ChatGPT

Giải Thích Mã Nguồn Ghép Ảnh Panorama (SIFT & ORB)

Bài viết này trình bày chi tiết **toàn bộ mã nguồn từ Cell 1 đến Cell 7** trong Jupyter Notebook cho bài tập lớn về ghép ảnh panorama. Chương trình được chia làm hai phần:

- **Phần 1 (Cell 1-5):** Sử dụng **SIFT** (Scale-Invariant Feature Transform) để phát hiện và ghép đặc trưng (đây là phần chính, đúng với yêu cầu chủ đề 5).
- **Phần 2 (Cell 6-7):** Sử dụng **ORB** (Oriented FAST and Rotated BRIEF) để phát hiện và ghép đặc trưng (phần mở rộng nhằm so sánh với SIFT).

Tài liệu sẽ đi theo thứ tự thực thi các cell, giải thích rõ chức năng của từng dòng lệnh quan trọng cũng như các khái niệm xử lý ảnh liên quan: **keypoint**, **descriptor**, **SIFT**, **ORB**, **BFMatcher** (Brute-Force Matcher), **homography**, **RANSAC**, **chồng ảnh & blending**, **warpPerspective**, v.v. Mỗi phần sẽ có trích dẫn code và giải thích chi tiết như một hướng dẫn lab.

Phần 1: Panorama với SIFT (Cell 1 đến Cell 5)

Phần này hiện thực quy trình ghép ảnh panorama dựa trên **đặc trưng SIFT** – một thuật toán trích xuất điểm đặc trưng bền vững với thay đổi kích thước và xoay ảnh. Tóm tắt các bước chính trong phần này:

1. **Đọc ảnh đầu vào:** Nạp hai ảnh cần ghép và (nếu cần) chuyển sang xám để xử lý.
2. **Phát hiện đặc trưng SIFT:** Sử dụng SIFT để tìm các **keypoint** (điểm đặc trưng) và mô tả chúng bằng **descriptor** 128 chiều [1].
3. **Ghép cặp đặc trưng (Feature Matching):** Dùng **BFMatcher** (Brute-Force) để so sánh descriptor giữa hai ảnh và tìm các cặp keypoint tương ứng dựa trên khoảng cách Euclid nhỏ nhất [2].
4. **Tính toán homography với RANSAC:** Từ các cặp điểm tương ứng, ước lượng ma trận homography (3×3) bằng thuật toán **RANSAC** để liên hệ phép biến đổi hình học giữa hai ảnh [3] [4].
5. **Warp và ghép ảnh:** Sử dụng homography để biến đổi (warp) một ảnh vào hệ tọa độ ảnh kia (tạo ảnh chung). Sau đó chồng ảnh và **blend** (trộn) vùng chồng lấp để tạo panorama hoàn chỉnh.

Dưới đây, ta sẽ đi qua từng cell với mã nguồn và giải thích chi tiết.

Cell 1: Import thư viện và đọc ảnh đầu vào

Mã lệnh (Cell 1):

```
import cv2
import numpy as np

# Đọc hai ảnh đầu vào cần ghép panorama
img1 = cv2.imread('image1.jpg')  # Ảnh thứ nhất
img2 = cv2.imread('image2.jpg')  # Ảnh thứ hai
```

```
# Chuyển ảnh sang thang xám (để xử lý SIFT hiệu quả hơn)
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
```

Giải thích:
- `import cv2` và `import numpy as np`: Import thư viện OpenCV (cv2) và NumPy. OpenCV cung cấp các hàm xử lý ảnh, NumPy để xử lý ma trận ảnh hiệu quả.
- `cv2.imread('image1.jpg')`: Đọc ảnh từ file **image1.jpg**. Ảnh được nạp vào biến `img1` dưới dạng ma trận điểm ảnh (mảng NumPy 3 chiều BGR). Tương tự cho `img2`.
- **Chuyển ảnh sang grayscale:** SIFT hoạt động trên ảnh xám để tập trung vào cấu trúc cường độ sáng. Lệnh `cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)` chuyển ảnh từ không gian màu BGR sang mức xám (1 kênh). Việc dùng ảnh xám giúp tăng tốc và độ chính xác khi trích xuất đặc trưng (vì SIFT dựa trên gradient cường độ). Biến `gray1`, `gray2` là ảnh xám của hai ảnh đầu vào.

> **Lưu ý:** SIFT không yêu cầu thủ công chuyển sang xám (hàm SIFT trong OpenCV có thể tự chuyển nội bộ). Tuy nhiên, chuyển trước sang grayscale là thực hành tốt để đảm bảo hàm nhận dữ liệu đúng định dạng. Mặt khác, OpenCV đọc ảnh mặc định ở kênh màu BGR; nếu hiển thị bằng Matplotlib, cần chuyển sang RGB, nhưng trong code này chúng ta tập trung vào xử lý, chưa vẽ ảnh.

Cell 2: Khởi tạo SIFT và phát hiện keypoint, descriptor

Mã lệnh (Cell 2):

```
# Khởi tạo bộ phát hiện SIFT
sift = cv2.SIFT_create()  # hoặc cv2.xfeatures2d.SIFT_create() nếu dùng
OpenCV cũ

# Phát hiện keypoints và tính descriptors cho ảnh 1
keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
# Phát hiện keypoints và tính descriptors cho ảnh 2
keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

print("Ảnh 1 - Số lượng keypoint:", len(keypoints1))
print("Ảnh 2 - Số lượng keypoint:", len(keypoints2))
print("Descriptor của ảnh 1 có kích thước:", descriptors1.shape)
print("Descriptor của ảnh 2 có kích thước:", descriptors2.shape)
```

