Alpha Composition

Parallel Computing

Luigi Ariano
luigi.ariano@stud.unifi.it

Edoardo Bonanni
edoardo.bonanni@stud.unifi.it

Master's Degree in Computer Engineering

Università degli Studi di Firenze

2022-2023

Abstract

The aim of this project is to calculate the alpha composition of a set of RGBA images. We propose **two sequential** and two **parallel implementations**, in order to measure how the parallel versions can improve the performances.

1 Introduction

Given a set of RGBA images as input, in this project, we implemented two sequential programs e two parallel ones in order to calculate the corresponding composite image: the first parallel version we propose uses **C++ OpenMP**, while the second one uses **Python Joblib**, as regards the two sequential versions, these are nothing more than implementations in different languages, C++ and Python, of the same algorithm. In this way, we can estimate how the parallel implementations can improve the performances, measured in terms of mean execution time and speedup.

2 Alpha Composition

Alpha composition makes it possible to draw layered graphics where the look of each layer can be controlled, which depends on the look of its underlying layer.

Blending combines a translucent upper layer with a lower layer producing a new blended color so, basically, the alpha channel of the upper layer sets the opacity of the layer itself.

If there are two images A and B, the alpha-composite of A over B, it will create an effect that allows to see both image A and B in the resulting image, as if A has transparency and through A is possible to see B. The over operator can be accomplished by applying the following formula to each pixel:

$$\alpha_o = \alpha_a + \alpha_b(1 - \alpha_a), \tag{1}$$

$$C_o = \frac{C_a\alpha_a + C_b\alpha_b(1 - \alpha_a)}{\alpha_o} \tag{2}$$

Here C_o, C_a and C_b stand for the color components of the pixels in the result, image A and image B respectively, applied to each color channel (red/green/blue) individually, whereas α_o, α_a and α_b are the alpha values of the respective pixels.

3 Implementation

In this section, we describe the sequential and parallel implementations of alpha composition. First, we will show the implementations in C++ (sequential and OpenMP) and then those in Python (sequential and Joblib). In the figure 1, all the input images used for composing the output image are shown.

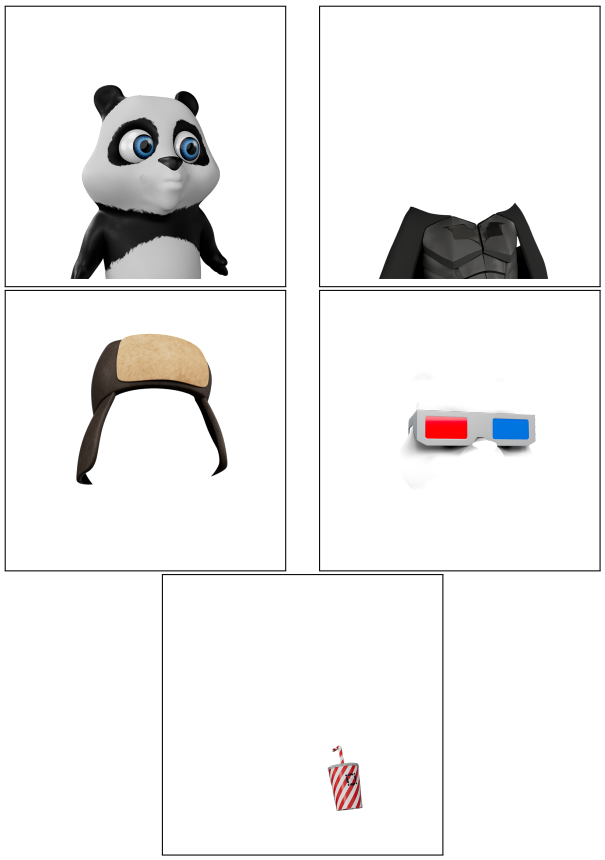

Figure 1: This figure shows the input images used for alpha composite to create a composite output image.

