Parallel Programming: Project Work

Angelo Caponnetto

November 2024

Abstract

This report summarizes the work done to parallelize the augmentation functions of the Albumentations library. Two different types of parallelization are addressed: the first uses Python's multi-threading, and the second leverages functions written in CUDA. Speedup curves will be collected, and the results will be compared.

1 Description of the problem

The goal of this work is to accelerate the augmentation functions of the Albumentations library. Several functions have been analysed and implemented: horizontal and vertical flip, conversion in grey scale, channel shuffle, crop and resize, occlusion, contrast and brightness.

Figure 1: Different types of augmentations.

Each function was implemented in two distinct versions: one utilizing Python's multi-threading capabilities and the other designed specifically for execution on CUDA-enabled GPUs.

2 Dataset and pre-processing

The dataset consists of 1,500 RGB images sourced from the ImageNet dataset, each with dimensions of 375x500 pixels. To maximize the efficiency of the parallelization process, the dataset is divided into batches before applying the transformations. The impact of batch size on performance will be analyzed in the following sections.

3 Python's multi-threading

The class **ParallelAlbumentation** has been written as follow. It takes images in form of matrix and divides all the dataset in batches of specific dimension.

```
1: class ParallelAlbumentation:
2:     constructor(dataset, b_size, n_w):
3:         dataset ← dataset
4:         b_size ← b_size    #batch size
5:         n_w ← n_w    #number of workers
6:         batches ← divide dataset in batches
```

The main method of the class is **transform_applier**: it applies the transformation on each batch in multi-thread mode.

Algorithm 1 transform_applier (transformation)

```
1: results = [ ]
2: with ThreadPoolExecutor(n_w):
3:     for batch in batches:
4:         results ← append transformation(batch)
5: return results
```

Note: *transformation* is any function of Albumentation library.

4 CUDA - Custom Augmentation Class

The class **CustomAugmentation** has been written as follow. Similar to the previous approach, the dataset is divided into batches, with each batch loaded onto the GPU. Images within each batch are stored in a flattened format to facilitate GPU-based computations. Additionally, memory space is allocated on the GPU for storing the augmented versions of the input images. The implementation leverages the **cupy** library.

```
1:  class CustomAugmentation:
2:      constructor(dataset, b_size, n_thr):
3:          dataset ← dataset
4:          n_thr ← n_thr    # threads per block
5:          b_size ← b_size   #batch size
6:
7:          batches_GPU ← divide dataset in
8:                        flatten batches
9:                        and load on GPU
10:
11:         new_img _GPU ← allocate memory
12:                        space on GPU for
13:                        augmented images
14:
15:         # x, y, z dim of CUDA blocks
16:         block_dim = (n_thr, n_thr, 1) # x, y, z
17:                     dim of CUDA blocks
18:
19:         # x, y, z dim of CUDA grid
20:         grid_dim = [(img_width + n_thr) / n_thr,
21:                     (img_height + n_thr) / n_thr, 1]
```

4.1 CUDA functions

Each function is a global function that operates on evey image of the batch and accepts as variables: *batches_GPU*, *new_img_GPU, b_size, img_width, img_height, img_channels* and all the other parameters that are useful for the specific transformation.

Algorithm 2 global void CUDA transformation

```
1: int  idx ← thread idx
2: int  idy ← thread idy
3:
4: if idx < n_thr  and  idy < n_thr:
5:     for i in range(0, b_size):
6:         apply the trasformation on image[i]
```

5 Performance evaluation

5.1 Hardware description

Following results were collected using the CPU AMD Ryzen 7 3750H and the GPU NVIDIA GTX 1660 Ti with max-Q. Here we report same characteristics:

- **CPU:** number of cores: 4, number of flows: 8, Frequency: 2.3 GHz, L1 Cache: 384 KB, L2 Cache: 2 MB, L3 Cache: 4 MB
- **GPU:** Architecture: Turing, Memory Size: 6 GB,Bandwidth: 288.0 GB/s, CUDA: 7.5, Shading Units: 1536, SM Count: 24, L1 Cache: 64 KB (per SM), L2 Cache: 1536 KB

5.2 Evaluation of the batch size

First, a transformation has been selected (horizontal flip) and times have been collected for batches of different size. $n\_thr = 16$ and $n\_w = 8$ at every run. The results are shown in 2.

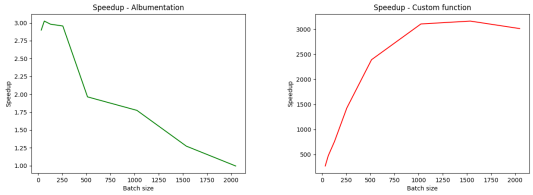

Figure 2: Speed Up for different batch sizes.

The observed trends are different: multi-threading achieves optimal performance with relatively small batch sizes, whereas CUDA functions perform best with larger batches. Additionally, the speed-ups

differ significantly, with a gap spanning three orders of magnitude.