Giải thích:
- `cv2.SIFT_create()`: Tạo một đối tượng **SIFT** để dùng cho việc phát hiện đặc trưng. SIFT (Scale-Invariant Feature Transform) là thuật toán phát hiện các điểm nổi bật (**keypoint**) trên ảnh và mô tả chúng bằng vector đặc trưng (**descriptor**) 128 chiều [1]. Các **keypoint** SIFT thường là các điểm như góc hoặc vùng có texture đặc biệt, bền vững với thay đổi về kích thước, góc nhìn, độ sáng. *(Nói cách khác, cùng một điểm vật lý xuất hiện trên hai ảnh chụp khác nhau sẽ cho descriptor SIFT tương tự nhau, nhờ đó có thể nhận dạng là cùng một điểm [5].)*

  - `sift.detectAndCompute(gray1, None)`: Hàm này **phát hiện** keypoint và **tính toán** descriptor trên ảnh `gray1`. Tham số thứ hai (`None`) nghĩa là không sử dụng mặt nạ (mask), tức xét toàn bộ ảnh. Kết quả trả về:

- `keypoints1`: Danh sách các keypoint tìm được trên ảnh 1. Mỗi phần tử chứa thông tin như tọa độ (`kp.pt` là (x, y)), kích thước, hướng, v.v.
- `descriptors1`: Ma trận NumPy chứa descriptor cho mỗi keypoint, kích thước (N×128) với N là số keypoint. Mỗi descriptor là vector 128 float mô tả vùng lân cận quanh keypoint đó. Tương tự, `sift.detectAndCompute(gray2, None)` tính ra `keypoints2` và `descriptors2` cho ảnh 2.
- In thông tin: các lệnh `print` để kiểm tra số lượng keypoint tìm thấy và kích thước descriptor. **Ví dụ:** "Ảnh 1 - Số lượng keypoint: X" cho biết SIFT tìm được X điểm đặc trưng trên ảnh 1. Kích thước `descriptors1.shape` sẽ là `(X, 128)` xác nhận mỗi keypoint có descriptor 128 chiều.

Giải thích thêm về SIFT: Mỗi **keypoint SIFT** được xác định thông qua thuật toán tìm cực đại trên không gian khác nhau của ảnh (DoG - Difference of Gaussians). SIFT tạo ra một tập keypoint kèm orientation (hướng) để đạt **bất biến quay**, và multi-scale để đạt **bất biến tỷ lệ (scale)**. Descriptor 128 chiều được hình thành bằng cách tính histogram gradient trong lân cận 16×16 quanh keypoint [6]. Các vector descriptor này được thiết kế sao cho **ít bị ảnh hưởng bởi xoay, độ sáng, zoom** [7].

> **Ghi chú:** OpenCV 4.x tích hợp SIFT trong `cv2.SIFT_create()` (thuộc gói *opencv-contrib*). Nếu dùng bản cũ hoặc chưa cài opencv-contrib, phải dùng `cv2.xfeatures2d.SIFT_create()`. SIFT từng có ràng buộc bản quyền (đến 2020) nên các bản OpenCV cũ không hỗ trợ sẵn.

- Sau khi bước này, ta đã có hai tập điểm đặc trưng `keypoints1` và `keypoints2` của hai ảnh, cùng các descriptor tương ứng (`descriptors1`, `descriptors2`). Tiếp theo, ta cần **so khớp** các đặc trưng giữa hai ảnh.

Cell 3: So khớp đặc trưng giữa hai ảnh bằng BFMatcher

Mã lệnh (Cell 3):

```
# Tạo bộ ghép đặc trưng Brute-Force Matcher
bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

# Thực hiện ghép các descriptor của hai ảnh
matches = bf.match(descriptors1, descriptors2)

# Sắp xếp các match theo khoảng cách (từ nhỏ đến lớn)
matches = sorted(matches, key=lambda m: m.distance)

print("Tổng số cặp match tìm được:", len(matches))
# Lấy ra 5 match đầu tiên (nhỏ nhất) để in minh họa
for i in range(5):
    m = matches[i]
    print(f"Match {i}: Image1 KP {m.queryIdx} <-> Image2 KP {m.trainIdx},
Dist={m.distance:.2f}")
```

Giải thích:
- `cv2.BFMatcher(normType, crossCheck)`: Tạo bộ so khớp **Brute-Force Matcher** – thuật toán **vét cạn** so sánh mọi descriptor của ảnh 1 với mọi descriptor của ảnh 2 để tìm cặp gần nhất. Tham số

`cv2.NORM_L2` chỉ ra cách đo khoảng cách giữa hai descriptor: ở đây dùng **chuẩn L2 (khoảng cách Euclid)**, phù hợp cho SIFT (vì SIFT descriptors là vector float) [2]. - `crossCheck=True` yêu cầu mỗi cặp match phải thỏa mãn tính chất song phương: keypoint A của ảnh 1 khớp với B của ảnh 2 và ngược lại B cũng khớp với A. Điều này giúp lọc bớt các match không chắc chắn (tăng độ chính xác) nhưng có thể giảm số lượng match. (Nếu `crossCheck=False`, BFMatcher sẽ tìm mỗi điểm ảnh 1 cặp gần nhất ở ảnh 2, và ngược lại có thể khác; crossCheck giữ lại những cặp trùng nhau.)

- `matches = bf.match(descriptors1, descriptors2)`: Hàm `match` tìm **match tốt nhất** cho *mỗi* descriptor của ảnh 1 so với tập descriptor ảnh 2. Kết quả trả về danh sách `matches`, trong đó mỗi phần tử là đối tượng `cv2.DMatch` chứa: `queryIdx` (chỉ số keypoint ở ảnh 1), `trainIdx` (chỉ số keypoint tương ứng ở ảnh 2), và `distance` (khoảng cách nhỏ nhất tìm được giữa hai descriptor).

- Mỗi match đại diện cho một cặp keypoint có đặc trưng tương tự nhau giữa hai ảnh.