3.1 RGBAImage Class

```
float* RGBAImage::createFlatImage() {
    int size = getSize();
    float* flatImage = new float[size];
    for(int i=0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            Vec4b pixel = image.at<Vec4b>(i, j);
            flatImage[(i * width * numChannels) + (j *
                 numChannels)] = (float)pixel[2];
            flatImage[(i * width * numChannels) + (j *
                 numChannels) + 1] = (float)pixel[1];
            flatImage[(i * width * numChannels) + (j *
                 numChannels) + 2] = (float)pixel[0];
            flatImage[(i * width * numChannels) + (j *
                 numChannels) + 3] = (float)pixel[3];
        }
    }
    return flatImage;
}
```

We defined a class that allows to manage RGBA images. The **OpenCV** library is used to read the image by providing the path as input, to obtain its width, height, number of channels and size. To get an efficient implementation we decided to use the **flat image** as input for the alpha composition algorithm. This structure allows to easily and quickly access each channel related to a pixel

since, for each pixel, the RGBA values are stored consecutively in the array. In addition to the previous reasons, this structure allows to parallelize accesses to pixel values in order to simultaneously calculate these values for the resulting image via the formulas 1 and 2 shown before.

```
def createFlatImage(self):
    size = self.getSize()
    flatImage = np.zeros(shape=size)
    for i in range(0, self._height):
        for j in range(0, self._width):
            pixel = self._image[i][j]
            flatImage[(i * self._width * self._numChannels) + (j *
                 self._numChannels)] = float(pixel[2])
            flatImage[(i * self._width * self._numChannels) + (j *
                 self._numChannels) + 1] = float(pixel[1])
            flatImage[(i * self._width * self._numChannels) + (j *
                 self._numChannels) + 2] = float(pixel[0])
            flatImage[(i * self._width * self._numChannels) + (j *
                 self._numChannels) + 3] = float(pixel[3])
    return flatImage
```

3.2 C++ Sequential Implementation

```
void sequentialC++(int imgSize, vector<float*> flatImages){
    float* flatMixImages = new float[imgSize];
    for(int i = 0; i < imgSize; i++)
        flatMixImages[i] = flatImages[0][i];
    for (int i = 0; i < imgSize; i += 4) {
        for (int j = 1; j < flatImages.size(); j++) {
            float alphaA = flatImages[j][i + 3] / 255;
            float alphaB = flatMixImages[i + 3] / 255;
            float alphaComponent = alphaA + alphaB * (1 - alphaA);
            if(alphaComponent == 0){
                flatMixImages[i] = 0;
                flatMixImages[i + 1] = 0;
                flatMixImages[i + 2] = 0;
                flatMixImages[i + 3] = 0;
            } else{
                flatMixImages[i] = (flatImages[j][i] * alphaA +
                     flatMixImages[i] * alphaB * (1 - alphaA)) /
                     alphaComponent;
                flatMixImages[i + 1] = (flatImages[j][i + 1] *
                     alphaA + flatMixImages[i + 1] * alphaB * (1 -
                     alphaA)) / alphaComponent;
                flatMixImages[i + 2] = (flatImages[j][i + 2] *
                     alphaA + flatMixImages[i + 2] * alphaB * (1 -
                     alphaA)) / alphaComponent;
                float alphaComponentNew = (((alphaComponent - 0) *
                     255) / 1);
                flatMixImages[i + 3] = alphaComponentNew;
            }
        }
    }
    free(flatMixImages);
}
```

In all implementations, a fundamental input parameter is the RGBA images vector formed by the flat images ordered from the lowest to the highest level.

First, an array equal to the images size is created which will correspond to the composite image and it's initialized with the RGBA values of the lowest level image, in order to skip the alpha composition of first image. As the flat array is defined, a pixel is made up of 4 consecutive values that correspond to the RGBA channels and therefore, to scroll through all the pixels of the image, a for loop is performed increasing the value by 4 at each iteration until the end of the array. For each pixel, the composition operation is repeated for all the images, in particular, a single image is considered at a time with respect to the resulting one and the alpha composition operation is applied.

This operation (see formulas 1 and 2) calculates the normalized alpha values of the two images and subsequently the value of the RGB color components by inserting the result found in the correct position of the array. Once the described algorithm is finished, the resulting image will be formed from the result of the alpha composition of the images provided as input.

