Monulens: Real-time mobile-based Landmark Recognition

Aditya Srinivas Timmaraju and Anirban Chatterjee
Dept. of Electrical Engineering
Stanford University
Stanford, CA - 94305
Email: adityast@stanford.edu, achatter@stanford.edu

Abstract—In this report, we present a real-time mobile-based landmark recognition application called Monulens. Contrary to most applications, Monulens does all processing on the mobile device alone, and does not require any server-side support. We present quantitative results regarding the time taken for processing a single query. We also present empirical accuracy of the application. We also formulate results for cross-device testing and observe an interesting trend.

I. INTRODUCTION

The unprecedented penetration of smartphones in the global market and the continuous increase in their memory capacity and processing power have enabled and spurred the development of several applications. In this report, we present one such application for Android, Monulens.

As tourists, people feel bothered about having to stand in queues to get access to digital assistants, which generally are old, used headphones that do not offer rich information about a landmark/monument. However, if an application that can recognize a landmark exists, then all the user will need to do is to point the smartphone at the structure of interest and wait for the detection result. Such an application would also serve as a good starting point to offer further capabilities centred around tourism, such as locating nearby food places, displaying past pictures of the landmark, morphing the user’s face on a region of interest in the landmark etc.

There has been interest in the problem of landmark recognition in the computer vision community for about a decade. However, developments in landmark recognition in a mobile application setting have been relatively recent. Advent of commercial large-scale applications like Google Goggles (see Fig 1) have spurred further interest in this area [4]. However, most commercial applications rely heavily on server-side support and processing. In [4], the authors present an approach that relies on incorporating priors from noisy user location data, in addition to using the image content itself for recognition.

Monulens does not rely on any server-side support for processing, which helps in curbing the latency and power requirements associated with communication to and from the server. The entire processing cycle takes place on the Android device itself and the match is displayed. While client-only implementations offer significant benefits in terms of power and latency, one of the major drawbacks of a server-independent implementation is that data-intensive applications consume a lot of storage space on the portable device. In the context of applications such as landmark recognition, storing data pertaining to tens of thousands of landmarks on a mobile device is practically infeasible. In order to overcome this drawback, we derived inspiration from “maps” applications. Many maps applications allow the user to download data pertaining only to the regions that they would be interested in. This data is downloaded as an array of latitude, longitude values, topology and contours. In the absence of an active connection, the user can still localize herself on the map, since GPS does not rely on a network connection. For our application, we are interested in keypoints detected on the buildings, their descriptors, geotagging data and a brief description of the buildings. This is embedded into a package associated with the user-requested geographical area, significantly reduces the database size. For Monulens, for example, we have found that a simple rule of thumb is that the database size in MegaBytes (MB) is approximately equal to the number of landmarks in the database. We have pictorially illustrated this concept of package-based maps-like application in Figure 2.

The algorithm to determine the closest match relies on an initial GPS-based location filtering phase, to prune the possible candidate landmarks. After this stage, a bag-of-visual-words based histogram matching is used, followed by a geometric consistency check with RANSAC-based homography estimation. Section 2 outlines the details of the algorithm. Section 3
describes the Android application. In section 4, performance results are presented. Section 5 concludes the report.

II. ALGORITHM

The problem of recognizing a landmark belonging to an image is quite challenging, owing to possible variability across the set of positive examples in terms of scale, viewpoint and illumination. While scale variation can be captured effectively using robust scale-invariant feature detectors and descriptors, the other two challenges are not directly amenable to modeling. If our image database consists of images of a landmark from different viewpoints, then this disadvantage is handled to an extent. Tackling illumination invariance is still an active research topic and we circumvent this issue by composing the database with images taken under different illuminations, though there is practically a limit to how much invariance can be captured by this approach.