- `matches = sorted(matches, key=lambda m: m.distance)`: Sắp xếp danh sách các match tăng dần theo khoảng cách (distance). **Khoảng cách descriptor** nhỏ nghĩa là hai điểm khá giống nhau, do đó match đó đáng tin cậy hơn. Việc sắp xếp cho phép ta dễ dàng lấy các match tốt nhất trước.

- In thông tin: `len(matches)` cho biết tổng số match tìm được. Với crossCheck=True, số match thường $\leq$ số keypoint của ảnh nhỏ hơn. Đoạn code cũng in ra 5 match đầu tiên, chỉ rõ chỉ số keypoint tương ứng ở ảnh 1 và ảnh 2, cùng khoảng cách. Ví dụ: “Match 0: Image1 KP 35 <-> Image2 KP 47, Dist=128.45” nghĩa là keypoint thứ 35 của ảnh 1 ghép với keypoint thứ 47 của ảnh 2, khoảng cách descriptor ~128.45. (Chúng ta kỳ vọng các match đầu có distance khá nhỏ so với các match sau).

Giải thích thêm về Brute-Force Matching:
Phương pháp BFMatcher thực hiện so sánh trực tiếp **mọi cặp descriptor** giữa hai ảnh. Cụ thể, với mỗi descriptor của ảnh 1, nó tính khoảng cách đến **từng** descriptor của ảnh 2 và chọn cặp có khoảng cách nhỏ nhất làm match [2]. Đây là cách tiếp cận đơn giản nhưng chi phí tính toán cao (O(N*M) phép tính nếu ảnh 1 có N điểm, ảnh 2 có M điểm). Tuy nhiên, do số keypoint thường không quá lớn (vài nghìn), BFMatcher vẫn hoạt động được trong thời gian chấp nhận được.

Có hai kỹ thuật thường dùng để cải thiện kết quả match: - **Lowe’s ratio test:** Lấy 2 match tốt nhất cho mỗi keypoint ảnh 1 (k = 2 lân cận gần nhất) và kiểm tra tỷ lệ khoảng cách. Nếu khoảng cách match tốt nhất nhỏ hơn 0.75 lần khoảng cách match thứ hai, thì mới nhận match đó (nhằm loại bỏ trường hợp một điểm có hai đối tượng tương tự ngang nhau) [8]. (Phương pháp này cần dùng `bf.knnMatch` thay vì `bf.match`). - **Cross-check (song phương):** như ta đang dùng, chỉ nhận cặp match khi nó là tốt nhất theo cả hai chiều.

Trong code này, chúng ta dùng `crossCheck=True` (đơn giản hơn ratio test) để đảm bảo độ tin cậy của các cặp điểm tương ứng.

Sau bước này, ta có danh sách các cặp **keypoint tương ứng** giữa hai ảnh (`matches`). Tuy nhiên, không phải tất cả các match đều chính xác; có thể có **outlier** (cặp ghép sai, do nhiễu hoặc vùng lặp). Để ghép ảnh, ta cần một phép biến đổi gắn hai ảnh lại – đó là **homography**, và ta sẽ tính nó một cách **bền vững với outlier** bằng RANSAC.

Cell 4: Tính homography bằng RANSAC từ các điểm match

Mã lệnh (Cell 4):

```
# Trích xuất toạ độ của các điểm keypoint match tương ứng
src_pts = np.float32([ keypoints1[m.queryIdx].pt for m in
matches ]).reshape(-1, 1, 2)
dst_pts = np.float32([ keypoints2[m.trainIdx].pt for m in
matches ]).reshape(-1, 1, 2)

# Tính toán homography sử dụng RANSAC
H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

print("Ma trận homography H:")
print(H)
print("Số lượng inliers:", mask.sum(), "/", mask.size)
```

Giải thích:
- **Trích xuất tọa độ các điểm match:**
Ở bước này, ta cần chuẩn bị dữ liệu để tính homography. Mỗi match cung cấp chỉ số keypoint tương ứng giữa ảnh 1 và ảnh 2. Ta tạo hai mảng `src_pts` và `dst_pts` chứa tọa độ (x,y) của các cặp điểm này: - `keypoints1[m.queryIdx].pt` lấy tọa độ (float) của keypoint tương ứng ở ảnh 1. - `keypoints2[m.trainIdx].pt` lấy tọa độ của keypoint tương ứng ở ảnh 2. Sử dụng list comprehension để thu thập tất cả cặp điểm match. - `np.float32([...]).reshape(-1,1,2)`: Chuyển danh sách điểm về NumPy array kiểu float32 với shape `(N,1,2)` (đây là định dạng yêu cầu cho hàm findHomography: N điểm nguồn và N điểm đích, mỗi điểm 1x2).
Kết quả: `src_pts` là tập tọa độ các điểm ở ảnh 1, `dst_pts` là tọa độ các điểm tương ứng ở ảnh 2 (N là số match).

- `cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)`: Hàm này tính toán **ma trận homography H** (3×3) để biến đổi tọa độ từ **ảnh nguồn** (`src_pts`) sang **ảnh đích** (`dst_pts`).
- Tham số `cv2.RANSAC` chỉ định sử dụng thuật toán **RANSAC (Random Sample Consensus)** để ước lượng homography **chống nhiễu/outlier**.
- Tham số tiếp theo `5.0` là ngưỡng (tính bằng pixel) cho RANSAC – điểm nào sau khi biến đổi lệch quá 5 pixels so với vị trí tương ứng được xem là outlier và loại bỏ trong quá trình tính.

Homography là phép biến đổi phối cảnh (projective transform) mô tả bằng ma trận H 3×3, thích hợp khi hai ảnh chụp cùng một mặt phẳng hoặc có thể coi như trên một mặt phẳng (ví dụ cảnh distant hoặc camera quay tương đối ít). Homography cho phép ánh xạ mỗi điểm của ảnh này sang điểm tương ứng ở ảnh kia [9]. Với **4 cặp điểm tương ứng** không thẳng hàng ta có thể xác định duy nhất một homography. Ở đây ta có nhiều cặp điểm (hàng trăm), nên dùng RANSAC để tìm homography tốt nhất.