3.3 OpenMP Implementation

```
void parallelOpenMP(int nThreads, int imgSize, vector<float*>
     flatImages){
    for(int n_thread = 2; n_thread <= nThreads; n_thread+=2) {
        float* flatMixImages = new float[imgSize];
        #pragma omp parallel num_threads(n_thread) default(none)
             shared(imgSize, flatMixImages, flatImages)
        {
            #pragma omp for schedule(static) nowait
            for (int i = 0; i < imgSize; i += 4) {
                float r = 0;
                float g = 0;
                float b = 0;
                float a = 0;
                for (int j = 0; j < flatImages.size(); j++) {
                    float alphaA = flatImages[j][i + 3] / 255;
                    float alphaB = a / 255;
                    float alphaComponent = alphaA + alphaB * (1 -
                         alphaA);
                    if(alphaComponent == 0){
                        r = 0;
                        g = 0;
                        b = 0;
                        a = 0;
                    } else{
                        r = (flatImages[j][i] * alphaA + r * alphaB
                             * (1 - alphaA)) / alphaComponent;
                        g = (flatImages[j][i + 1] * alphaA + g *
                             alphaB * (1 - alphaA)) /
                             alphaComponent;
                        b = (flatImages[j][i + 2] * alphaA + b *
                             alphaB * (1 - alphaA)) /
                             alphaComponent;
                        a = (((alphaComponent - 0) * 255) / 1);
                    }

                }
                flatMixImages[i] = r;
                flatMixImages[i + 1] = g;
                flatMixImages[i + 2] = b;
                flatMixImages[i + 3] = a;
            }
        }
        free(flatMixImages);
    }
}
```

The OpenMP (**Open Multi-Processing**) API supports multi-platform shared-memory parallel programming in C/C++. It defines a portable, scalable model with a simple and flexible interface for developing parallel applications and it doesn't require restructuring the serial program, the use only needs to add compiler directives to reconstruct the serial program into a parallel one.

In this case, firstly it's necessary to create an array equal to the images size which will correspond to the composite image.

In order to better parallelize our code, we had to choose appropriate pragma omp directives based on our task. In fact, the execution must be parallelized based on the num-

ber of threads chosen and, since we have set *default(none)* which requires each variable in the construct to be specified explicitly as shared or private, it's important to define also which variables are shared between threads. For us, the shared variables are the vector containing the RGBA images, the resulting image and their size.

The section to be parallelized regards access to the pixels of the image, while guaranteeing the sequentiality of the operations necessary for the calculation of the RGBA values which will constitute the composite image.

Given the structure defined and the type of parallelization chosen, it is not necessary to define a critical section regarding the writing of the values in the resulting array, obtaining an output image correctly composed from the pixels of the input images.

Furthermore, for the previous reasons, the type of scheduling can be *static* as the distribution of the pixel values of the images is the same and not even needing synchronization between threads (*nowait* keyword) which access the various pixels thanks to the type of structure we created for the composite image.

3.4 Python Sequential Implementation

```
def sequentialPython(imgSize, flatImages):
    flatMixImages = np.zeros(shape=imgSize)
    for i in range(0, imgSize):
        flatMixImages[i] = flatImages[0][i]
    for i in range(0, imgSize, 4):
        for j in range(1, len(flatImages)):
            alphaA = flatImages[j][i + 3] / 255
            alphaB = flatMixImages[i + 3] / 255
            alpha_component = alphaA + alphaB * (1 - alphaA)
            if alpha_component == 0:
                flatMixImages[i] = 0
                flatMixImages[i + 1] = 0
                flatMixImages[i + 2] = 0
                flatMixImages[i + 3] = 0
            else:
                flatMixImages[i] = ((flatImages[j][i] * alphaA +
                     flatMixImages[i] * alphaB * (1 - alphaA)) /
                     alpha_component)
                flatMixImages[i + 1] = ((flatImages[j][i + 1] *
                     alphaA + flatMixImages[i + 1] * alphaB * (1 -
                     alphaA)) / alpha_component)
                flatMixImages[i + 2] = ((flatImages[j][i + 2] *
                     alphaA + flatMixImages[i + 2] * alphaB * (1 -
                     alphaA)) / alpha_component)
                flatMixImages[i + 3] = (((alpha_component - 0) *
                     255) / 1)
    del flatMixImages
```

For this implementation there is not much to say because it's almost the same as the sequential one written in C++. A note to make concerns the structure used to represent flat images: **NumPy** arrays which are preferable to Python lists since they occupy less memory for the same arrays size and are faster in reading and writing the items. The RGBAImage class was also created in Python to replicate the flat structure of the images in order to easily manage the parallel access to the pixels of these and maintain the same structure implemented in C++, using the same formulas described above to compute the pixel values of the output image.