As a pre-processing step, the query images are resized to a resolution of 640x480 pixels. For matching the query image with images from the database, we use bag-of-visual words histograms [5]. The bag-of-visual words cluster has been trained on the San Francisco Landmark Data Set [4]. When a query frame arrives, the current geo-location information is used to filter out all landmarks that lie outside a radius of 150 meters. The query image is then matched with all candidate landmark images from the database. At this stage, a homography is estimated between the query image and the top retrieved image from the matching phase. The RANSAC [6] algorithm is used to estimate the homography. If the number of inliers in the homography model exceeds a threshold, a match is declared, otherwise the next query frame is processed to look for a potential match. A logic flow diagram of the approach is presented in Fig 3.

In section 4, for performance and cross-device comparison, we use the same algorithm as above, except that the bag-of-visual words histogram is replaced by a vocabulary tree based histogram.

III. APPLICATION

We have developed an Android application that can be used to detect landmarks following the steps that have been outlined in the previous section. Since Android uses Java for front-end design, the app’s front-end was coded in Java; however, image-processing algorithms were implemented by making native calls to OpenCV functions though the Java Native Interface (JNI). This ensures faster processing of the preview frames, while making it trivial to port the algorithm from a PC implementation to an Android mobile. Also, we have chosen to work with the ORB (Oriented and Rotated BRIEF) feature descriptor [2]. ORB is a binary descriptor which is two orders of magnitude faster to compute than SIFT, and hence is more reliable for real-time mobile based applications. It is also more memory efficient, reducing the memory requirement on smartphone.

The following are the steps involved in the application’s control flow:

A. Preview frame processing in Java

The Activity Manager launches an OpenCV camera preview frame, which makes it possible to encode the preview frame data as a cvMat, which can then be broadcasted through JNI to the desired native function by accessing the native memory address. In order to save on processing power, we have implemented a click-and-detect approach where the user can click a button to learn more about the landmark in the current camera view. A button-click detector detects the users intention and either sends the preview frames to the background native function for processing or renders the frame as is, without any processing. Since the bag-of-words (BoW) cluster is fixed and can be pre-determined, it is encoded as a raw resource and shipped with the .apk file. The Java front-end uncompressed this file and broadcasts the array pertaining to the BoW cluster to the JNI function.

B. Preview frame processing in OpenCV

For each query that arrives, the native function pushes the image down the BoW cluster and extracts the histogram. The histograms that are associated with the images in the database can be downloaded as a packet. This is located in a private folder on the SDCard. The native function extracts the BoW histograms for all the database images from the corresponding files in that folder and performs an L1-norm based distance comparison. For the category corresponding to the top match image, the reference image is also accessed from the database (each category has a unique reference image associated with it). Then, a RANSAC-based homography estimation is performed and the inlier threshold criterion is applied.

C. Result display and release of resources

Once a valid match is found, the search result is rendered on the preview frame and the memory resources are released.
The app automatically reverts to the non-search mode. This preserves processing power, while removing unnecessary congestion in the memory.

The screenshots of the working app are shown in Figures 6, 7, 8, 9, 10.

IV. RESULTS

We perform cross-device testing between an iPhone 5 and an Nvidia Tegra Note 7 in the following manner. First, we collect a dataset of images from six different landmarks on the Stanford campus. Our dataset contains 53 images from iPhone and 42 images from Tegra Note 7. We then perform leave-one-out testing, by considering each image from the ‘test’ category as a query image and matching it against all other images in the ‘data’ category. The ‘data’ and ‘test’ categories are alternated to give cross-device testing results. We measure the precision, that is, the ratio of the number of times a query is classified correctly to the number of queries tested. The results are tabulated in Table I.

As can be inferred from Table I, when test and data images are from the same device, the performance is better than when they are from two different devices. This could be attributed to the differences between the two cameras, such as between their exposure durations, shutter speeds, f-numbers, resolutions.

We also observed variation in performance with the number of top retrievals ($k$) from the histogram matching step that are considered for homography estimation. For instance, if $k = 2$, we consider the top 2 matches from the histogram matching step for homography estimation. The results are presented in Fig 5. The time taken for processing of the various stages of the algorithm are shown in Fig 4. As expected, RANSAC-based homography estimation is the most time consuming step in the process, requiring about 523 ms. The next most time intensive step is the bag-of-visual words histogram computation (248 ms), followed by ORB feature extraction (197 ms). The entire processing of a query frame takes 974 ms, which is considerably fast for most practical applications.