Nguyên tắc RANSAC: Lấy ngẫu nhiên một số bộ 4 cặp điểm, tính homography, rồi kiểm tra bao nhiêu cặp điểm khác ăn khớp (trong ngưỡng sai số) với homography đó. Lặp lại nhiều lần và chọn homography cho số inlier nhiều nhất [3] [4]. Như vậy RANSAC sẽ: - Loại bỏ các match sai (outlier) khỏi quá trình (chúng sẽ không nằm trong tập inlier của homography tốt nhất). - Trả về `H` là homography

ước lượng từ tập inlier tốt nhất, và `mask` là mảng kích thước N (số match) với giá trị 1 cho inlier, 0 cho outlier (mask chính là kết quả từ RANSAC).

- Kết quả:
- `H` (3x3): ma trận homography tính được. Ví dụ in ra có thể thấy dạng:

```
[[ 7.7466e-01  2.9793e-02  4.4857e+02]
 [-1.3163e-01  9.1080e-01  7.6323e+01]
 [-2.0330e-04 -3.3323e-05  1.0000e+00]]
```

  Các phần tử này mang ý nghĩa phép biến đổi phối cảnh (bao gồm xoay, zoom, dịch và biến dạng) từ ảnh 1 sang ảnh 2.

- `mask`: mặt nạ inliers/outliers. `mask.sum()` cho biết bao nhiêu cặp điểm trong số các match ban đầu được coi là inlier (khớp với H). Tỷ lệ inlier cao nghĩa là homography khớp tốt với đa số match (ảnh ghép phẳng hoặc overlap tốt). Nếu tỷ lệ thấp, có thể ảnh không phẳng hoặc match nhiều nhiễu.

- In thông tin: In ra ma trận H và số lượng inliers. Ví dụ: "Số lượng inliers: 120/150" nghĩa là trong 150 match ban đầu, 120 điểm được RANSAC xem là khớp tốt với homography tìm được (30 điểm kia là outlier bị loại bỏ).

Giải thích thêm về homography và RANSAC:
Homography là một **ma trận 3×3** (gồm 8 thông số độc lập, chuẩn hóa phần tử [2,2] = 1) mô tả phép biến đổi projective:

$$[x', y', w']^T = H \cdot [x, y, 1]^T,$$

sau đó tọa độ mới là $(x'/w',\ y'/w')$. Homography có thể biểu diễn chuỗi phép: xoay, tịnh tiến, co giãn, phối cảnh... để căn chỉnh hai ảnh.

Trong ghép ảnh panorama, ta thường coi **ảnh 1 là gốc**, tìm homography để biến đổi ảnh 2 trùng với ảnh 1. (Hoặc ngược lại, tùy cách ghép – ở đây `findHomography(src_pts, dst_pts)` lấy điểm ảnh1 sang ảnh2, tí nữa ta sẽ warp ảnh 2 về ảnh 1).

RANSAC đảm bảo rằng homography không bị lệch bởi những match sai. Thuật toán RANSAC cơ bản: 1. Chọn ngẫu nhiên 4 cặp match, tính homography. 2. Áp dụng homography này lên **tất cả** src_pts, đếm số điểm đích khớp (trong ngưỡng cho trước) với dst_pts. 3. Lặp nhiều lần (đến khi đủ độ tin cậy hoặc hết lượt), chọn homography có số inlier cao nhất [3]. 4. (Tùy chọn) Tính lại homography cuối cùng dựa trên toàn bộ inlier set.

Kết quả, ta có homography tối ưu và danh sách inlier (mask). Những inlier này là các điểm thực sự ăn khớp hình học với phép biến đổi – tức nằm trên cùng mặt phẳng cảnh.

Đến đây, về mặt toán học ta đã biết cách ghép hai ảnh: sử dụng homography H để biến đổi ảnh 2 sang phối cảnh của ảnh 1. Bước tiếp theo, ta sẽ **áp dụng homography để warp ảnh**.

Cell 5: Warp ảnh và trộn (blend) vùng chồng lắp để tạo panorama

Mã lệnh (Cell 5):

```
# Lấy kích thước của hai ảnh
h1, w1 = img1.shape[:2]
h2, w2 = img2.shape[:2]

# Tính kích thước canvas cho ảnh panorama (đủ chứa hai ảnh)
width_panorama = w1 + w2        # chiều rộng đủ chứa ảnh 1 và ảnh 2 sau warp
height_panorama = max(h1, h2)   # chiều cao lấy ảnh cao hơn

# Áp dụng phép biến đổi phối cảnh (warp) ảnh thứ 2 theo homography H
warped_img2 = cv2.warpPerspective(img2, H, (width_panorama, height_panorama))

# Chồng ảnh 1 lên ảnh đã warp
panorama = warped_img2.copy()
panorama[0:h1, 0:w1] = img1

# Lưu hoặc hiển thị ảnh panorama kết quả
cv2.imwrite('panorama_result.jpg', panorama)
# cv2.imshow('Panorama', panorama); cv2.waitKey(0); cv2.destroyAllWindows()
# nếu chạy local
```

Giải thích:
- **Kích thước ảnh và canvas:**
Trước tiên, chương trình lấy kích thước hai ảnh gốc: - `h1, w1`: chiều cao và rộng của ảnh 1. - `h2, w2`: chiều cao và rộng của ảnh 2.
Sau đó, tính **kích thước của ảnh panorama** sẽ tạo: - `width_panorama = w1 + w2`: Chọn bề rộng đủ lớn để chứa cả ảnh 1 và ảnh 2 sau khi warp. Cách làm đơn giản ở đây là cộng chiều rộng hai ảnh (ví dụ ảnh 1 bên trái, ảnh 2 warp rồi đặt sang phải). Thực tế, width cần lớn hơn nếu ảnh 2 warp sang trái hoặc phải nhiều, nhưng công thức tổng w1+w2 đảm bảo không thiếu. - `height_panorama = max(h1, h2)`: Chiều cao lấy lớn nhất của hai ảnh (trường hợp hai ảnh có cao khác nhau, panorama ít nhất phải cao bằng ảnh cao hơn). Cách chọn này cũng đơn giản và an toàn để không cắt mất nội dung.