3.5 Joblib Implementation

```
def parallelJoblib(nThreads, imgSize, flatImages):
    folder = "shared_mem/"
    if not os.path.exists(folder):
        os.makedirs(folder)
    output_filename_memmap = os.path.join(folder, 'flatMixImage')
    for n_thread in range(2, nThreads + 1, 2):
        chunk = int(imgSize / n_thread / 4)
        bounds = []
        for i in range(0, n_thread - 1):
            bounds.append((i * 4 * chunk, (i + 1) * 4 * chunk))
        bounds.append(((n_thread - 1) * 4 * chunk, imgSize))
        flatMixImage = np.memmap(output_filename_memmap,
             shape=imgSize, mode='w+')
        Parallel(n_jobs=n_thread, backend="loky", mmap_mode="r+")(
            delayed(create_mixed_image_parallel)
            (flatMixImage[bounds[i][0]:bounds[i][1]],
            [image[bounds[i][0]:bounds[i][1]] for image
            in flatImages]) for i in range(0, n_thread))
        del flatMixImage
    try:
        shutil.rmtree(folder)
    except:
        print('Could not clean-up automatically.')

def create_mixed_image_parallel(flatMixImages, flatImages):
    for i in range(0, len(flatMixImages), 4):
        r, g, b, a = 0, 0, 0, 0
        for j in range(0, len(flatImages)):
            alphaA = flatImages[j][i + 3] / 255
            alphaB = a / 255
            alpha_component = alphaA + alphaB * (1 - alphaA)
            if alpha_component == 0:
                r = 0
                g = 0
                b = 0
                a = 0
            else:
                r = (flatImages[j][i] * alphaA + r * alphaB * (1 -
                     alphaA)) / alpha_component
                g = (flatImages[j][i + 1] * alphaA + g * alphaB *
                     (1 - alphaA)) / alpha_component
                b = (flatImages[j][i + 2] * alphaA + b * alphaB *
                     (1 - alphaA)) / alpha_component
                a = (((alpha_component - 0) * 255) / 1)
        flatMixImages[i] = r
        flatMixImages[i + 1] = g
        flatMixImages[i + 2] = b
        flatMixImages[i + 3] = a
```

Joblib is a set of tools to provide lightweight pipelining in Python. In particular, it guarantees a simple parallel computing, is optimized to be fast and robust on large data and has specific optimizations for NumPy arrays.

Joblib needs a shared and temporary memory space in which to edit the resulting image.

This library performs the same function in a parallel way, so it is fundamental to split the image's pixel based on the number of threads in order to reduce the overhead. In this way the image is split efficiently reducing this problem, instead of considering each pixel individually and executing the function many times increasing the execution time **massively**.

Chunking is based on the number of threads which splits the pixels of the image ensuring that RGBA values are not separated between threads by this evaluation: int($imgSize/n\_thread/4$). The operations inside the parallelized function are performed sequentially on the basis of the pixel block considered, avoiding critical sections in writing the new values.

As in the previous case, the parallel access to the pixels of the input images allows to obtain a correctly composed

output image reducing the computational time compared to the sequential version.

Another aspect to consider for the Joblib version is the type of backend to choose for parallel code execution. The available backend types, which we also tested for final evaluation purposes, are as follows:

- *loky* backend is based on process-based parallelism. This module is used to launch separate Python worker processes to run tasks concurrently on separate CPUs, overcomes GIL limitation through shared memory programming paradigm. It can also induce a significant overhead as the input and output data need to be serialized in a queue for communication with the worker processes;
- *multiprocessing* previous process-based backend, but less robust and more overhead than *loky*;
- *threading* is based on thread-based parallelism. It is a very low-overhead backend but it suffers from the Python Global Interpreter Lock if the called function relies a lot on Python objects.

The difference obtained in the results that will be shown mainly concerns the first two backends mentioned compared to the last one as they are based on two types of parallelism: thread-based and process-based. The global interpreter lock (**GIL**) prevents more than one piece of Python bytecode from running concurrently. This means that for anything other than I/O bound tasks, excluding some small exceptions, using multithreading won't provide any performance benefits the way it would in C/C++ using OpenMP. Multiprocessing let to spawn processes instead of creating threads, so instead of having our parent process spawning threads to parallelize things, we spawn subprocesses to handle our work. Each subprocess we spawn will have its own Python interpreter and its own GIL but these subprocesses can run on different cores, obtaining parallelism.

