Orthomosaic

In [1]:
```
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving IX-11-01917_0004_0004.JPG to IX-11-01917_0004_0004.JPG
Saving IX-11-01917_0004_0005.JPG to IX-11-01917_0004_0005.JPG
```

In [12]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import imageio
import imutils
cv2.ocl.setUseOpenCL(False)
```

In [15]:
```
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving IX-11-01917_0004_0001.JPG to IX-11-01917_0004_0001.JPG
Saving IX-11-01917_0004_0002.JPG to IX-11-01917_0004_0002.JPG
```

In [28]:
```
trainImg = imageio.imread('IX-11-01917_0004_0004.JPG')
trainImg_gray = cv2.cvtColor(trainImg, cv2.COLOR_RGB2GRAY)

Img = imageio.imread('IX-11-01917_0004_0005.JPG')
# Opencv defines the color channel in the order BGR.
# Transform it to RGB to be compatible to matplotlib
Img_gray = cv2.cvtColor(Img, cv2.COLOR_RGB2GRAY)

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, constrained_layout=False, figsize=(16,9))
ax1.imshow(Img, cmap="gray")
ax1.set_xlabel("image", fontsize=14)
```

```python
ax2.imshow(trainImg, cmap="gray")
ax2.set_xlabel("image (Image to be transformed)", fontsize=14)

plt.show()
```

In [31]:

```python
def detectAndDescribe(image, method=None):
    """
    Compute key points and feature descriptors using an specific method
    """

    assert method is not None, "You need to define a feature detection method. Values are: 'sift', 'surf'"

    # detect and extract features from the image
    if method == 'sift':
        descriptor = cv2.xfeatures2d.SIFT_create()
    elif method == 'surf':
        descriptor = cv2.xfeatures2d.SURF_create()
    elif method == 'brisk':
        descriptor = cv2.BRISK_create()
    elif method == 'orb':
        descriptor = cv2.ORB_create()
```

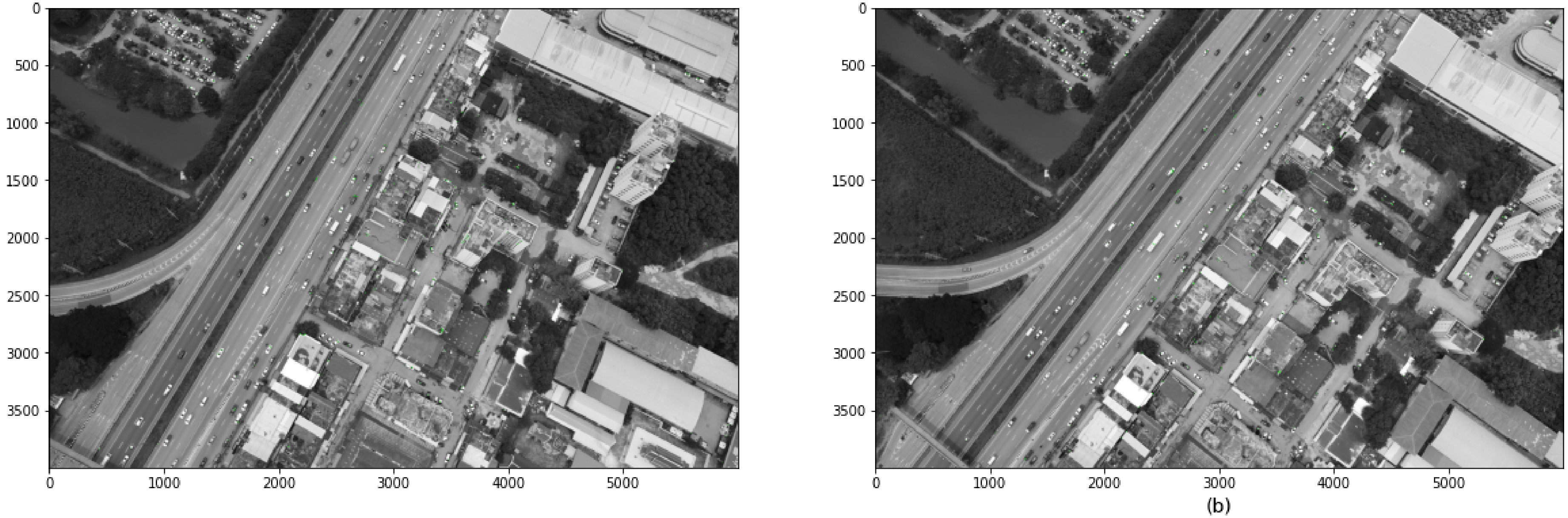
```
    # get keypoints and descriptors
    (kps, features) = descriptor.detectAndCompute(image, None)

    return (kps, features)
```

In [32]:
```
kpsA, featuresA = detectAndDescribe(trainImg_gray, method=feature_extractor)
kpsB, featuresB = detectAndDescribe(queryImg_gray, method=feature_extractor)
```

In [33]:
```
# display the keypoints and features detected on both images
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(20,8), constrained_layout=False)
ax1.imshow(cv2.drawKeypoints(trainImg_gray,kpsA,None,color=(0,255,0)))
ax1.set_xlabel("", fontsize=14)
ax2.imshow(cv2.drawKeypoints(Img_gray,kpsB,None,color=(0,255,0)))
ax2.set_xlabel("(b)", fontsize=14)

plt.show()
```



In [34]:
```
def createMatcher(method,crossCheck):
    "Create and return a Matcher Object"
```

