SkyStitch: a Cooperative Multi-UAV-based Real-time Video Surveillance System with Stitching

Xiangyun Meng, Wei Wang and Ben Leong
School of Computing
National University of Singapore
idmmeng@nus.edu.sg, weiwang@comp.nus.edu.sg, benleong@comp.nus.edu.sg

ABSTRACT

Recent advances in unmanned aerial vehicle (UAV) technologies have made it possible to deploy an aerial video surveillance system to provide an unprecedented aerial perspective for ground monitoring in real time. Multiple UAVs would be required to cover a large target area, and it is difficult for users to visualize the overall situation if they were to receive multiple disjoint video streams. To address this problem, we designed and implemented *SkyStitch*, a multiple-UAV video surveillance system that provides a single and panoramic video stream to its users by stitching together multiple aerial video streams. SkyStitch addresses two key design challenges: (i) the high computational cost of stitching and (ii) the difficulty of ensuring good stitching quality under dynamic conditions. To improve the speed and quality of video stitching, we incorporate several practical techniques like distributed feature extraction to reduce workload at the ground station, the use of hints from the flight controller to improve stitching efficiency and a Kalman filter-based state estimation model to mitigate jerkiness. Our results show that SkyStitch can achieve a stitching rate that is 4 times faster than existing state-of-the-art methods and also improve perceptual stitching quality. We also show that SkyStitch can be easily implemented using commercial off-the-shelf hardware.

Categories and Subject Descriptors

I.4 [**Image processing and computer vision**]: [Applications]; I.2.9 [**Robotics**]: [Autonomous vehicles]

Keywords

UAV; stitching; real-time

1. INTRODUCTION

The proliferation of commodity UAV technologies like quadcopters has made it possible to deploy aerial video surveillance systems to achieve an unprecedented perspective when monitoring a target area from the air. UAV-based surveillance systems have a broad range of applications including firefighting [2], search and rescue [3], and border monitoring [8].

MM'15, October 26–30, 2015, Brisbane, Australia.

DOI: http://dx.doi.org/10.1145/2733373.2806225.

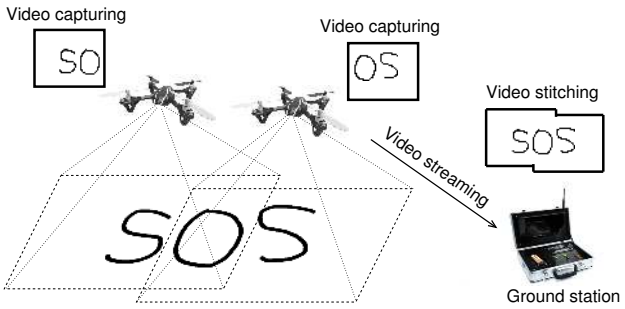


Figure 1: Illustration of SkyStitch. Each quadcopter transmits a live video stream to the ground station, which stitches them together to provide a single composite view of the target area.

For example, a UAV could be deployed to hover over a bushfire to transmit a live video feed to firefighters using wireless links to allow them to appraise the situation from an aerial perspective. However, because on-board cameras have only a limited field of view, multiple UAVs would be required to cover a large target area. If the firefighters were to receive multiple disjoint video streams from each UAV, it would be difficult for them to put together a consistent view of the situation on the ground.

To address this problem, we designed and implemented SkyStitch, a quadcopter-based HD video surveillance system that incorporates *fast video stitching*. With SkyStitch, users on the ground would receive a single composite live video feed produced from the video streams from multiple UAVs. This is illustrated in Figure 1, where an SOS sign is marked on the ground by victims trapped in a bushfire.

Although the high-level idea of SkyStitch is relatively straightforward, there are two practical design challenges. The first challenge is that image stitching inherently has a very high computational cost. Existing stitching algorithms in both industry and academia [5, 26, 13] were designed for better stitching quality rather than shorter stitching time. Although some algorithms can achieve shorter stitching times [6, 27, 31], they impose an additional requirement that the cameras be in fixed relative positions, which is infeasible in our operating scenario as the cameras are mounted on moving quadcopters. The other challenge is that existing image stitching algorithms are designed for static images, and not video. We found that even though the individual frames produced by image stitching algorithms might look fine, the video obtained by naively stitching them together would usually suffer from perspective distortion, perspective jerkiness and frame drops.

