PC-2023/24 Image Augmenter

Author
E-mail address
francesco.marchini1@edu.unifi.it

Abstract

Image augmentation involves creating new training examples from existing ones. This is particularly useful for deep neural networks that require a large amount of training data to achieve good results and prevent overfitting. This paper presents an efficient parallel system that takes an image dataset as input and returns an augmented dataset. The code is parallelized using the joblib Python library.

1. Introduction

This report describes the use of *joblib* [1] library for parallelising Python code. In particular, we propose a program that, when given input images, produces a large dataset of augmented images using the *Albumentation* [2] library. The aim of this research is to achieve the highest possible performance by making the best possible use of code parallelisation. Starting with a dataset of 28 images, we replicated them 13 times using various transformations, resulting in a dataset of 364 images. The code repository is available on *GitHub* by clicking on the following link: Parallel-image-augmenter.

2. Image Augmentation algorithm

This section explains how the code works, focusing on the most relevant parts for the future parallelization. The implementation of the Image Augmenter algorithm can be divided into various steps:

- create a vector of transformations;
- load the images from disk;
- apply all transformations to all images;
- save the results to disk.

The first section are only responsible for setting up the environment, instantiating the transformations for the future operations. The remaining parts are the core of the code, so we will act on them to minimise the execution time of the whole program. The `joblib.Parallel` class will be used to make the code multi-threaded, allowing it to independently read, process, and write images across all available processes.

2.1. Parallelization

This section will be use to discuss the software design choice, explaining how the code works. To evenly distribute the workload among all processes, we must create a list of paths that will serve as input for the `parallel_read` function. This function takes a list of image paths and return the correspondents images. The main process is responsible of the correct number of paths to give at each sub-process, so the whole paths list is divided in batch following the number of processes available.

```
image_paths = next(os.walk('in_images'))[2]
num_images = len(image_paths)

batch_size = round(num_images // num_thread)
path_batches = [image_paths[i:i+batch_size] for i
     in range(0, num_images, batch_size)]
```

The `augment_images` function is the core component of the program. This function receives

a list of image paths and a list of transformations as input and is responsible for calling the `read_images` function to read the images and then write them to disk. The function body consists of two nested for loops: an outer loop that iterates through the images and an inner loop that iterates through the transformations. Every image is transformed and store in a list to be saved to disck.

```
def augment_images(images, transformations):
    cv2.setNumThreads(0)
    images = _parallel_read(image_to_read)
    transformed_images = []
    for i, image in enumerate(images):
        for j, transformation in enumerate(
    transformations):
            transformed_image = transformation(
    image=image)['image']
            transformed_images.append(
    transformed_image)
    size = len(transformations)
    for i, name in enumerate(image_to_read):
        for j, image in enumerate(
    transformed_images[i * size: (i+1) * size]):
            out_path = f'out_images/augmented_{j}
    _{name}'
            cv2.imwrite(out_path, image)
    return None

def _parallel_read(image_to_read):
    images = [cv2.imread(f'in_images/{image}')
    for image in image_to_read]
    return images
```

The `cv2.setNumThreads(0)` is use to set the *cv2*'s available threads to only one to avoid overhead due to the creation of additional threads. By utilising the `Parallel` class from the *joblib* library, we can input each batch of images into the augmentation function using a specific number of processes: `num_proc`.

```
transformed_images = Parallel(n_jobs=num_proc)(
    delayed(augment_images)(batch,
    transformations) for batch in image_batches)
```

The variable `num_proc` is controlled by the user's input. If not specified, the machine will allocate all available threads.

```
if len(sys.argv) > 1:
    num_thread = int(sys.argv[1])
else:
    num_thread = multiprocessing.cpu_count()
```

2.2. Process overhead

This software design choice are justified by the results of some previous experiments. If we decide to load the images on the main process and then pass them to the sub-processes, we incur a large overhead, which is compounded if we wait for the transformed images to be returned to the main process for storage. Hence the decision to delegate all read and write operations to disk to individual processes, to avoid excessive workload required only to transfer images through process pipes and not to transform them.