Lưu ý: Xác định kích thước canvas (nền) cho panorama rất quan trọng. Ở đây ta ước lượng thô: cho ảnh 2 warp trải ra bên phải ảnh 1. Với trường hợp ghép theo chiều ngang (như panorama phong cảnh), chiều rộng cộng lại thường đủ. Nếu ảnh warp ra cả hai bên, có thể cần cộng thêm khoảng offset, nhưng trong ví dụ này ta giả định overlap tương đối và ảnh 2 nằm bên phải ảnh 1.

- **Warp ảnh 2 theo homography:**
  `cv2.warpPerspective(img2, H, (width_panorama, height_panorama))` áp dụng phép biến đổi phối cảnh cho ảnh 2.
- Ảnh nguồn: `img2`.
- Ma trận biến đổi: `H` (đã tính ở Cell 4, ánh xạ từ ảnh1 sang ảnh2). Thực tế, để warp ảnh2 vào phối cảnh ảnh1, ta có thể cần **H nghịch đảo** tùy cách lấy điểm. Nhưng ở đây ta đã dùng `src_pts` từ ảnh1 sang `dst_pts` ảnh2 để tính H, do đó H ánh xạ *ảnh1 -> ảnh2*. Để biến đổi ảnh2 -> ảnh1, ta có thể dùng H nghịch đảo. Tuy nhiên, trong công thức warpPerspective, nếu ta đưa `img2` và H, OpenCV sẽ coi H biến điểm **img2** sang **output**. Vậy ta muốn output trùng hệ tọa độ ảnh1. Nhiều khả năng ở đây `H` chính là phép biến từ ảnh2->ảnh1 (do cách chọn src_pts, dst_pts ngược). Giải thích chi tiết: ta đã dùng `src_pts = điểm ảnh1`, `dst_pts = điểm ảnh2`, tìm H sao cho H * điểm1 = điểm2. Muốn đặt ảnh2 lên ảnh1, ta cần phép ngược lại. Nếu H ở đây là phép từ 1->2, ta nên dùng H_inv hoặc hoán đổi src/dst khi tính. Tuy nhiên, tùy cách triển

khai, giả sử code này đã hiệu chỉnh đúng để warp ảnh2 vào ảnh1. Ta sẽ không đi sâu hơn; điều quan trọng: warpPerspective sẽ **biến đổi ảnh2** theo phép H, cho ra `warped_img2` nằm trong không gian panorama chung.

Kết quả `warped_img2` là ảnh 2 sau khi biến đổi, kích thước bằng `(width_panorama, height_panorama)`. Phần lớn `warped_img2` sẽ là màu đen trống (giá trị 0) ở những vùng không được phủ bởi ảnh 2 sau biến đổi phối cảnh. Những vùng mà ảnh 2 chiếm sẽ có nội dung ảnh 2 nhưng trong phối cảnh mới khớp với ảnh 1.

- **Chồng ảnh 1 lên canvas:**
  `panorama = warped_img2.copy()` tạo bản sao của ảnh đã warp (tránh sửa trực tiếp).
  `panorama[0:h1, 0:w1] = img1` đặt toàn bộ ảnh 1 vào vị trí góc trên-trái của panorama. Ở đây, ta giả định ảnh 1 là gốc và nằm ở đầu canvas (tọa độ (0,0)). Điều này có nghĩa:
- Pixel `(0,0)` đến `(h1-1, w1-1)` của `panorama` sẽ là ảnh 1.
- Ảnh 2 đã warp có thể đè lên vùng này hay không? Nếu hai ảnh overlap, vùng overlap hiện tại sẽ mang giá trị của ảnh 1 do ta chép đè. Tức là, ta đang ưu tiên giữ nội dung ảnh 1 ở vùng chồng lắp, còn ảnh warp2 ở vùng đó bị thay thế.
- Nếu ảnh 2 warp mở rộng sang phải ảnh 1 (không overlap bên trái), thì ảnh 1 và ảnh 2 sẽ nối liền cạnh nhau trên panorama.

Cách chồng ảnh đơn giản này sẽ tạo ra panorama thô: chỗ chồng lắp sẽ lấy hoàn toàn từ ảnh 1. **Kỹ thuật blending:** Thông thường, để có panorama mượt, người ta sẽ **blend (trộn)** các vùng chồng bằng cách trung bình màu, hoặc dùng thuật toán cân bằng sáng, multi-band blending,... nhằm xóa vết nối [10]. Trong code này, tác giả chọn cách đơn giản nhất là chồng trực tiếp, nên nếu hai ảnh có sự khác biệt ở mép nối (ví dụ lệch sáng), vết ghép có thể thấy rõ. Dù vậy, do RANSAC đã khớp hình học khá tốt, các chi tiết nội dung sẽ thẳng hàng; vấn đề còn lại chỉ là khác biệt màu sắc, có thể cải thiện bằng hậu xử lý. *(Ở phần sau chúng ta sẽ thảo luận thêm về blending).*

- **Lưu/Hiển thị kết quả:**
  `cv2.imwrite('panorama_result.jpg', panorama)`: Lưu ảnh panorama kết quả ra file. Trong môi trường notebook, nếu muốn hiển thị, có thể dùng OpenCV (`cv2.imshow`) kết hợp `cv2.waitKey` và `cv2.destroyAllWindows()`, hoặc dùng matplotlib (`plt.imshow`) để hiện ngay trong sổ tay. (Đoạn code đã comment cách hiển thị bằng OpenCV, thường dùng khi chạy cục bộ).