4 Tests

All of the following tests are executed on a **laptop** computer with the following specifications:

- OS: Microsoft Windows 10;
- CPU: 12th Gen Intel(R) Core(TM) i7-12700H 2.70 GHz;
- GPU: NVIDIA GeForce RTX 3050 Ti Laptop;
- RAM: 16GB @3600 MHz.

Tests consider mean execution times which are calculated by repeating the computation of resulting image **100** times for each test image for every implementation proposed. In particular, the test images differs in size: we consider the same image scaled at different resolution 480p, 720p, 1080p, 2K and 4K both for width and height, since we use square images. For each resolution, the input images used have the same size as, regardless of the object present, the border pixels are transparent in order to have consistency in size.

Regarding the parallel approaches, we repeat these tests also changing the number of threads in order to effectively understand which is the best case in terms of performance.

The maximum number of threads chosen for parallel versions is equal to **20**, which is equal to the number of logical threads present in the used processor.

In the results, we expect to see a significant improvement in execution times up to 6 threads having a processor with 6 P-cores and slightly less up to 14 (8 E-cores), obtaining an asymptotic improvement up to 20 threads used.

5 Results

In this section we discuss about the results achieved by the tests. As expected, in both cases, the results achieved following the parallel approaches are significantly better than the sequential ones, moreover, the version developed in C++ is extremely faster than the one in Python. Regardless of the type of implementation used, sequential or parallel, and the type of language chosen, C++ or Python, in output we get a correctly composed image shown in figure 2.

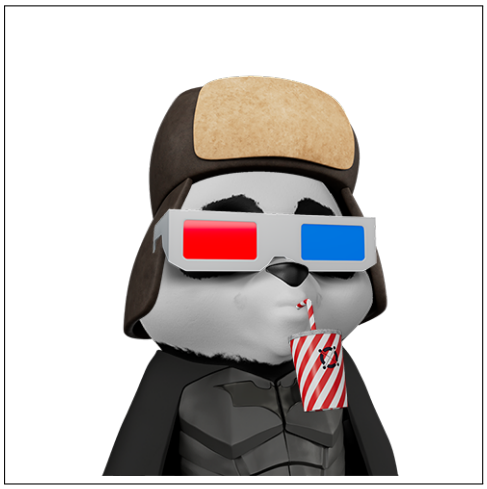

Figure 2: This figure shows the output image after alpha composition.

5.1 C++ Implementation Results

In figure 3 is shown the mean execution times obtained with OpenMP, repeating tests on images with different size and varying the threads' number. As we can notice, obviously, the bigger the image the more the execution time increases.

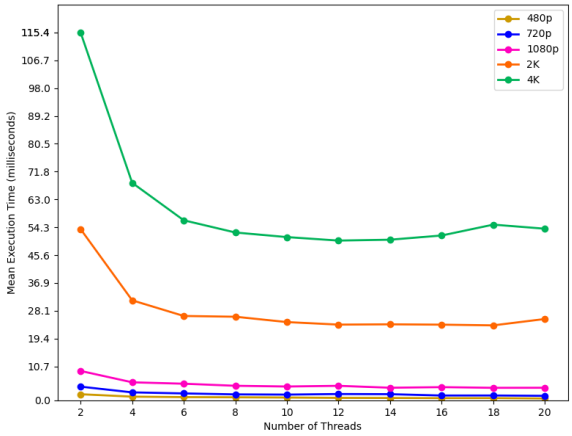

Figure 3: The mean execution times for the developed algorithm in OpenMP based on the number of threads and the size of the images are shown.

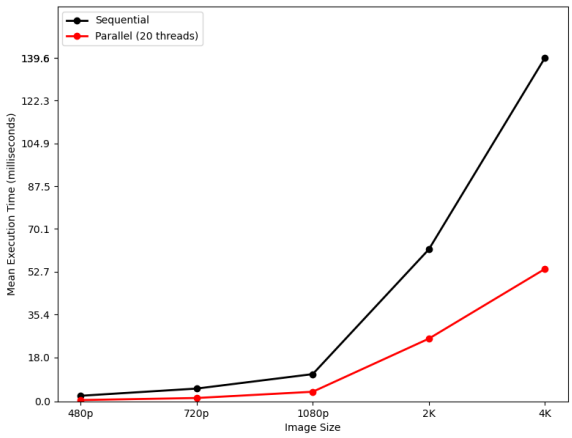

Figure 4: Comparison of execution times for each image size between the sequential and parallel approach. For parallel implementation, for each image size we considered the minimum time obtained varying threads' number.

