PC-2023/24 Parallel Lowe's Algorithm

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

Abstract

David Lowe's algorithm is well-known for searching for SIFTs (Scale Invariant Feature Transforms) within images. These features are extracted to create a unique key that identifies the image. The key, once saved within a vector, can be used to compare images and determine their degree of similarity. This study focuses on two approaches to parallelizing the algorithm using Python.

Future Distribution Permission

1. Introduction

In this report, we are going to examine parellalisation in python. In particular, we propose a version of Lowe's algorithm for searching for SIFTs within images. The aim of this study is to optimise sequential code by running it on multiple threads simultaneously, using the *multiprocessing* library [1]. The algorithm in question is based on difference of Gaussian for Scale Invariant Feature Transform detection. To highlight the property of being scale and transformation invariant, we utilise an image dataset that has been augmented. This not only enriches our dataset but also enables us to verify if the program accurately identifies the same SIFTs from the transformed images.

2. Lowe's algorithm

This section explains how the code works, focusing on the most relevant parts for the future parallelization. As previously mentioned, Lowe's algorithm is capable of identifying SIFTs within an image. For this purpose, all input images are processed through four basic steps:
(1) The image is replicated k times by gradually adding more blur, creating a stack of images to work on.
(2) Each image is subtracted from the image of the underlying layer, according to the Difference of Gaussian (DoG) method. In this way, we obtain a Gaussian difference matrix for each scaled layer.
(3) Analysing one matrix at a time, we look for its major peaks, discarding the less significant ones. This gives us interesting points that are candidates to be SIFT features.
(4) The points that actually become SIFT are selected on the basis of an a priori defined threshold. This results in a series of point matrices representing the features found at each scale level. By merging all these matrices, it is possible to apply the points to the original image to see the result.

2.1. Parallelization

To implement the algorithm described above, we use the *OpenCV* Python library, specifically two main for loops are used to compute the SIFTs of the image at each octaves and scale. At each scale, the operation of computing the SIFT features is independent of each other.

```python
def detect_keypoints(image, num_octaves=4,
    num_scales=5, sigma=1.6, contrast_threshold
    =0.04):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY
    )
    sift = cv2.SIFT_create()

    keypoints = []
    for octave in range(num_octaves):
        for scale in range(num_scales):
            scale_level = sigma * 2 ** (scale / num_scales)

            blurred = cv2.GaussianBlur(gray, (0, 0), sigmaX=scale_level)

            kp = sift.detect(blurred, None)

            if contrast_threshold is not None:
                kp = [k for k in kp if k.response > contrast_threshold]

            for k in kp:
                k.pt = tuple(np.array(k.pt) * 2 ** octave)

            keypoints.extend(kp)

        if octave < num_octaves - 1:
            gray = cv2.resize(gray, (gray.shape[1] // 2, gray.shape[0] // 2))

    return keypoints
```

It is clear that operations within the nested loop can be parallelised. Therefore, let's rewrite the *detect_keypoints* function to enable the execution of the second loop on multiple threads. A pool of threads is created, with one thread for each scale level, to perform each operation for calculating SIFTs on a separate process. Then, for each octave, a list of blurred images is created, one for each scale level. Before running the parallel code, we divide the vector of images into as many batches as there are levels of scale.

```python
def detect_keypoints(image, num_octaves=4, num_scales=5, sigma=1.6, contrast_threshold=0.04):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    keypoints = []
    pool_size = num_scales
    pool = multiprocessing.Pool(processes=pool_size)
    print('Number of processes: {}'.format(pool_size))
    for octave in range(num_octaves):


        scale_levels = [(sigma * 2 ** (scale / num_scales)) for scale in range(num_scales)]
        blurred_list = [cv2.GaussianBlur(gray, (0, 0), sigmaX=scale_level) for scale_level in scale_levels]

        batch_size = round(len(blurred_list) / pool_size)
        blurred_batches = [blurred_list[i:i+batch_size] for i in range(0, len(blurred_list), batch_size)]
        args = [(contrast_threshold, octave, batch) for batch in blurred_batches]

        results = pool.map(keypoints_processor, args)

        keypoints_list = [keypoint for keypoints_batch in results for keypoint in keypoints_batch]
        keypoints_restored = [cv2.KeyPoint(x=k[0][0], y=k[0][1], size=k[1], angle=k[2], response=k[3], octave=k[4], class_id=k[5]) for k in keypoints_list]
        keypoints.extend(keypoints_restored)
        # Resize image for the next octave
        if octave < num_octaves - 1:
            gray = cv2.resize(gray, (gray.shape[1] // 2, gray.shape[0] // 2))

    pool.close()
    pool.join()
    return keypoints
```

Using modern machines, we have noticed that the SIFT identification process for a single image is quite fast. However, processing a dataset of images is a bottleneck due to the large number of images that need to be processed.