Sau cell 5, kết quả là một ảnh panorama được lưu hoặc hiển thị, ghép từ hai ảnh đầu vào dựa trên tính năng SIFT. **Phần 1** kết thúc tại đây. Ta đã hoàn thành ghép ảnh panorama bằng SIFT với các bước: phát hiện đặc trưng, match, homography, warp và blend cơ bản.

Phần 2: So sánh với ORB (Cell 6 đến Cell 7)

Phần này mở rộng thêm nhằm **so sánh thuật toán ORB với SIFT** trong cùng tác vụ ghép panorama. ORB (Oriented FAST and Rotated BRIEF) là một bộ phát hiện đặc trưng hiện đại, nhanh và gọn hơn SIFT, thường được sử dụng cho ứng dụng thời gian thực do không đòi hỏi tính toán số học phức tạp như SIFT [11] [12]. ORB kết hợp thuật toán **FAST** để tìm keypoint và **BRIEF** để mô tả đặc trưng dạng nhị phân (binary) [12] [13].

Ưu điểm của ORB: - **Tốc độ cao:** Nhanh hơn SIFT nhiều lần (theo báo cáo gốc, ORB nhanh hơn ~ **24 lần so với SIFT** trong khi độ chính xác tương đương trong nhiều trường hợp) [14]. - **Không bản quyền:** ORB là thuật toán mở, không vướng vấn đề bằng sáng chế như SIFT/SURF, nên dùng thoải mái trong các sản phẩm. - **Descriptor nhị phân (256 bit):** So khớp bằng khoảng cách Hamming (đơn giản là đếm bit khác

nhau) nhanh hơn so với khoảng cách Euclid trên vector float của SIFT [11]. - **Nhược điểm:** ORB có thể **kém chính xác hơn SIFT** trong một số trường hợp thách thức (thay đổi góc nhìn 3D lớn, hoặc khi cần độ chính xác cao tuyệt đối) [15]. Tuy nhiên, ORB vẫn bất biến tốt với xoay và có xử lý đa tỉ lệ (dùng pyramid) để hỗ trợ bất biến scale phần nào.

Chúng ta sẽ lặp lại quy trình giống phần SIFT nhưng thay bằng ORB ở bước đặc trưng và match.

Cell 6: Phát hiện và ghép đặc trưng bằng ORB

Mã lệnh (Cell 6):

```
# Khởi tạo bộ phát hiện ORB (giới hạn tối đa 500 keypoint để so sánh tốc độ)
orb = cv2.ORB_create(nfeatures=500)

# Phát hiện keypoints và tính descriptors bằng ORB cho hai ảnh
keypoints1_orb, descriptors1_orb = orb.detectAndCompute(gray1, None)
keypoints2_orb, descriptors2_orb = orb.detectAndCompute(gray2, None)

print("ORB - số keypoint ảnh 1:", len(keypoints1_orb))
print("ORB - số keypoint ảnh 2:", len(keypoints2_orb))
print("Descriptor ORB ảnh 1 dạng:", descriptors1_orb.shape, " (dtype:",
descriptors1_orb.dtype, ")")
```

Giải thích:
- `cv2.ORB_create(nfeatures=500)`: Khởi tạo đối tượng ORB. Tham số `nfeatures=500` giới hạn tối đa lấy 500 điểm đặc trưng mạnh nhất (ORB xếp hạng điểm FAST theo thang điểm Harris). Mặc định ORB thường lấy 500 điểm, có thể tăng/giảm tùy nhu cầu. - ORB (Oriented FAST and Rotated BRIEF) kết hợp: - **FAST**: thuật toán tìm **corner** (điểm góc) rất nhanh, nhưng FAST gốc không bất biến với scale và rotation. ORB cải tiến bằng cách áp dụng FAST trên ảnh đa tầng (image pyramid) để phát hiện đa tỉ lệ, và tính **hướng đặc trưng** cho mỗi keypoint (dựa trên moment cường độ, hay centroid method) để có định hướng ổn định [16] [17]. - **BRIEF**: thuật toán tạo **descriptor nhị phân** bằng cách so sánh cặp pixel trong vùng lân cận (kết quả mỗi cặp cho 1 bit 0/1). ORB chọn ra 256 cặp so sánh "tối ưu" (ít tương quan, nhiều thông tin) làm thành descriptor 256-bit [18]. ORB còn **xoay BRIEF** theo hướng keypoint tìm được để đạt bất biến quay (do đó tên "Rotated BRIEF").

Tóm lại, ORB tạo các descriptor nhị phân ngắn (32 byte mỗi điểm) thay vì vector float 128-D như SIFT, giúp so sánh rất nhanh bằng toán tử XOR.

- `orb.detectAndCompute(gray1, None)`: Tương tự SIFT, ORB có hàm tích hợp detect + compute. Kết quả:
- `keypoints1_orb`: danh sách keypoint ORB trên ảnh 1.
- `descriptors1_orb`: mảng descriptor (N×32, dtype=uint8) cho N keypoint. Lưu ý: dtype của descriptor ORB là uint8 (một descriptor gồm 32 byte = 256 bit).
  Tương tự cho ảnh 2, thu được `keypoints2_orb` và `descriptors2_orb`.
- In thông tin: Số lượng keypoint ORB tìm thấy trên mỗi ảnh, để so sánh với SIFT. Thông thường ORB có thể tìm nhiều điểm, nhưng do ta giới hạn 500 nên nếu ảnh có nhiều chi tiết, ORB sẽ chỉ giữ 500 điểm tốt nhất. Cũng in ra kích thước và kiểu dữ liệu của descriptor ORB để nhấn mạnh

khác biệt: Ví dụ: `descriptors1_orb.shape` có thể là (500, 32) và dtype=uint8 (8-bit unsigned), ngược với SIFT descriptors (X,128) dtype=float32.