Subsequently, in figure 4, we also compare the execution times obtained with the different image size, using both the sequential and the parallel approach. In particular, regarding the parallel implementation with OpenMP, in this graphic are represented the minimum execution times obtained from the previous test, varying the number of threads. We can see that we achieved with the parallel approach good improvements mostly with the biggest image and for 20 threads, as we expected.

Finally, figure 5 compares the different speedup obtained for each image size, varying the number of threads.

The speedup is defined as $S_P = t_S/t_P$, where P is the number of processors, t_S is the completion time of the sequential algorithm and t_P is the completion time of the parallel algorithm. Basically, the speedup is perfect to ideal if $S_P = P$.

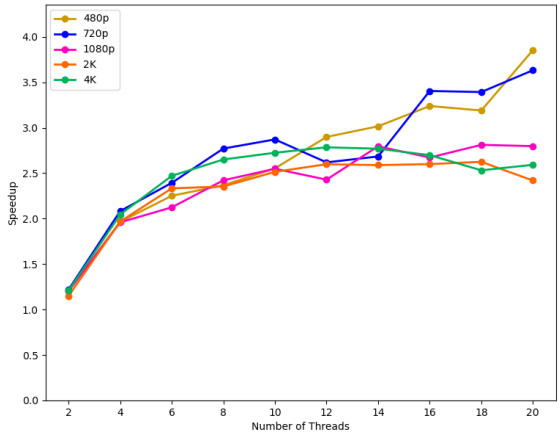

Figure 5: Comparison of speedup with OpenMP for different image size and varying threads' number.

We achieved the best results with the 480p image, using 20 threads: indeed in this case we obtained a 3.85 of speedup.

5.2 Python Implementation Results

Also for Python implementation, we show the mean execution times obtained with Joblib in figure 6, repeating tests on images with different size, varying the threads' number and backend type.

Another interesting aspect that we show in the following 7 figure concerns the mean execution times obtained by varying the size of the image for the 3 backends we tested, setting the number of threads equal to **20** with which we obtained the better execution times. What is interesting to note, and what we expected to happen, is that the best configuration found concerns the use of the *loky* backend, which uses multi-process parallelism and introduces less overhead than the other *multiprocessing* backend. The worst is obviously the *threading* since, being a multi-threaded process, it is limited by the GIL and we don't get a real improvement in terms of execution time even as the number of threads increases.

Finally, in figure 8, we also compare the mean execution times obtained with the different image size, using both the sequential and the best parallel approach (i.e. with *loky* backend). Also in this case, we achieved with the parallel approach good improvements mostly with the biggest image.

Also in this case, the speedup obtained is shown by the figure 9. We can see that, in general, we have obtained excellent results, in fact, all the speedups are very high