2.2. Input Parallelization

Unlike before, we are now going to modify how an image dataset is passed as input to the code. In the sequential code version, we iterate through all the images and apply the *detect_keypoints* function one by one to find SIFTs.

```python
image_index = 0
for i in range(0, 7):
    image = cv2.imread('images/flower{}.jpg'.format(i))
    augmented_images = augmentation.data_augmentation(image)
    for augmented_image in augmented_images:
        keypoints = detect_keypoints(augmented_image, contrast_threshold=0.02)

        image_with_keypoints = cv2.drawKeypoints(augmented_image, keypoints, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

        output_image_path = 'results/flower{}_keypoints.jpg'.format(image_index)
        cv2.imwrite(output_image_path, image_with_keypoints)
        image_index += 1
```

In order to optimize the execution time of the algorithm, in the parallel code, we create a list of images, the dataset, and split it into batches based on the number of threads in use. After we create a pool of threads using the *multiprocessing* library to execute the *detect_keypoints* function and passing them the image batches. By following this strategy, we can evenly distribute the workload among the available threads and parallelize the sequential for loop.

```python
images = []
image_paths = ['images/flower{}.jpg'.format(i)
    for i in range(7)]
loaded_images = [cv2.imread(image_path) for
    image_path in image_paths]
for image in loaded_images:
    augmented_images = augmentation.
    data_augmentation(image)
    images.extend(augmented_images)

pool_size = multiprocessing.cpu_count() * 2
pool = Pool(processes=pool_size)

batch_size = round(len(images) / pool_size)
image_batches = [images[i:i+batch_size] for i in
    range(0, len(images), batch_size)]

keypoints_batches = pool.map(
    detect_keypoints_for_batch, image_batches)
```

3. Speedup

This section analyses the two parallel versions of the original code, highlighting the more efficient one based on the achieved speedup. All tests were performed on an 11th generation Intel i7 [2] CPU with 4 cores and eight threads.
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.

3.1. Parallel algorithm

To evaluate the performance of Lowe's algorithm in parallel, we executed the code while varying the number of scales used, between 1 and 16. We perform five tests for the sequential code and five for the parallel code for each scale, and then calculate the average execution time. Finally, the average speedup achieved on all tests is $S_P = 1.7393$. More results are displayed on Fig. 1. As mentioned earlier, this approach does not significantly improve execution time. We recommend using this method when processing large images and when interested in processing multiple levels of scale.

3.2. Parallel input

In this section we tested a dataset of 35 images, passing it first as input to the sequential program and then to the parallel program, each time changing the number of operative threads (always between 1 and 16) in order to assess its performance. Fig. 2 shows the reduction in execution time of parallel code as the number of threads increases until a stall is reached. The optimal approach is achieved, before reaching a massive context-switch, by utilising nine threads: $S_P = 3.52$. All results about speedups are available in Fig. 3.

4. Conclusion

This study proposes two different approaches to the parallelisation of a problem such as the identification of SIFTs within one or more images. When the aim of the research is to analyse a single, large image, it is preferable to adopt an approach that parallelises the internal operations of Lowe's algorithm itself. If, on the other hand, we are interested in processing a massive dataset of images, then it is better to use an approach that allows us to distribute the workload over several processes. This allows us to process multiple images simultaneously in parallel.

References

[1] Multiprocessing library. https://docs.python.org/3/library/multiprocessing.html.
[2] Intel. i7 11th generation, 2021. https://www.intel.com/.