3. Speedup

The performance achieved with the sequential version will now be compared to that of the parallel version in terms of execution time. In order to reach this task, a Python script will be utilized to execute both sequential and parallel versions of the program varying the number of threads used. The script calculates the speedup achieved at each iteration and generates a graph displaying the results in terms of speedup for threads. All tests were performed on an 11th generation Intel i7 [3] CPU with 4 cores and 8 threads.

3.1. Final results

The speed up is computed as follow:

$$S_P = t_s/t_p \tag{1}$$

Where t_s is the execution time of sequential program and t_p the execution time of the parallel program. In this case, we run the parallel code using between one and sixteen threads to evaluate performance at each step. As expected given the CPU used, the best approach is achieved by utilising eight threads: $S_P = 2.39$. All result are available in Fig. 2 and on Table **??**.

Considering the 4 cores of the 11th Intel i7 used, we will measure our speedup in terms of efficiency using the following formula:

$$E_P = S_P/P \tag{2}$$

Where P is number of CPU units used. The achieved efficiency is: $E_P = 0.5970$; highlighting an efficiency of 60% on our machine. The second graph (Fig.3) displays the results of the same experiment, indicating only the execution times

Thread #	Speedup
1	1
2	1.4569
3	1.8399
4	1.94467
5	2.0062
6	2.1715
7	2.1891
8	2.3877

Table 1: Speedup increases as the number of threads used increases.

of the parallel algorithm as the number of threads used changes.

4. Conclusion

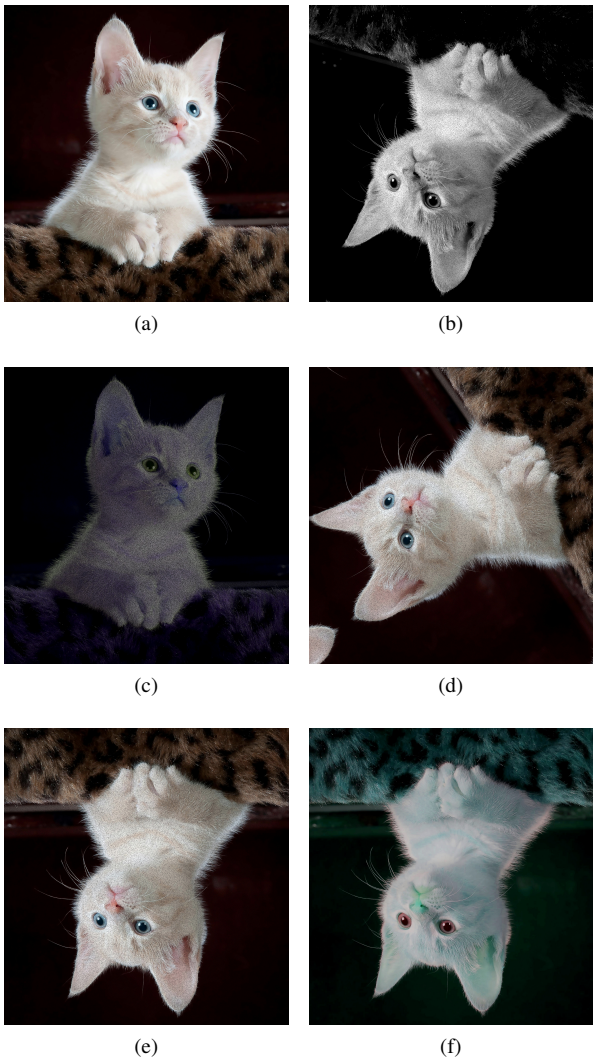
In our case all tests were performed using a machine with 16GB of RAM and 1GB of swap area. This creates a huge bottleneck as the size of the initial dataset or the number of transformations to be applied to the images increases. RAM is very easily saturated by these types of programs, requiring access to mass memory to write temporary data to end the execution, resulting in excessive overhead and loss of performance. Testing the program on this machine resulted in the compromise of processing 28 high-resolution images with a maximum number of 13 transformations. This prevents the RAM from becoming saturated, which would require disk writes. Fig. 1 shows an example of some transformations applied on a single image.

References

[1] Joblib, 2008-2021. https://joblib.readthedocs.io/en/stable/.

[2] Albumentations, 2020. https://albumentations.ai/.