```
    if method == 'sift' or method == 'surf':
        bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=crossCheck)
    elif method == 'orb' or method == 'brisk':
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=crossCheck)
    return bf
```

In [35]:

```
def matchKeyPointsBF(featuresA, featuresB, method):
    bf = createMatcher(method, crossCheck=True)

    # Match descriptors.
    best_matches = bf.match(featuresA,featuresB)

    # Sort the features in order of distance.
    # The points with small distance (more similarity) are ordered first in the vector
    rawMatches = sorted(best_matches, key = lambda x:x.distance)
    print("Raw matches (Brute force):", len(rawMatches))
    return rawMatches
```

In [36]:

```
def matchKeyPointsKNN(featuresA, featuresB, ratio, method):
    bf = createMatcher(method, crossCheck=False)
    # compute the raw matches and initialize the list of actual matches
    rawMatches = bf.knnMatch(featuresA, featuresB, 2)
    print("Raw matches (knn):", len(rawMatches))
    matches = []

    # loop over the raw matches
    for m,n in rawMatches:
        # ensure the distance is within a certain ratio of each
        # other (i.e. Lowe's ratio test)
        if m.distance < n.distance * ratio:
            matches.append(m)
    return matches
```

In [37]:

```
print("Using: {} feature matcher".format(feature_matching))

fig = plt.figure(figsize=(20,8))

if feature_matching == 'bf':
    matches = matchKeyPointsBF(featuresA, featuresB, method=feature_extractor)
    img3 = cv2.drawMatches(trainImg,kpsA,Img,kpsB,matches[:100],
                           None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
```

```python
    elif feature_matching == 'knn':
        matches = matchKeyPointsKNN(featuresA, featuresB, ratio=0.75, method=feature_extractor)
        img3 = cv2.drawMatches(trainImg,kpsA,Img,kpsB,np.random.choice(matches,100),
                               None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)


    plt.imshow(img3)
    plt.show()
```

```
Using: bf feature matcher
Raw matches (Brute force): 197
```

In [39]:

```python
def getHomography(kpsA, kpsB, featuresA, featuresB, matches, reprojThresh):
    # convert the keypoints to numpy arrays
    kpsA = np.float32([kp.pt for kp in kpsA])
    kpsB = np.float32([kp.pt for kp in kpsB])

    if len(matches) > 4:

        # construct the two sets of points
        ptsA = np.float32([kpsA[m.queryIdx] for m in matches])
        ptsB = np.float32([kpsB[m.trainIdx] for m in matches])
```

```
        # estimate the homography between the sets of points
        (H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,
            reprojThresh)

        return (matches, H, status)
    else:
        return None
```

In [40]:
```
M = getHomography(kpsA, kpsB, featuresA, featuresB, matches, reprojThresh=4)
if M is None:
    print("Error!")
(matches, H, status) = M
print(H)
```

```
[[ 9.13030472e-01 -8.67489570e-02  5.63909415e+02]
 [ 7.57272314e-02  1.06018417e+00 -1.99196684e+02]
 [-2.46414289e-05  1.99958718e-05  1.00000000e+00]]
```

In [41]:
```
# Apply panorama correction
width = trainImg.shape[1] + Img.shape[1]
height = trainImg.shape[0] + Img.shape[0]

result = cv2.warpPerspective(trainImg, H, (width, height))
result[0:Img.shape[0], 0:Img.shape[1]] = Img

plt.figure(figsize=(20,10))
plt.imshow(result)

plt.axis('off')
plt.show()
```

In [42]:
```
# transform the panorama image to grayscale and threshold it
gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY)[1]

# Finds contours from the binary image
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```python
cnts = imutils.grab_contours(cnts)

# get the maximum contour area
c = max(cnts, key=cv2.contourArea)

# get a bbox from the contour area
(x, y, w, h) = cv2.boundingRect(c)

# crop the image to the bbox coordinates
result = result[y:y + h, x:x + w]

# show the cropped image
plt.figure(figsize=(20,10))
plt.imshow(result)
```

Out[42]: <matplotlib.image.AxesImage at 0x7fd135c20bd0>

References:
https://www.kaggle.com/deepzsenu/orthomapping-and-image-stitching