To address these challenges, we made a few key observations about our unique operating environment and developed several practical techniques to improve both stitching efficiency and quality. One observation is that, although each UAV has only a limited pay-

load, each on-board computer has a fair amount of computational power. In SkyStitch, we divide the video stitching process into several tasks and offload some pre-processing to the UAVs. In this way, we can improve the speed of stitching by reducing the workload of the ground station, which would be a potential bottleneck when there are a large number of UAVs. Another observation is that we can exploit the hints from instantaneous flight information like UAV attitude and GPS location from the on-board flight controller to greatly improve the efficiency of stitching and rectify perspective distortion. Our third observation is that image stitching and optical flow have complementary properties in terms of long-term stability and short-term smoothness. Therefore, a Kalman filter-based state estimation model can be applied by incorporating optical flow information to significantly reduce perspective jerkiness.

We deployed SkyStitch on a pair of low-cost self-assembled quadcopters and evaluated its performance for two ground scenarios with very different feature distributions. Our experiments showed that SkyStitch achieves a shorter stitching time than existing methods with GPU acceleration [26], e.g., SkyStitch is up to 4 times faster for 12 concurrent aerial video sources. In other words, for a fixed frame rate, SkyStitch can support many more video sources and thus cover a much larger target area in practice. For example, at a stitching rate of 20 frames per second (fps), SkyStitch is able to stitch videos from 12 quadcopters while existing methods can only support three with the same hardware. In addition, SkyStitch is able to achieve better stitching quality by rectifying perspective distortion, recovering stitching failures and mitigating video jerkiness. Furthermore, SkyStitch can be easily implemented using commercial off-the-shelf hardware thereby reducing the cost of practical deployment.

The rest of this paper is organized as follows. In Section 2, we present the prior work related to SkyStitch. In Section 3, we discuss the design details of SkyStitch, followed by its implementation in Section 4. In Section 5, we present the performance evaluation of SkyStitch. Finally, we conclude this paper in Section 6.

2. RELATED WORK

Image stitching has been widely studied in the literature. Most available stitching algorithms are focused on improving stitching quality without considering stitching time [26, 16, 13]. Recently, image stitching algorithms have been applied in UAV-based incremental image mosaicking [32]. Basically, a UAV flies over a target area along a pre-planned route and continuously takes images, which are transmitted to a ground station and incrementally stitched to produce a large image of the target area. Unlike SkyStitch, such applications do not have a stringent requirement on stitching rate. Wischounig et al. improved the responsiveness of mosaicking by first transmitting meta data and low-resolution images in the air, and only transmitting the high-resolution images at a later stage, i.e., after landing [30]. There are also commercial systems for UAV-based image mosaicking. The company NAVSYS claims that its system can achieve a stitching rate of one image per second [12].

There are several existing video stitching algorithms that can achieve fast stitching rate, but most of them only work for a static camera array like the one used by Google Street View [10]. Tennoe et al. developed a real-time video stitching system using four static HD cameras installed in a sports stadium [27]. The key idea of their design is to pipeline the stitching process and implement it in GPU, and it is able to achieve a stitching rate of 30 fps. Adam et al. utilized belief propagation to obtain depth information of images captured on a pair of fixed cameras [9]. Their GPU-based implementation is able to achieve a rate of 37 fps when stitching two 1600×1200 images, but it requires a calibration process to compute the overlapping region of the camera views. Another GPU-based method relied on specific photogrammetric procedure to find the cameras' orientation and translation relative to a common coordinate system, and it could support a stitching rate up to 360 fps when stitching four 1360×1024 images [15]. A CPU-based method of video stitching can be found in [31]. Although these methods achieve fast stitching rates, they assume that the cameras are in a static position and require system calibration and are thus not applicable to stitching videos from highly dynamic UAVs.

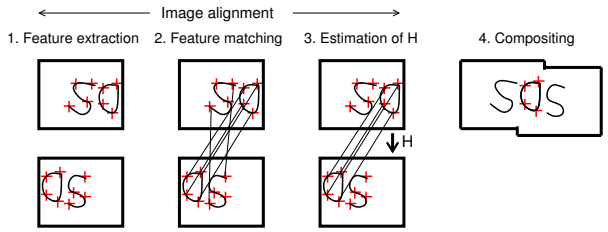


Figure 2: Overview of image stitching.