We tested the app on each of them 30 times and recorded the number of times a correct recognition is obtained. The results have been tabulated in Table II.

We have also performed empirical testing of the application, for two of the landmarks, namely Packard Electrical Engineering building and Gates Computer Science building.

![Fig. 3. An Illustration of the Logic Flow Diagram of the Algorithm](image1)

![TABLE I. CROSS-DEVICE TESTING](image2)

<table>
<thead>
<tr>
<th>Test</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone</td>
<td>94.44</td>
</tr>
<tr>
<td>Tegra Note 7</td>
<td>87.04</td>
</tr>
<tr>
<td>iPhone</td>
<td>87.18</td>
</tr>
<tr>
<td>Tegra Note 7</td>
<td>94.87</td>
</tr>
</tbody>
</table>

![Fig. 4. Pie-chart depicting processing time for different steps](image3)

![Fig. 5. Precision vs k](image4)

![Fig. 6. Screenshot 1](image5)
TABLE II. EMPIRICAL TESTING

<table>
<thead>
<tr>
<th>Category</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packard</td>
<td>86.66</td>
</tr>
<tr>
<td>Gates</td>
<td>90</td>
</tr>
</tbody>
</table>

Fig. 7. Screenshot 2

V. CONCLUSION AND FUTURE SCOPE

We have presented a real-time (sub-second query time) mobile application for landmark recognition on an Android device. We have enumerated the accuracy of the application, as determined from empirical testing. We have also tabulated the performance results for cross-device testing and observed an interesting phenomenon. We have discussed about the trade-offs involved in server-support based applications and purely client-based applications. We now discuss further improvements that can be made to the application.

A. Algorithm optimization

1) GPU function calls: In the processing time analysis, we have seen that both keypoint detection/description and RANSAC are the computationally expensive steps. We have used standard OpenCV functions to compute the same. However, most modern portable devices are equipped with GPUs that can provide significant boost to the performance by employing GPU variants of the same functions. An example of this is the GPU::ORB function that makes calls to a GPU variant of the ORB keypoint detector(descriptor).

Fig. 8. Screenshot 3

2) SIMD instructions: Many recently proposed keypoint descriptors such as FREAK [3] which have already been implemented in OpenCV, can bank on SIMD-type SSE2/SSE3 registers, thereby boosting the performance at least 2 times for floating-point data types by using vector instructions.

3) Cache-blocking/Prefetching: Since our application is data intensive, standard cache-blocking techniques and prefetching files into cache can lead to significant improvement in performance.

4) Pre-computing keypoints/descriptors of reference images: Since keypoint detection and description are computationally expensive, they can be pre-computed for the reference images. Currently, they are being computed in real-time and this adds to the overall delay of the processing flow.

B. Additional features

1) Overlaying pointers: Since Monulens is capable of recognizing landmarks, an interesting feature that could be added is the capability to overlay pointers on the image, showing directions for a guided tour etc. There are currently many augmented reality platforms such as Vuforia, which allow us to do the same by integrating software development kits (SDKs).

2) Accessing ratings: With many travel guide services such as Lonely Planet and Tripadvisor launching public APIs, we can access ratings and other info about popular landmarks by making REST API calls. This can allow the tourists to share their real-time experiences on social networks too.

ACKNOWLEDGMENT

We would like to thank Prof. Girod for the informative and inspiring lectures and the teaching staff for organizing a seamless course. We would also like to thank Dr. Peter Vajda for mentoring us on the project.

REFERENCES

APPENDIX

The division of efforts is as follows:

- Aditya: Literature survey, algorithm, performance results
- Anirban: Android Application development, On-Device testing
- Combined: Dataset collection, poster, demo, report