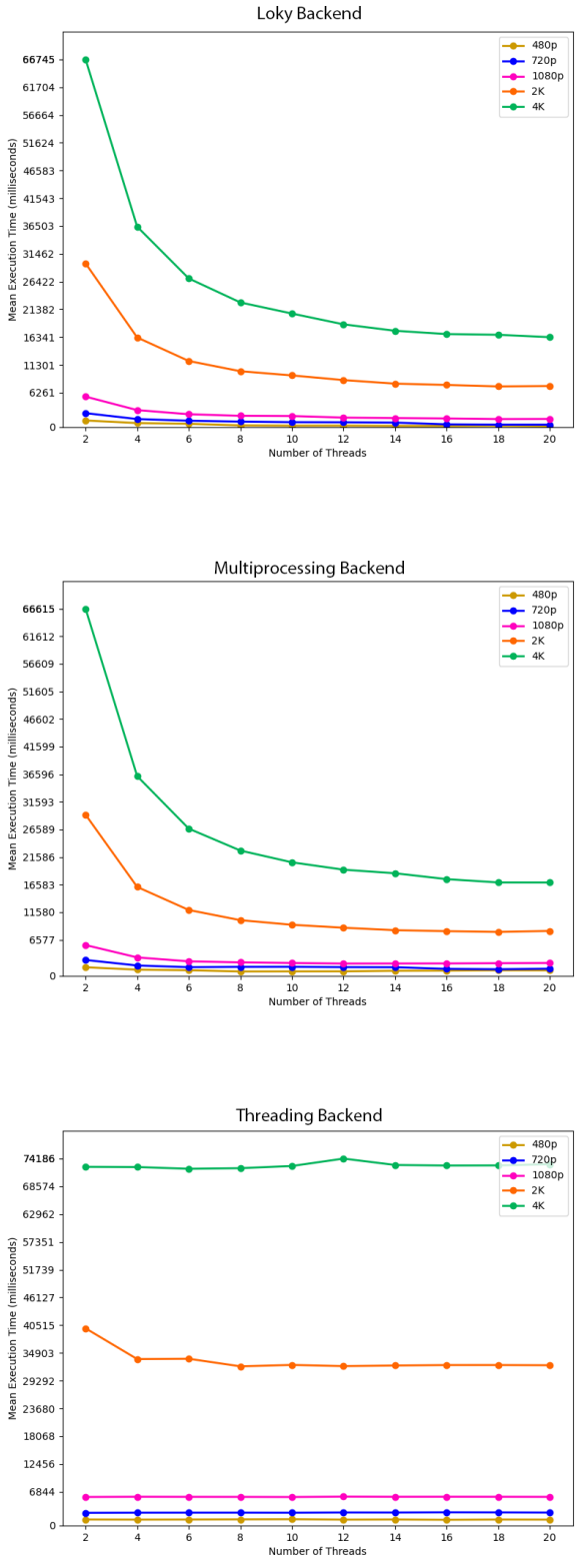

Figure 6: The mean execution times for the developed algorithm in Joblib based on the number of threads, the backed used and the size of the images are shown.

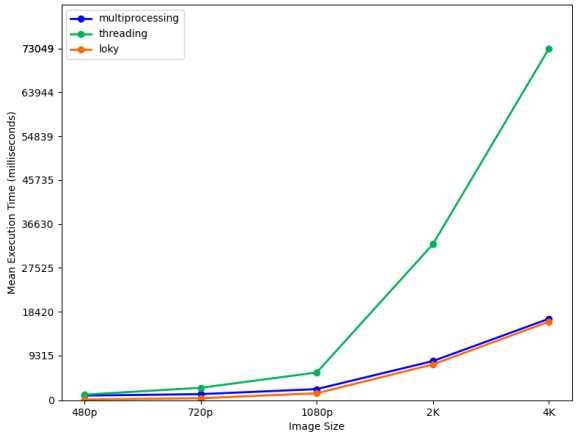

Figure 7: Comparison of execution times for each image size among all Joblib backend. The loky backend turns out to be the best, followed by multiprocessing. The threading backend is extremely worse than the other two in terms of performance.

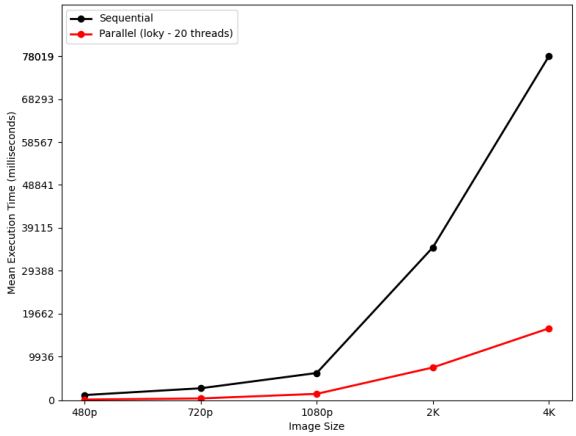

Figure 8: Comparison of execution times for each image size between the sequential and parallel approach. For parallel implementation, for each image size we considered the minimum time obtained varying threads' number and backend type.

with a significant improvement in performance, especially for the 720p image which has a value of 6.02 and, although much slower than the C++ version, the parallel Python version is much faster than its sequential counterpart.

If we had developed a version that divided the image into single pixels rather than chunks based on the number of threads, we would have obtained a slower parallel version than the sequential one precisely due to some communication and memory overhead when exchanging input and output data with the worker Python processes.

