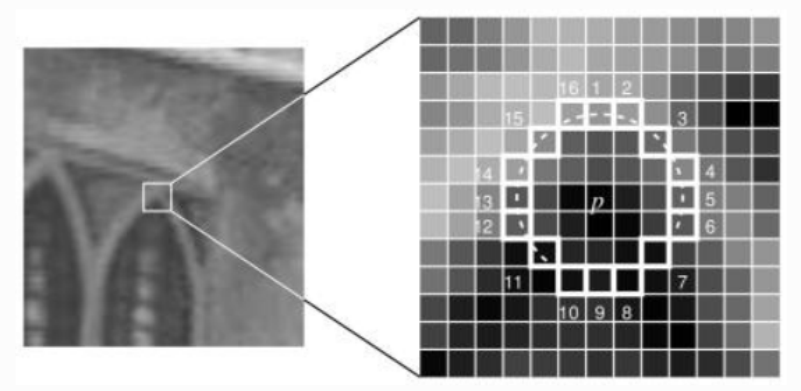
4.2 Back to the 21st century! Using FAST and ORB

The Harris corner detector has been employed in a vast amount of works since its proposal back in 1988 and it's still being used in many approaches because of its simplicity and performance. However, the computer vision community has obviously moved on in recent years and nowadays is being focused in using faster methods to detect corners. One of the most successful and popular methods for detecting corners is called **FAST** (**F**eatures from **A**ccelerated **S**egment **T**est), which, has its name suggest is clearly faster than most of the methods developed so far. Published in 2006, the work by Rosten and Drummond claims to operate around 20x faster than the Harris method and finds a considerably large amount of keypoints (sometimes too much!).

In a nutshell, in its original form, it operates by comparing the grey level of a certain pixel in the image with a surrounding circle of 16 pixels. If at least 12 consecutive pixels in that circle are brighter (or darker) than the candidate, then it is a considered as a corner. Being based just in pixel comparisons, you can imagine its speed!



In fact, by wisely selecting the first 4 pixels to compare, fast rejection of possible candidates can be easily applied. Some variations of the original proposal have been developed later, turning the FAST-based approaches a prominent method for detecting corners nowadays.

However, this method **does not provide a descriptor** for the detected corners, so, as we learnt before, they must be augmented with a descriptor in order to be matched! We could use again NCC and a patch, but in this case we are going to explore the ORB method, which is a detection and description method developed in 2011 by Rublee *et.al*.

ORB stands for **O**riented FAST and **R**otated **B**RIEF and combines the FAST detector with a modified version of the BRIEF descriptor. In short, ORB operates as follows:

1. It detects FAST corners and computes its main orientation.

- It rotates the sourrounding patch of the keypoint according to the main orientation.
- It computes the BRIEF descriptor by comparing the grey level in a set of wisely selected pairs of pixels within the (rotated) patch, yielding a **binary** sequence that

corresponds to the descriptor.

Since the resulting descriptors are binary (i.e. sequences of bits), they can be easily compared by using the so-called Hamming distance, which, essentially, computes the number of different bits in the two descriptors.

In [1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams['figure.figsize'] = (20.0, 20.0)
images_path = './images/'
```

ASSIGNMENT 1: Using ORB

Now let's try a simple example of using ORB. Write a script that:

1. Loads the images 'park_l.jpeg' and 'park_r.jpeg' in grayscale.

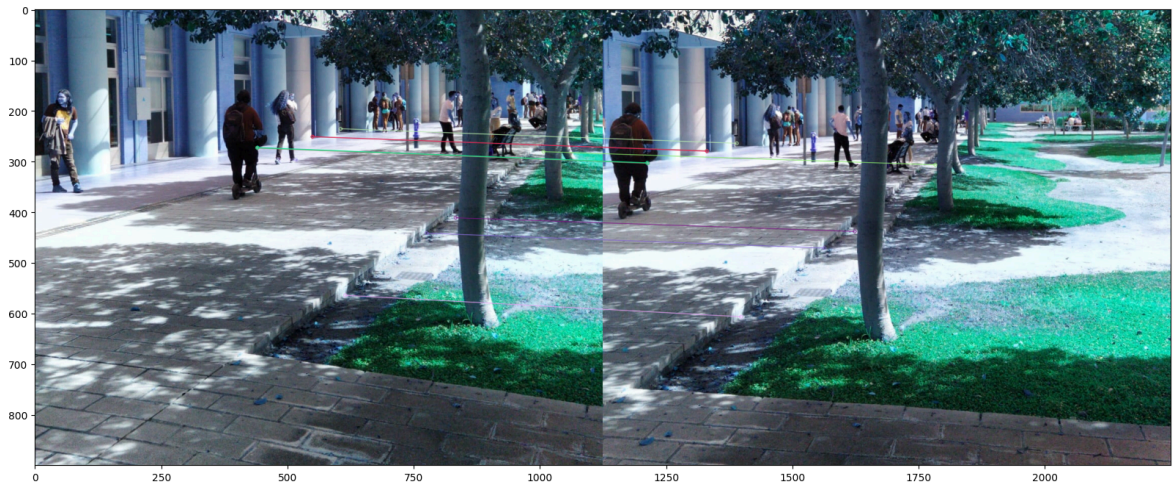
- Detect ORB keypoints.
- Compute their ORB descriptor.
- Use a Brute-Force matcher to find correspondences between both sets of keypoints. A Brute-Force matcher simply compares a certain descriptor in a list with all the rest of descriptors in an exhaustive search.
- Order the matches according to their distance (have a look to `sorted()` and use as key the lambda function `x:x.distance`). Set the `crossCheck` argument to `True` in order to get more robust matches.
- Once all matches are defined, call `cv2.drawMatches` and display the resulting image with the 30 best matches. Try `cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS` as flag in this function call.

Tip: Search for some documentation regarding ORB detection/description and brute-force matching in OpenCV.

In [2]:
```
# Assignment 1
# Detect and describe keypoints using ORB in two images ('park_l.jpeg' an
# then match them using a brute-force matcher.
# Write your code here

# Load the two previous images and convert to grayscale
img1 = cv2.imread(images_path + 'park_l.jpeg')
img2 = cv2.imread(images_path + 'park_r.jpeg')

gra1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gra2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# Detect the ORB keypoints using the OpenCV method
# -- create the ORB detector
orb = cv2.ORB_create()

# Create a brute-force matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

```
# -- detect ORB keypoints
kp1, desc1 = orb.detectAndCompute(gra1, None)
kp2, desc2 = orb.detectAndCompute(gra2, None)


# -- compute the descriptors with ORB


# Note: detection and description can be done in just one call

# Match descriptors.
# -- match the descriptors using the BF matcher
matches = bf.match(desc1, desc2)


# Sort them in the order of their distance.
# -- sort the matches
matches = sorted(matches, key = lambda x:x.distance)


# Display both images side-by-side along with the matches
# -- create a new image that contains both images
img3 = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None, flags=cv


# And finally show them
plt.imshow(img3)
```

Out[2]: <matplotlib.figure.AxesImage at 0x78e4dd5827e0>



Thinking about it

Now you can compare this output with the one produced by Harris + NCC...

- **What could you conclude?**

  The ORB method is much faster and easier to use than the Harris + NCC method.

Conclusion

This was a short but intense notebook covering state-of-the-art keypoint detection and description techniques such as FAST and ORB. It is exciting to stay informed of modern techniques!