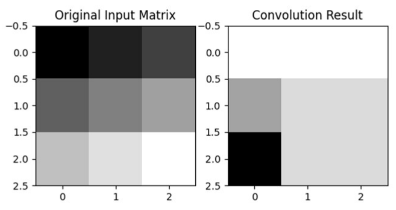
1.1) (Code is attached in zip file but will copy and past it here as well)

```
def Convolution(input_matrix, kernel_matrix):
    input_height, input_width = input_matrix.shape
    kernel_height, kernel_width = kernel_matrix.shape
    # Get the dimensions of the input matrix and the kernel
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2
    # Calculate the padding needed for convolution to keep the output size the same
    padded_input = np.pad(input_matrix, ((pad_height, pad_height), (pad_width, pad_width),),
mode='constant') # Pad the input matrix with zeros
    result = np.zeros_like(input_matrix, dtype=np.float32) # Create a result matrix of the same size
as the input_matrix with all values initialized to zero
    for i in range(input_height):
        for j in range(input_width): # Loop through each pixel
            region = padded_input[i:i + kernel_height, j:j + kernel_width]# Extract a region
            result[i, j] = np.sum(region * kernel_matrix)# Perform the convolution operation
    return result
```

1.2) I have tested the Convulation function with a random input and random custom kernel
(code is in zip file but below is copy and pasted)

```
input_matrix = np.array([[3, 6, 9],[12, 15, 18],[21, 24, 27]], dtype=np.float32) # Create a 3x3
sample input matrix
custom_kernel = np.array([[1, -1],[2, -2]], dtype=np.float32) # Create a 2x2 sample kernel
convolution_result = Convolution(input_matrix, custom_kernel) # Perform the convolution

plt.subplot(1, 2, 1)
plt.imshow(input_matrix, cmap='gray')
plt.title('Original Input Matrix') # Create subplot for the original input matrix

plt.subplot(1, 2, 2)
plt.imshow(convolution_result, cmap='gray')
plt.title('Convolution Result') # Create subplot for the convolution result

plt.show()
```

The result of this was the following:



1.3)

My implementation employs nested loops to iterate through each pixel in the input image, applying the convolution operation with a specified kernel. To ensure accurate convolution along the image borders, zero-padding is incorporated. However, this approach may lack the optimization found in built-in OpenCV functions, especially for large images.

In contrast, the OpenCV implementation stands out for its high level of optimization, resulting in significantly enhanced speed compared to my implementation. OpenCV also offers the flexibility to specify the border handling strategy, allowing for precise control over how border pixels are managed during convolution.

Both implementations produce similar results, however due to potential differences in optimization there might be a slight numerical variation. The OpenCV implementation is expected to be significantly faster, especially for larger images or kernels. Therefore, for small-scale operations, a manual implementation such as mine is sufficient. But for real-time/production-level applications, libraries like OpenCV is recommended where performance is critical.

TASK 2

2.1)

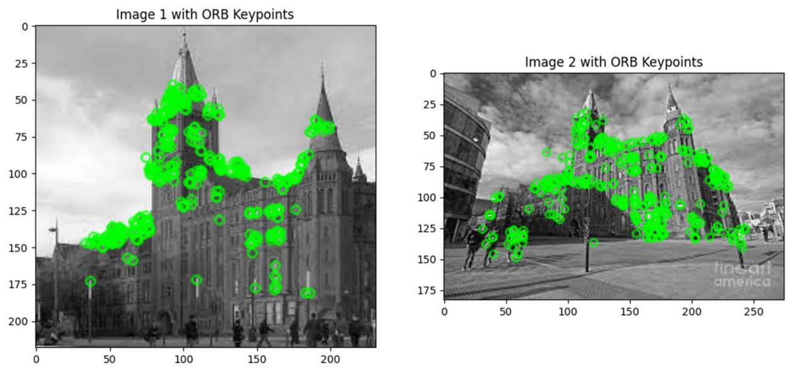
SURF employs a 64-dimensional vector as its feature descriptor, while SIFT generates a more extensive 128-dimensional descriptor for each key point. On the other hand, ORB stands out by utilizing concise binary descriptors.

In achieving scale and rotation invariance, the SURF algorithm leverages integral images and Haar wavelet responses, enabling efficient feature detection across various scales and orientations. SIFT, on the other hand, employs Gaussian blurring and incorporates orientation histograms to ensure robustness in the face of rotations. Meanwhile, ORB achieves it through a pyramid of blurred images and capitalizing on the orientation information computed during the FAST corner detection process.

When considering speed and efficiency, ORB emerges as the fastest among the three algorithms. This is attributed to its integration of the FAST corner detector and BRIEF descriptor. Following closely in terms of speed is SURF, leveraging integral images and clever approximations to accelerate the computation of feature descriptors. On the other hand, while SIFT is renowned for its robustness and accuracy, it tends to be computationally expensive.

