# Image Reader

Parallel Programming for Machine Learning

Luca Leuter

# Introduction

One of the most commonly performed operations on a PC is to read images and perform operations on them. This project focuses on the first part, following two approaches for reading JPEG images: a sequential and a parallel one

UNIVERSITÀ DEGLI STUDI FIRENZE

# Implementation details

- **OpenCV library** allows loading an image into an object of type Mat, which contains a matrix with the pixel values of each image
- **Chrono** used to keep track of time
- **Filesystem** to perform operations with system folders

UNIVERSITÀ DEGLI STUDI FIRENZE

# Sequential Approach

```
Mat* images;
void loadImagesSequential(vector<string>
    imageStrings){
   images = new Mat[imageStrings.size()];
   for(int i=0; i<imageStrings.size(); i++)
      images[i] = imread(imageStrings[i]);
}
```

```
Mat* getImagesSequential(){
   return images;
}
```

# Parallel Approach

```
Mat* images1;
void loadImagesParallel(vector<string>
    imgStrings){
#pragma omp parallel private()
   // Parallel loading of images1
   images1 = new Mat[imgStrings.size()];
#pragma omp for
   for(int i=0; i<imgStrings.size(); i++)
      images1[i] = imread(imageStrings[i]);
}
```

```
Mat* getImagesParallel(){
   return images;
}
```

UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI FIRENZE

# Setup

The tests were run on a PC with an AMD A8-6410 processor with 4 cores and 4 threads. The images are 1000 and have a resolution ranging from 720p to 1080p.

# Test 1

For the first test, the first 500 images of the dataset are considered. The test is a comparison between sequential and parallel version

# Test 1

```
int maxThreads = omp_get_max_threads();
for (auto &p:
    filesystem::directory_iterator(imagePath))
   imageStrings.emplace_back(p.path().string());
   vector<string> imageStringsSubset(
   imageStrings.begin(),imageStrings.begin()
       + subsetDimension);
   int sequentialTime = 0;
```

# Test 1

```
for (int i = 0; i < nTest; i++) {
   Mat *images;
   auto start =
       chrono::system_clock::now();
   loadImagesSequential(imageStringsSubset);
   images = getImagesSequential();
   auto end =
       std::chrono::system_clock::now();
   auto elapsed =
       chrono::duration_cast<chrono::milliseconds>(end
       - start);
   sequentialTime += elapsed.count();
   delete[] images;
}
logFile << "Tempo medio trascorso per il
    caricamento delle immagini in modo
    sequenziale:\n"
      << sequentialTime / nTest << "\n";
```

# Test 1

```
for(int k = 2; k < (4*maxThreads + 1);
    k++) {
    int parallelTime = 0;
    // Cambia il numero di thread
    omp_set_num_threads(k);
    // Caricamento parallelo delle immagini
    for (int i = 0; i < nTest; i++) {
        Mat *images;
        auto start =
            std::chrono::system_clock::now();
        loadImagesParallel(imageStringsSubset);
        images = getImagesParallel();
        auto end =
            std::chrono::system_clock::now();
        auto elapsed =
            std::chrono::duration_cast<std::chrono::millise
            - start);
        parallelTime += elapsed.count();
        delete[] images;
    }
    cout << "Tempo medio trascorso per il
        caricamento delle immagini in modo
        parallelo: "
        << parallelTime / nTest << "\n";
    logFile << parallelTime / nTest << "\n";
}
```

# Test 2

For the second test a subset of images is considered each time until all the images are loaded. The test start with 400 images and increase by 50 each time until the last one

# Test 2

```
for (int i =
     (int)imageStrings.size()/2.5; i <
     imageStrings.size(); i+=50) {
    Mat *images;
    sequentialTime = 0;
    vector<string>
        imageStringsTest(imageStrings.begin(
        imageStrings.begin() + i);
    auto start =
        chrono::system_clock::now();
    loadImagesSequential(imageStringsTest);
    images = getImagesSequential();
    auto end =
        chrono::system_clock::now()
    auto elapsed =
        chrono::duration_cast<chrono::millis
        (end - start);
    sequentialTime += elapsed.count();
    delete[] images;
    cout << "Tempo trascorso per il
        caricamento di " << i << "
        immagini in modo sequenziale: "
        << sequentialTime << "\n";
    logFile2 << sequentialTime << "\n";
}
```