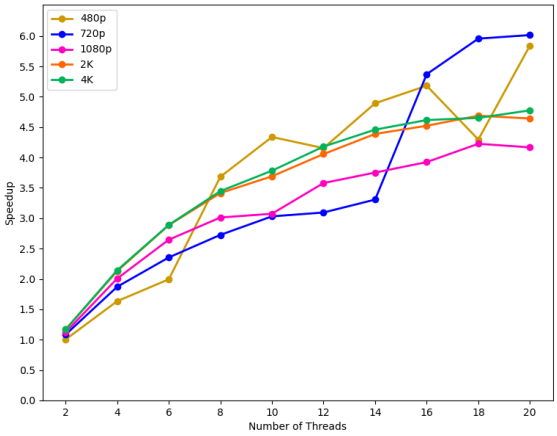

Figure 9: Comparison of speedup with Joblib loky backend for different image size and varying threads' number.

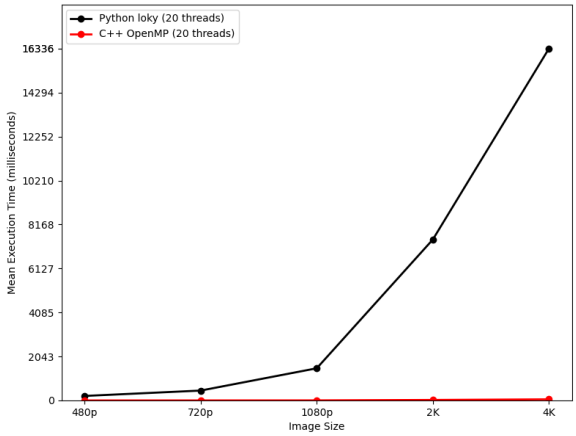

Figure 11: Comparison of execution times for each image size between the two parallel approaches: OpenMP and Joblib.

5.3 Comparison between C++ and Python implementations

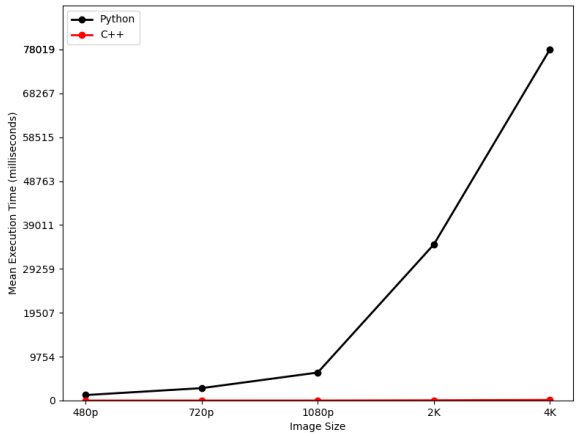

Figure 10: Comparison of execution times for each image size between the two sequential approaches: C++ and Python.

In the end, we also compare the sequential and parallel approaches proposed in this project.

As expected, due to the nature of the two languages, the mean execution time of the sequential version in C++ compared to the one in Python is significantly lower.

C++ is a *compiled* language, meaning a compiler program reads over all your source code all at once and translates it into machine code, something that the CPU can read and understand directly. This has the advantage of speed, but comes at the downside of a heavy initial cost (compiling is slow) as well as platform dependence.

Python is an *interpreted* language, meaning it requires a separate program (i.e. CPython) to read over your source code and actually perform the instructions it encodes. This has the advantage of allowing the code to potentially work on many different platforms, but will go through an interpretation stage before it can be compiled resulting in a performance hit.

Basically, for these reasons, there is no comparison for the execution times, with the limitations highlighted no doubt also reflecting for the parallel versions implemented in OpenMP and Joblib.

However, in the 10 and 11 figures, we show both the execution time differences for both the sequential versions in C++ and Python and for the parallel ones implemented with OpenMP and Joblib (always considering 20 threads per both and with backend set to *loky* for Joblib).

6 Conclusions

As to be expected, both parallel approaches dramatically improve run time performance. From the results obtained, the best approach obviously turned out to be the one in C++ since both the sequential and the parallel implementation gave better results than those in Python due to the limitations explained previously. If one were to consider the improvement achieved through the parallel approaches over their sequential counterparts, the version implemented in Python turns out to be more effective for this purpose.

In the end, due to the processor used, the execution time trend dropped drastically up to 6 threads, less up to 14 and we got an asymptotic improvement up to 20.