SURF excels in real-time applications demanding high speed, making it a preferred choice for tasks like object recognition and tracking. SIFT, on the other hand, has gained extensive utilization across diverse computer vision applications, ranging from image stitching to 3D reconstruction. ORB stands out as a fitting solution for real-time scenarios where speed and accuracy are essential, such as in applications like robot navigation and augmented reality.

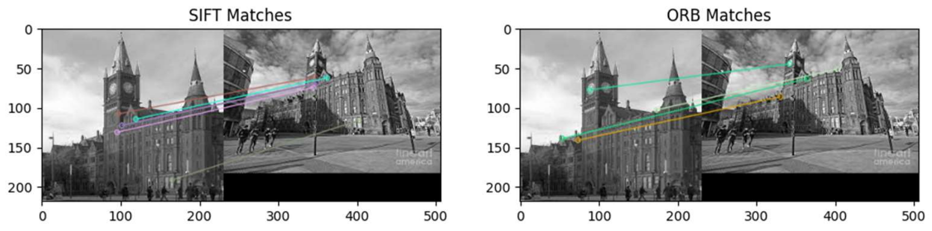
2.2) Image attached below is the detected points in both images, victoria1.jpg and victoria2.jpg.



Code from the result above is below:

```
image_path1 = r'C:\Users\akter\OneDrive\Documents\Uni\Year 3\Comp338\A1\victoria1.jpg'

image_path2 = r'C:\Users\akter\OneDrive\Documents\Uni\Year 3\Comp338\A1\victoria2.jpg'


img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)

img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)


orb = cv2.ORB_create()


keypoints1, descriptors1 = orb.detectAndCompute(img1, None)

keypoints2, descriptors2 = orb.detectAndCompute(img2, None)


# Visualize keypoints

image1_with_keypoints = cv2.drawKeypoints(img1, keypoints1, None, color=(0, 255, 0), flags=0)

image2_with_keypoints = cv2.drawKeypoints(img2, keypoints2, None, color=(0, 255, 0), flags=0)


# Display the images

plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)

plt.imshow(image1_with_keypoints, cmap='gray')
plt.title('Image 1 with ORB Keypoints')


plt.subplot(1, 2, 2)

plt.imshow(image2_with_keypoints, cmap='gray')

plt.title('Image 2 with ORB Keypoints')


plt.show()
```

2.3)Result:



SIFT is known for its accuracy and robustness in feature matching. It computes a 128-dimensional descriptor for each keypoint, providing rich information for matching. ORB, on the other hand, uses binary descriptors, which are faster to compute and require less memory. However, they may be less distinctive than SIFT descriptors.

I believe SIFT is considered more effective in capturing the unique of the images more than ORB as it matches exhibit more distinctively and accurately and the matches are well-distributes and coherent. However, ORB may have been favoured if it was application where speed was an essential, since in the results it was only one match, it was an acceptable compromise in accuracy.

Code for the result:

```
image_path1 = r'C:\Users\akter\OneDrive\Documents\Uni\Year 3\Comp338\A1\victoria1.jpg'

image_path2 = r'C:\Users\akter\OneDrive\Documents\Uni\Year 3\Comp338\A1\victoria2.jpg'
```

```
img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

# Initialize SIFT and ORB detectors
sift_detector = cv2.SIFT_create()
orb_detector = cv2.ORB_create()

# Detect keypoints and compute descriptors for SIFT
keypoints1_sift, descriptors1_sift = sift_detector.detectAndCompute(img1, None)
keypoints2_sift, descriptors2_sift = sift_detector.detectAndCompute(img2, None)

# Detect keypoints and compute descriptors for ORB
keypoints1_orb, descriptors1_orb = orb_detector.detectAndCompute(img1, None)
keypoints2_orb, descriptors2_orb = orb_detector.detectAndCompute(img2, None)

# Initialize Brute-Force Matcher
bf_matcher = cv2.BFMatcher()

# Match descriptors using Brute-Force Matcher for SIFT
matches_sift = bf_matcher.knnMatch(descriptors1_sift, descriptors2_sift, k=2)

# Match descriptors using Brute-Force Matcher for ORB
matches_orb = bf_matcher.knnMatch(descriptors1_orb, descriptors2_orb, k=2)

# Apply ratio test for SIFT matches
good_matches_sift = []
for match1, match2 in matches_sift:
  if match1.distance < 0.75 * match2.distance:
    good_matches_sift.append(match1)

# Apply ratio test for ORB matches
```

```
good_matches_orb = []

for match1, match2 in matches_orb:

    if match1.distance < 0.75 * match2.distance:

        good_matches_orb.append(match1)


# Visualize the matches

img_matches_sift = cv2.drawMatches(img1, keypoints1_sift, img2, keypoints2_sift,
good_matches_sift, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

img_matches_orb = cv2.drawMatches(img1, keypoints1_orb, img2, keypoints2_orb,
good_matches_orb, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)


# Display the results

plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)

plt.imshow(img_matches_sift)

plt.title('SIFT Matches')


plt.subplot(1, 2, 2)

plt.imshow(img_matches_orb)

plt.title('ORB Matches')


plt.show()
```