For the next studies, batch sizes will be fixed:

- $b\_size\_albumentation = 128$
- $b\_size\_custom = 1024$

5.3 Speed-ups evaluation

Speed-up curves have been collected for every implemented function. Trends are similar, even if reached speed ups and execution times change. In figure 3 and 4, data for horizontal flip functions are reported (see appendix A for the other results).

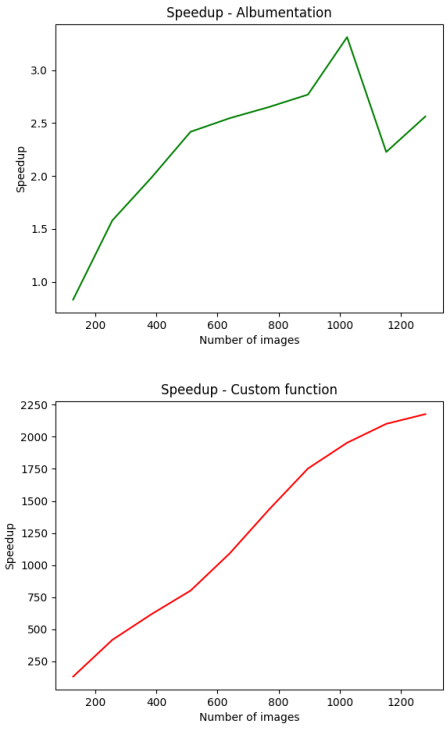

Figure 3: Speed Up for horizontal flip.

- **ParallelAlbumentation:** The maximum speed-up is achieved when the number of images is around 1024, that is, $128(batch\ \ size) \times 8(active\ \ threads)$. This result shows that performance improvement reaches its optimum when all threads are equally engaged, avoiding the overhead of idle threads. With a lower number of images, the overhead from idle threads can reduce the speed-up.

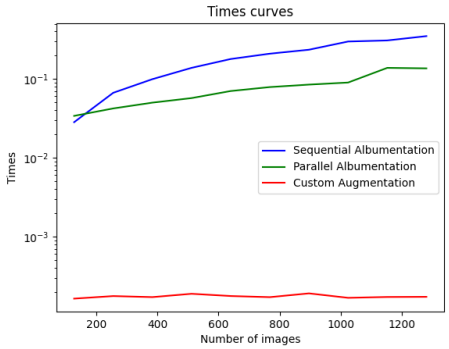

Figure 4: Times for horizontal flip.

- **CustomAugmentation:** For the custom class, the speedup increases as the number of images grows. This could be due to the fact that for a single image, the transformation operations are relatively light, and the efficiency improvements are less noticeable. However, when the number of images increases, the operations take more time, making the benefits of parallelization more apparent.

6 Conclusion

The project successfully demonstrates the effectiveness of parallelization in accelerating image augmentation functions from the Albumentations library. Two approaches were implemented: Python multi-threading and CUDA-based transformations.

Python multi-threading showed reasonable improvements for small batch sizes, making it suitable for environments without GPU support or when handling lightweight datasets.
CUDA-based implementations, on the other hand, achieved significantly higher speed-ups, particularly

with larger batch sizes. By leveraging GPU parallelism, these implementations efficiently processed transformations, such as horizontal and vertical flips, greyscale conversion, and channel shuffling. The scalability and performance gains observed with CUDA make it the preferred choice for high-performance computing tasks in image processing.

The evaluation highlighted a critical trade-off: while Python multi-threading is easy to implement and sufficient for smaller tasks, CUDA's power is unlocked with larger datasets and batch sizes. This reinforces the importance of choosing the right parallelization strategy based on the specific requirements and hardware capabilities of the system.

In conclusion, the CUDA-based approach proved to be far superior in terms of speed and scalability, making it the ideal solution for large-scale image augmentation tasks.

A All the results

Here there are reported all the results for every implemented function.

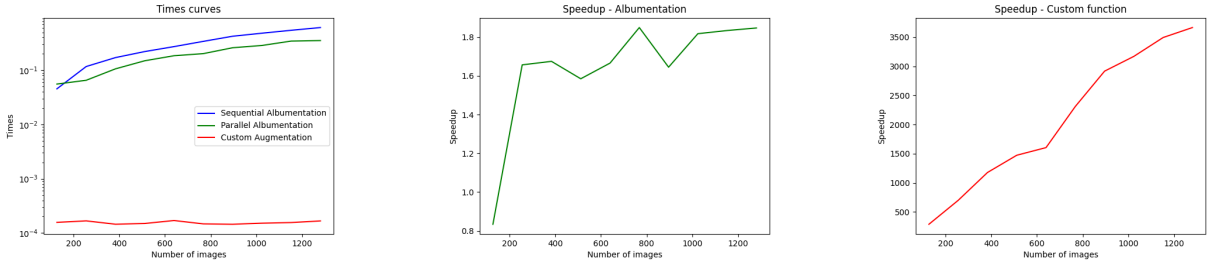

Figure 5: Speed Up for brightness.