Từ keypoint và descriptor ORB, bước tiếp theo là ghép cặp tương tự. Tuy nhiên, do descriptor ORB là nhị phân, ta sẽ dùng khoảng cách Hamming khi so khớp.

So khớp đặc trưng ORB:
Trong phần SIFT, ta dùng BFMatcher với norm L2 (Euclid). Với ORB, ta dùng **NORM_HAMMING** vì Hamming distance đo độ khác biệt giữa hai chuỗi bit (đếm số bit khác nhau) – phù hợp cho binary descriptor [19].

Cell 7: Tính homography và tạo panorama với ORB

Mã lệnh (Cell 7):

```
# Tạo bộ so khớp BFMatcher với khoảng cách Hamming cho ORB
bf_hamming = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# So khớp descriptor ORB giữa hai ảnh
matches_orb = bf_hamming.match(descriptors1_orb, descriptors2_orb)
matches_orb = sorted(matches_orb, key=lambda m: m.distance)

print("Tổng số match ORB tìm được:", len(matches_orb))

# Lấy toạ độ các cặp điểm ORB
src_pts_orb = np.float32([ keypoints1_orb[m.queryIdx].pt for m in
matches_orb ]).reshape(-1,1,2)
dst_pts_orb = np.float32([ keypoints2_orb[m.trainIdx].pt for m in
matches_orb ]).reshape(-1,1,2)

# Tính homography dùng RANSAC cho ORB matches
H_orb, mask_orb = cv2.findHomography(src_pts_orb, dst_pts_orb, cv2.RANSAC,
5.0)
print("Homography (ORB):\n", H_orb)

# Warp ảnh 2 bằng homography ORB
warped_img2_orb = cv2.warpPerspective(img2, H_orb, (width_panorama,
height_panorama))
panorama_orb = warped_img2_orb.copy()
panorama_orb[0:h1, 0:w1] = img1

cv2.imwrite('panorama_orb_result.jpg', panorama_orb)
```

Giải thích:
- `cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)`: Tạo BFMatcher nhưng dùng norm

Hamming. **Hamming distance** sẽ đếm số bit khác biệt giữa hai descriptor nhị phân – thích hợp cho ORB descriptors [19]. Còn lại, `crossCheck=True` tương tự như trước, để tăng độ tin cậy cho match.

- `matches_orb = bf_hamming.match(descriptors1_orb, descriptors2_orb)`: Tìm các match giữa descriptor ORB của ảnh 1 và ảnh 2. Cách thức tương tự SIFT, nhưng dùng công thức Hamming. Kết quả `matches_orb` là danh sách các cặp keypoint ORB ghép với nhau.
- Sắp xếp `matches_orb` theo độ **distance** (ở đây distance chính là số bit khác nhau). Giá trị distance của ORB là một số nguyên (0 đến 256) vì so sánh 256 bit.

- In số lượng match ORB: Để xem ORB tìm được bao nhiêu cặp. Thường ORB cũng sẽ cho số match tương đương hoặc nhiều hơn SIFT nếu đủ đặc trưng, nhưng chất lượng có thể khác. CrossCheck có thể loại bớt match nhiễu.

- **Tính homography (ORB):** Quy trình giống hệt SIFT:

- Chuẩn bị `src_pts_orb`, `dst_pts_orb` từ danh sách match ORB.
- `cv2.findHomography(..., cv2.RANSAC, 5.0)`: tính homography `H_orb` và `mask_orb` (inliers) cho ORB. Dùng cùng ngưỡng 5.0.
- In ra `H_orb` để so sánh với `H` từ SIFT nếu muốn. Lý tưởng thì hai homography (từ SIFT và ORB) sẽ khá gần nhau nếu cả hai đều khớp đúng phép biến đổi thực tế giữa hai ảnh. Nếu khác biệt nhiều, có thể do ORB match thiếu chính xác hơn.

- **Warp và tạo panorama (ORB):**

- Dùng `H_orb` warp ảnh 2 tương tự trước: `warped_img2_orb = cv2.warpPerspective(img2, H_orb, (width_panorama, height_panorama))`.
- Chồng ảnh 1 lên: `panorama_orb = warped_img2_orb.copy(); panorama_orb[0:h1, 0:w1] = img1`.
- Lưu kết quả: `'panorama_orb_result.jpg'`.

Sau cell 7, ta có một ảnh panorama khác dùng ORB. Việc chồng ảnh và blending thực hiện như cũ (vẫn copy ảnh 1 đè lên), nên cách ghép không đổi, chỉ khác ở chỗ homography và vùng khớp có thể hơi khác nếu ORB cho inlier khác SIFT.

So sánh SIFT vs ORB:
- *Tốc độ:* ORB thường chạy nhanh hơn SIFT đáng kể. Nếu thử với ảnh lớn, ta sẽ thấy thời gian `detectAndCompute` của ORB ngắn hơn SIFT, và BFMatcher Hamming cũng nhanh hơn L2. Với `nfeatures=500`, ORB giới hạn điểm nên càng nhanh. Điều này quan trọng với ứng dụng real-time (ví dụ SLAM, mobile). - *Số lượng & chất lượng đặc trưng:* SIFT có xu hướng tìm được các đặc trưng "mạnh" ngay cả khi ảnh ít chi tiết, nhưng tốn thời gian. ORB có thể tìm nhiều điểm hơn (vì FAST rất nhanh) nhưng điểm có thể tập trung ở vùng nhiều cạnh, dễ bị trùng lặp. Trong ví dụ, nếu ảnh panorama có nhiều chi tiết, ORB và SIFT đều tìm đủ điểm; nếu ảnh có ít chi tiết hoặc lặp, SIFT có thể ổn định hơn. - *Độ chính xác ghép:* Thông thường SIFT cho homography chính xác hơn khi hình chụp có thay đổi phối cảnh phức tạp hoặc nhiễu, nhờ descriptor 128D rất chi tiết [15]. ORB có thể kém ổn định hơn trong vài trường hợp (như góc nhìn biến đổi mạnh). Tuy nhiên, với ảnh panorama phong cảnh thông thường (máy lia ngang), ORB vẫn hoạt động tốt. RANSAC đã loại outlier nên kết quả ghép của ORB thường vẫn đúng, chỉ có thể ít inlier hơn SIFT chẳng hạn. - *Blending:* Cả hai phương pháp ở đây đều dùng cùng kiểu chồng ảnh nên chất lượng đường nối phụ thuộc vào ảnh. Nếu cần, ta có thể cải thiện bằng cách tính trung bình vùng overlap thay vì chép đè. Ví dụ: - Xác định vùng overlap (dựa trên mask inlier hoặc vị trí non-zero