As smartphones become ubiquitous, some researchers have tried in recent times to improve viewers' viewing experience by managing video streams from multiple smartphones. Kaheel et al. proposed a system called Mobicast which could stitch several 352×288 video streams from smartphones that captured the same event [22]. While the frame rate of Mobicast was able to meet real-time requirements, its stitching quality in real experiments was generally poor, with a stitching failure rate of some 60%. El-Saban et al. improved the frame rate of a similar system by considering the temporal correlation of frames [17]. Since most smartphones are equipped with various sensors, we believe that the methods proposed in this paper could potentially also be applied to improve the performance of smartphone-based video stitching systems.

3. REAL-TIME VIDEO STITCHING

Image stitching involves the alignment and blending of a pair of images with some overlap into a single seamless image. Video stitching is effectively the stitching of a series of image pairs (i.e., video frame pairs). Image stitching is computationally expensive and it takes several seconds to stitch a pair of HD images using Hugin [5], a popular image stitching software, on a workstation with a 3.4 GHz CPU. The challenge of real-time video stitching is to ensure that we can stitch a set of video frames within the required video frame interval. In this paper, we will use image and video frame interchangeably. In this section, we describe the steps required in image stitching and explain how SkyStitch improves the efficiency and quality of video stitching.

The key task in image stitching is to find the transformation between two images. Assume the ground is a planar surface, and let each pixel location in an aerial image be represented by a *homogeneous coordinate* $(x, y, 1)$. Then the pixel locations of the overlapping area in the two images are related by

$$[x'\ y'\ 1]^T \sim \mathbf{H} \cdot [x\ y\ 1]^T \quad (1)$$

where $\mathbf{H}$ is a 3×3 homography matrix for transformation. In order to align two images, we need to estimate $\mathbf{H}$.

The most popular method for $\mathbf{H}$ estimation is the *feature*-based method illustrated in Figure 2. Features refer to the unique parts in an image, e.g., corners are often considered as features. A feature is usually represented using a high-dimensional vector, called *feature descriptor*, which encodes feature information like gradient and orientation histogram in the neighborhood. A feature-based

method for estimating $\mathbf{H}$ generally consists of the following three steps: (i) feature extraction, (ii) feature matching and (iii) robust homography estimation (e.g., using RANSAC [18]).

Once the image alignment (i.e., $\mathbf{H}$) is determined, it remains to combine the images into a single composite image while minimizing visual artefacts such as seams, blurring and ghosting. A more detailed discussion on image stitching can be found in [26].

The challenge of video stitching is that most of the abovementioned steps in image stitching are computationally expensive. Our main contributions in SkyStitch are several techniques to reduce the processing time of video frame stitching to the scale of tens of milliseconds, while maintaining good stitching quality, as follows: (i) we reduce the workload of the ground station by offloading feature extraction to quadcopters (§ 3.1); (ii) we exploit flight status information to speed up feature matching and rectify perspective distortion (§ 3.2); (iii) we propose an efficient GPU-based RANSAC implementation (§ 3.3); and (iv) we utilize optical flow and Kalman filtering to improve video smoothness (§ 3.4). Finally, we developed a new approach for closing loops when stitching multiple video sources (§ 3.5).

3.1 Offloading Feature Extraction

In traditional image stitching systems, feature extraction and the subsequent processing steps are all performed on a single computer. However, such an approach might be problematic for SkyStitch because the ground station would become the computation bottleneck, especially when the number of quadcopters is large.

We noticed that there is an on-board computer in each quadcopter, and thus we can offload feature extraction to the quadcopters to reduce the amount of computation at the ground station. In other words, instead of transmitting only a video frame to the ground station, a quadcopter would also compute and append the corresponding features, which are simply a series of high-dimensional vectors, for each frame.

The computer on our quadcopters is an NVIDIA Jetson TK1 board. We implemented feature extraction using its GPU, which has 192 CUDA cores. We adopted the latest ORB method [25] for feature extraction, as it has been shown to achieve comparable accuracy to other methods such as SIFT [23] and SURF [11] but at a much lower computational cost. Profiling shows that extracting 1,000 features from a 1280×1024 image takes less than 35 ms.

3.2 Exploiting Flight Status Information

It is difficult to achieve short stitching time and high stitching quality at the same time. First, because of wind turbulence, the on-board camera will not always be pointing vertically downwards. Consequently, the images taken from a tilted camera will almost certainly have some amount of perspective distortion, which affects the quality of stitching.