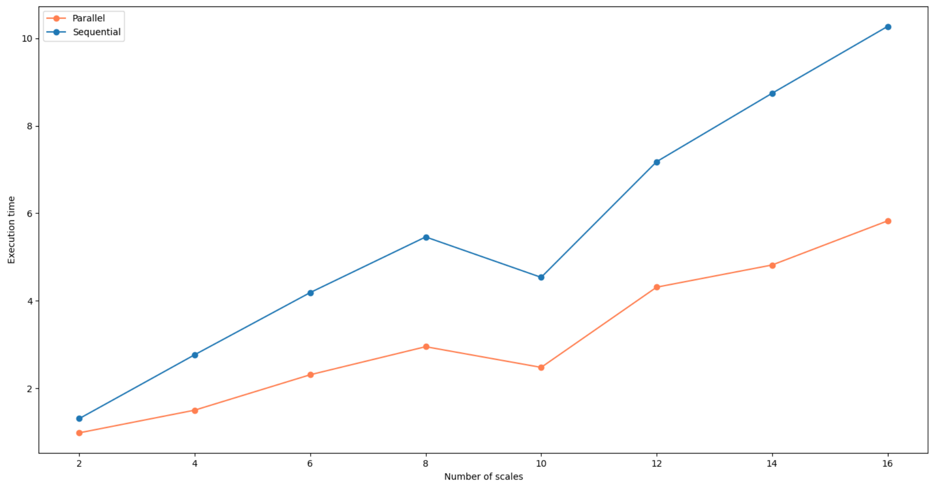

Figure 1. Execution time on different scales

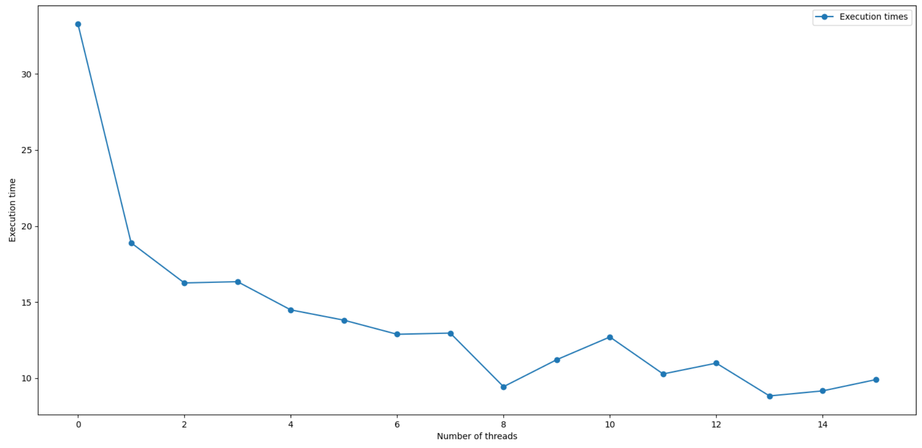

Figure 2. Execution time on different num of threads

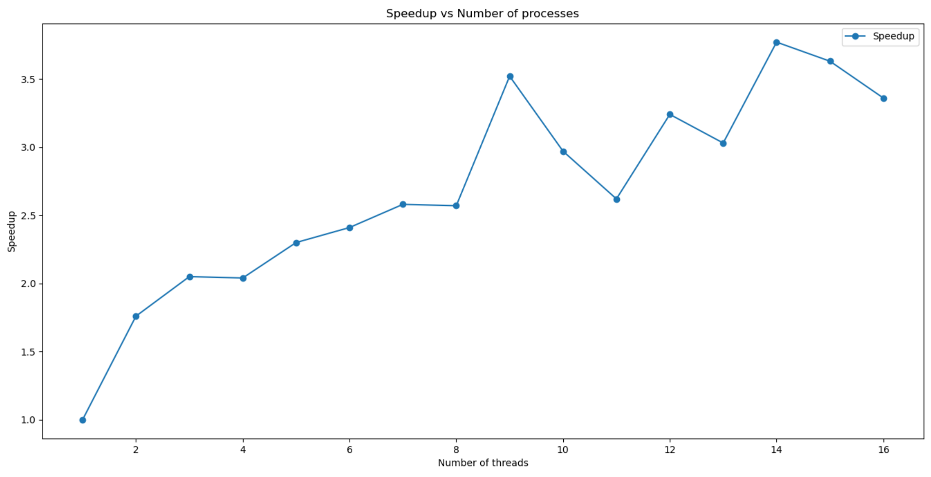

Figure 3. Num of threads vs Speedup