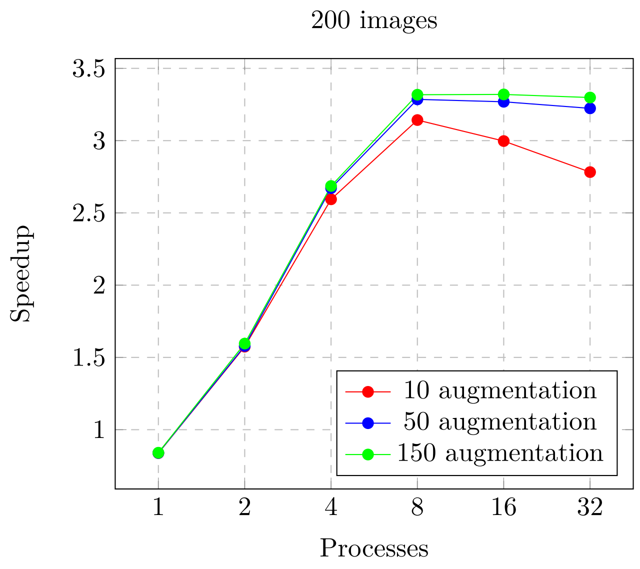

Figure 3: Speedup 200 images 1, 2, 4, ..., 32 processes

5. Conclusion

In this paper we have seen how the parallel implementation of image augmentation results in faster and more efficient execution compared to its sequential version.
We can do a consideration about RAM saturation: the size of the dataset or the number of transformations applied to the images could saturate the RAM if their number increases too much (depending on the hardware).
To prevent this problem we chose to write the images to disk as soon as they were processed, instead of writing them all together at the end of the process.

References

[1] Albumentation. Albumentation library. URL: `https://albumentations.ai/docs/introduction/image_augmentation/`.

[2] Datacamp. A complete guide to data augmentation. URL: `https://www.datacamp.com/tutorial/complete-guide-data-augmentation`.

[3] joblib. Joblib: running python functions as pipeline jobs. URL: `https://joblib.readthedocs.io/en/stable/`.

[4] DIVE INTO DEEP LEARNING. Image augmentation. URL: `https://d2l.ai/chapter_computer-vision/image-augmentation.html`.

[5] OpenCV. Opencv library. URL: `https://opencv.org/`.

[6] Real Python. What is the python global interpreter lock (gil)? URL: `https://realpython.com/python-gil/`.