Second, feature matching between two images is time-consuming since all the features have to be pairwise compared. For example, matching the features from two images with 1,000 features each would require one million pairwise comparisons of feature descriptors. This would take around 16 ms on a workstation with a 3.4 GHz CPU. Furthermore, for each feature in an image, there is at most one correct feature match in the corresponding image. This means that the probability of incorrect matching will be high when the number of features is large. If there are many mismatched features, the quality of stitching will suffer.

We observed that we have more information about the images to be stitched compared to conventional image stitching. In particular, because quadcopters are equipped with various on-board sensors for flight control, we are able to derive a relationship between the two images from the flight status information. We can detect and correct for perspective distortion and determine the region of overlap between two images to significantly reduce the target region in the images for feature matching.

3.2.1 Orthorectification of Perspective Distortion

Perspective distortion occurs when the camera is not pointing vertically downwards to the ground. With orthorectification, we can transform the image so that it looks as if the camera is pointed vertically downwards.

Since the degree of camera tilt is determined by the angles of roll and pitch, we can correct for the resulting perspective distortion using the attitude information from the flight controller. Let θ_r, θ_p and θ_y be the Euler angles of roll, pitch and yaw in the coordinate system of the flight controller, respectively (see Figure 3). Then the transformation matrix (denoted by the matrix $\mathbf{R}$) to map from strictly downwards camera direction to the tilted direction can be expressed as follows:

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_r & \sin\theta_r \\ 0 & -\sin\theta_r & \cos\theta_r \end{bmatrix} \begin{bmatrix} \cos\theta_p & 0 & -\sin\theta_p \\ 0 & 1 & 0 \\ \sin\theta_p & 0 & \cos\theta_p \end{bmatrix} \begin{bmatrix} \cos\theta_y & \sin\theta_y & 0 \\ -\sin\theta_y & \cos\theta_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

We include the yaw angle in $\mathbf{R}$ so that orthorectification can also align the two images to a common orientation (i.e., to the north).

As the angles in the attitude information are with respect to the flight controller itself, we need another two simple transformations for orthorectification. One is needed to transform the pixel locations of an image (originally in pixel coordinates) into world coordinates that is centered at the camera. This can be easily achieved using *camera intrinsic matrix* $\mathbf{K}$. The other transformation (denoted by matrix $\mathbf{B}$) is needed to map from the world coordinates centered at the camera to the world coordinates centered at the flight controller. Both $\mathbf{K}$ and $\mathbf{B}$ can be easily obtained via calibration.

In summary, the orthorectification process consists of (i) mapping the pixels of an image into the world coordinates system of the flight controller (using matrices $\mathbf{K}$ and $\mathbf{B}$), (ii) correcting for the effect due to camera tilt (using matrix $\mathbf{R}$), and (iii) mapping them back to pixel coordinates (again using matrices $\mathbf{K}$ and $\mathbf{B}$). The orthorectification matrix (denoted by $\mathbf{M}$) can be expressed as,

$$\mathbf{M} = \mathbf{K}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^{-1}\mathbf{K}^{-1} \tag{3}$$

Let $(x, y, 1)$ be the original homogeneous pixel coordinates. Then the corresponding pixel coordinates after orthorectification (x', y', z') can be expressed using matrix multiplication,

$$[x'\ y'\ z']^T = \mathbf{M}\,[x\ y\ 1]^T \tag{4}$$

3.2.2 Search Region Reduction for Feature Matching

To reduce feature matching time, we take the spatial locations of features into consideration. Like image orthorectification, we exploit the available flight status information to find the mapping of pixel locations between two images. We first transform the pixel locations from pixel coordinates to a world coordinate system, with the center of the camera sensor as the origin, and merge the two world coordinate systems using the reported GPS locations (see Figure 4). Recall (x', y', z') are the coordinates of a pixel location after orthorectification. Let s be the number of meters represented by the side of a pixel, $W \times H$ be the image resolution (in pixels), θ_f be the angle of horizontal field of view in the camera, and h be the quadcopter altitude. Then,

$$s = 2h\tan(\frac{\theta_f}{2})/W \tag{5}$$

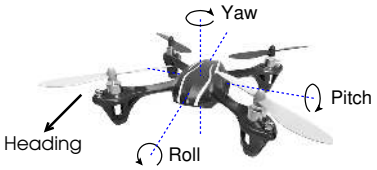
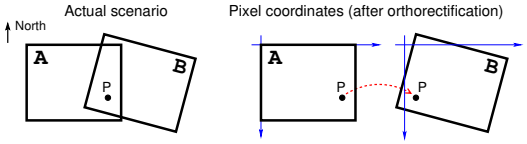
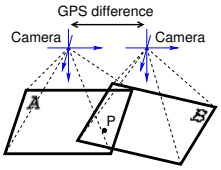