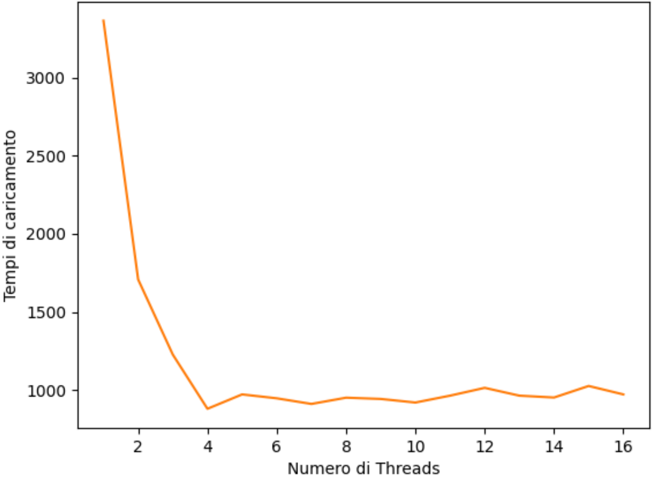
# Test 2

```
omp_set_num_threads(maxThreads);
for (int i =
    (int)imageStrings.size()/2.5; i <
    imageStrings.size(); i+=50) {
   int parallelTime = 0;
   Mat *images;
   vector<string>
       imageStringsTest(imageStrings.begin(),
       imageStrings.begin() + i);
   auto start =
       chrono::system_clock::now();
   loadImagesParallel(imageStringsTest);
   images = getImagesParallel();
 auto end =
     chrono::system_clock::now();
 auto elapsed =
     chrono::duration_cast<chrono::milliseconds>
    (end - start);
 parallelTime += elapsed.count();
 delete[] images;
 logFile2 << parallelTime<< "\n";
}
```

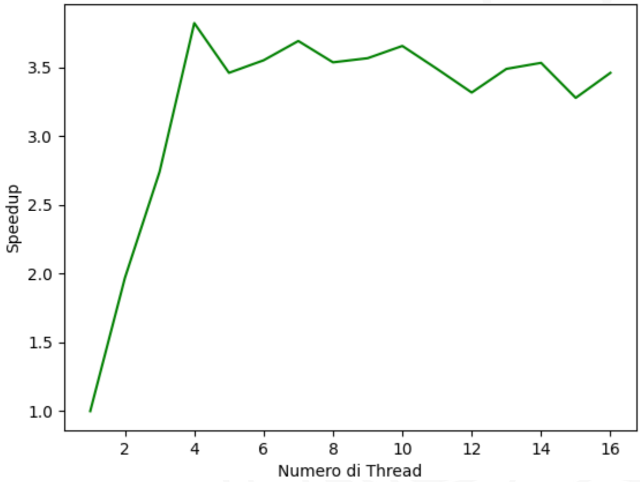
UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI FIRENZE

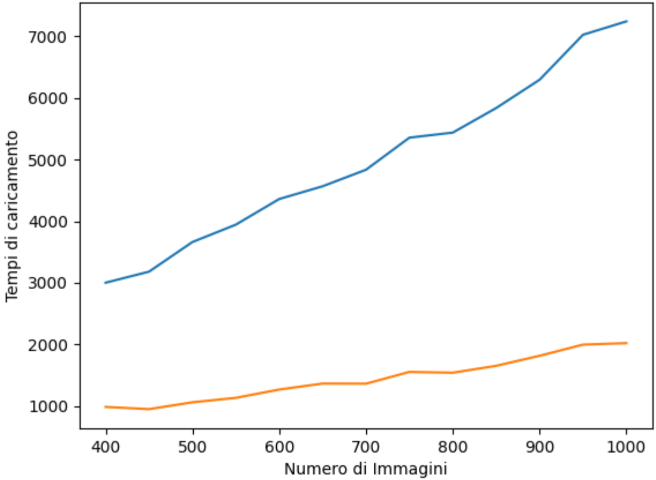
# Results of test 1



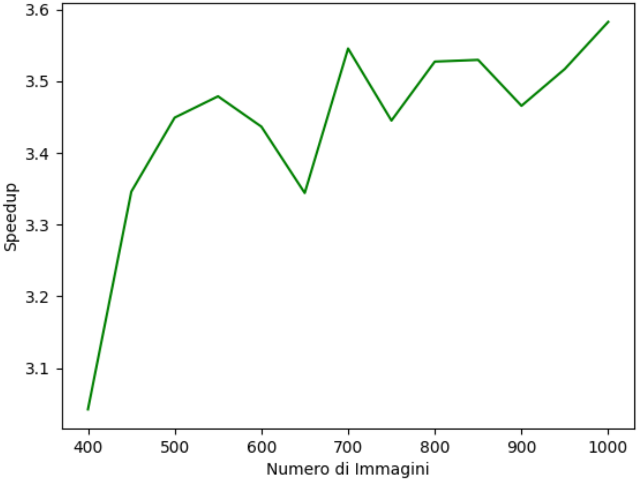
# Results of test 1



# Results of test 2



# Results of test 2



# Conclusion

From the results of the tests, it was found that increasing the number of threads beyond the limit does not lead to an improvement, therefore it is always better to keep them at the maximum possible for the machine. Furthermore, it has also been seen that increasing the number of images does not necessarily result in an improvement in speedup.