[3] Intel. i7 11th generation, 2021. https://www.intel.com/.



(a) (b) (c) (d) (e) (f)

Figure 1: Original image with a sample of 5 augmented images

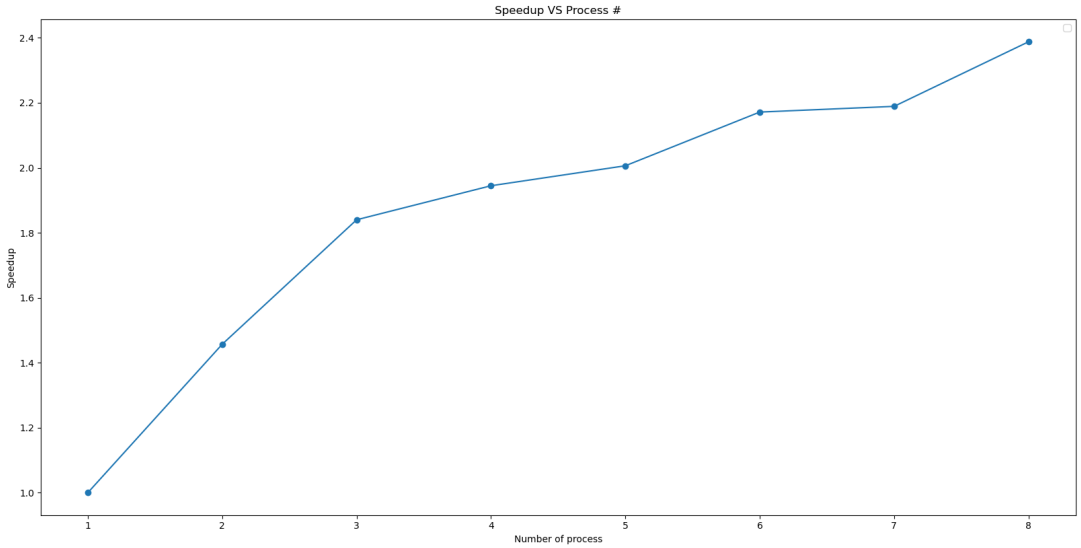


Figure 2: Speedup VS Number of Processes to augment 28 images with 13 transformation.

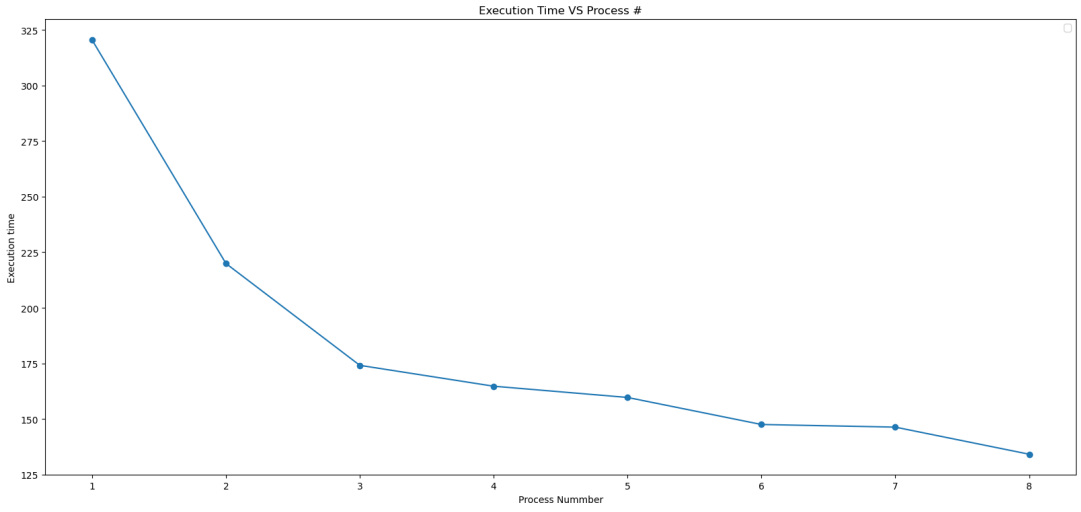


Figure 3: Execution time VS Number of Processes to augment 28 images with 13 transformation.