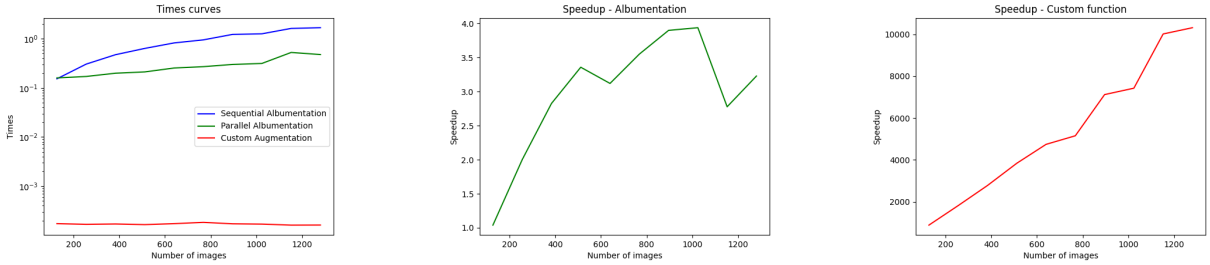

Figure 6: Speed Up for channel shuffle.

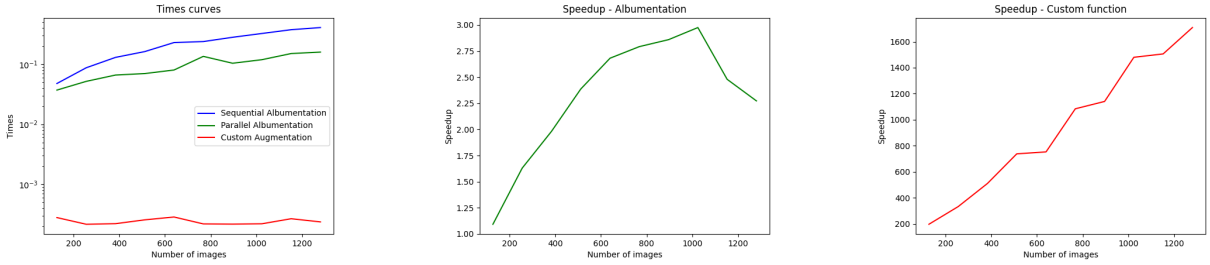

Figure 7: Speed Up for crop.

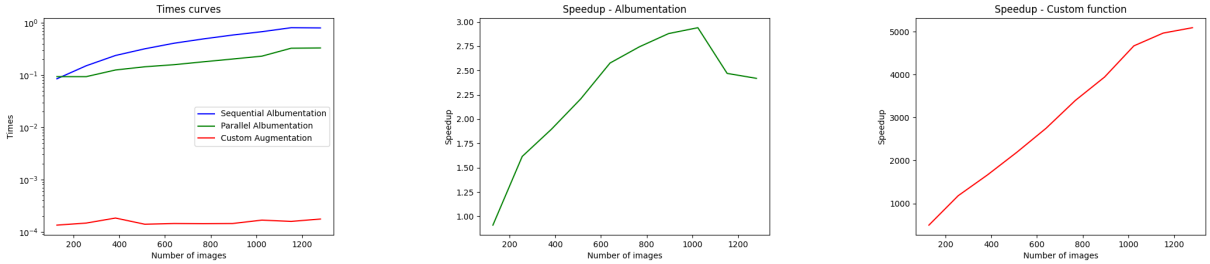

Figure 8: Speed Up for grey.

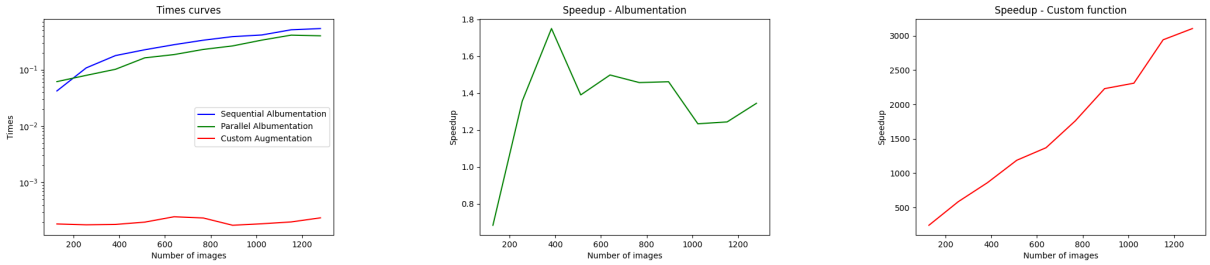

Figure 9: Speed Up for occlusion.