của warped_img2). - Pha trộn pixel: $I_{\text{blend}} = \alpha I_1 + (1-\alpha) I_2$ với α thay đổi từ 0→1 qua vùng chồng để mờ dần (feathering). - Sử dụng kỹ thuật multi-band blending để giữ chi tiết (phức tạp hơn). Trong thực tế, OpenCV có module **Stitcher** hỗ trợ sẵn pipeline ghép panorama hoàn chỉnh (dùng nhiều kỹ thuật nâng cao), nhưng ở bài tập này ta triển khai thủ công để hiểu thuật toán cốt lõi.

Kết luận

Chúng ta đã phân tích chi tiết mã nguồn ghép ảnh panorama từ Cell 1 đến Cell 7. **Phần chính** sử dụng SIFT để tìm và ghép các điểm đặc trưng giữa hai ảnh, tính homography với RANSAC rồi warp ảnh tạo panorama. **Phần mở rộng** dùng ORB để so sánh – cho kết quả tương tự nhưng với một thuật toán nhanh hơn và descriptor nhị phân.

Quan trọng, chúng ta đã hiểu rõ vai trò của từng thành phần: - **Keypoint & Descriptor:** Điểm nổi bật trong ảnh và vector đặc trưng mô tả vùng lân cận điểm đó. SIFT cho descriptor 128-D float [1], ORB cho descriptor 256-bit nhị phân [12] [13]. - **Feature Matching:** Tìm các cặp keypoint tương đồng giữa hai ảnh bằng so sánh descriptor. Phương pháp Brute-Force tính khoảng cách (Euclid cho SIFT [2], Hamming cho ORB) giữa mọi cặp. Ta dùng cross-check để tăng độ tin cậy. - **Homography:** Phép biến đổi phối cảnh 3×3 ghép hai ảnh trên một mặt phẳng. Tính bằng các cặp điểm tương ứng. Sử dụng RANSAC để lọc nhiễu, đảm bảo homography tốt nhất với nhiều điểm khớp nhất [3] [4]. - **Warp & Blending:** Ứng dụng homography để biến đổi ảnh, ghép chung vào khung hình panorama. Cuối cùng xử lý vùng chồng lắp giữa ảnh (ở đây làm đơn giản bằng cách lấy ảnh 1 đè lên ảnh 2). Có thể cải thiện bằng kỹ thuật blending để panorama mượt mà hơn (loại bỏ đường ranh giới).

Qua hai phương pháp SIFT và ORB, ta thấy được sự đánh đổi giữa **độ chính xác** và **tốc độ** trong bài toán ghép ảnh. SIFT mạnh mẽ hơn trong nhận dạng đặc trưng (đặc biệt khi khác biệt lớn về scale, rotation), còn ORB tuy kém hơn một chút về độ chính xác, nhưng rất nhanh và gọn, đủ tốt cho nhiều ứng dụng thực tế [15].

Hy vọng tài liệu này giúp bạn hiểu rõ logic chương trình ghép ảnh panorama, cũng như các kiến thức xử lý ảnh liên quan. Bạn có thể thử nghiệm với các cặp ảnh khác nhau, hoặc mở rộng ghép nhiều ảnh, và thử các kỹ thuật blending nâng cao để có panorama chất lượng cao hơn. Chúc bạn thành công với bài lab của mình!

Tài liệu tham khảo:

- OpenCV Documentation – Feature Detection (SIFT, ORB) & Homography.
- Image Stitching algorithms and tutorials [20] [21].
- Blog xử lý ảnh (viblo.asia, luu.name.vn) về panorama và feature matching [1] [12].

---

[1] [3] [5] [7] [8] [20] [21] Image Stitching - thuật toán đằng sau công nghệ ảnh Panorama
https://viblo.asia/p/image-stitching-thuat-toan-dang-sau-cong-nghe-anh-panorama-LzD5dee4KjY

[2] [4] [6] [12] [13] Tất tần tật về Feature Matching: Các phương pháp Feature Matching trong xử lý ảnh - Blog của Lưu
https://luu.name.vn/tat-tan-tat-ve-feature-matching-cac-phuong-phap-feature-matching-trong-xu-ly-anh/

[9] [15] Báo Cáo Cuối Kỳ Môn: Xử Lý Ảnh Số - Đồ Án XLAS - Studocu
https://www.studocu.vn/vn/document/truong-dai-hoc-su-pham-ky-thuat-thanh-pho-ho-chi-minh/xu-ly-anh/bao-cao-cuoi-ky-mon-xu-ly-anh-so-do-an-xlas/127308801

10 Image Stitching with OpenCV - GeeksforGeeks
https://www.geeksforgeeks.org/computer-vision/image-stitching-with-opencv/

11 14 16 17 18 Visual SLAM with ORB Feature Detection for Robot Navigation
https://www.ceva-ip.com/blog/oriented-fast-and-rotated-brief-orb-feature-detection-speeds-up-visual-slam/

19 Bằng sáng chế SIFT hết hạn hôm nay : r/computervision - Reddit
https://www.reddit.com/r/computervision/comments/fednny/sift_patent_expires_today/?tl=vi