Figure 3: Flight attitude of quadcopter: roll, pitch and yaw.



(a) Mapping a pixel location from one image to another.



(b) Merging coordinate systems.

Figure 4: Overlap determination using flight status information.

Then the final world coordinates (x_w, y_w, z_w) can be obtained by,

$$\begin{cases} x_w = s\,(\frac{x'}{z'} - \frac{W}{2}) \\ y_w = s\,(\frac{y'}{z'} - \frac{H}{2}) \\ z_w = h \end{cases} \tag{6}$$

In other words, each pixel of an image is mapped to a point (x_w, y_w) on the ground, with the center of the camera sensor as the coordinate origin. The two world coordinate systems can then be correlated using the difference in the GPS locations. With such information, we only need to search in a small local neighborhood for the matched feature. Because the area of the local neighborhood is often much smaller than the whole image, we can significantly reduce the search time of feature matching as compared with the traditional method of searching the entire image for the target feature.

This simple idea is illustrated in Figure 5. P is the location in Image 2 that we map for a feature point from Image 1, and r is the search radius for the matched feature. Since an image covers a finite 2D space, a grid indexing approach is preferred. We divide Image 2 into 50×40 bins. 50×40 was chosen because it is proportional to the aspect ratio of the image and it yields bins that are small enough. We would use more bins for a bigger image. If a bin is within a distance r from P (i.e., the colored bin), all the features in the bin will be considered as candidates for feature matching.

It remains for us to determine an appropriate value for the search radius r. Fortunately, the simple approach of setting a fixed and conservative value for r based on empirical experiments seems to be sufficient. In particular, we found that $r = 200$ pixels was sufficient in our implementation. In Section 3.4, we show that r can be further reduced if we estimate and compensate for GPS errors.

3.3 Optimizing RANSAC

The step after feature matching is to use the standard RANSAC method [18] to remove outliers. RANSAC arbitrarily selects four matched features to compute a homography matrix $\mathbf{H}$, applies this estimated $\mathbf{H}$ on all the matched features, repeats it for N iterations and finds the one with the most inliers. In challenging cases where the inlier ratio is low, we often set N to a large value (e.g., 500) in the hope that at least one subset of the four matched features would not contain any outliers.

While a large N improves RANSAC's likelihood of finding the correct homography, it also increases its computation time. For example, it takes about 11 ms to perform a 512-iteration RANSAC on a computer with a 3.4 GHz Intel CPU. This means that it is impossible to stitch more than 4 videos at 30 fps. In order to speed up RANSAC, we noticed that the traditional Singular Value Decomposition (SVD) method is unnecessary for 4-point homography and can be replaced with Gauss-Jordan elimination. The Gauss-Jordan method runs efficiently on GPU due to its simple and branchless structure, whereas an SVD implementation is usually much more complicated and GPU-unfriendly. Finally, we implemented an OpenCL-based RANSAC homography estimator that can compute a 512-iteration RANSAC in 0.6 ms.

To further improve RANSAC performance when the number of image pairs is large, we maximize parallelism and minimize IO overhead by performing RANSAC on all image pairs in one pass (Figure 6). First, candidate homographies of all image pairs are computed in a single kernel execution. The candidate homographies are then routed to corresponding image pairs for reprojection error computation and inlier mask generation. Finally, the scores and inlier masks of all candidate homographies are downloaded to main memory. Our RANSAC pipeline has only four CPU-GPU memory transactions and the amount of parallelism at each stage is much higher than performing RANSAC sequentially for each image pair.

In addition to execution speed, we also made improvements to the accuracy of RANSAC. Traditionally, each RANSAC iteration is scored based on the number of inliers. While this approach works well most of the time, we found that when the inliers are concentrated in a particular region, it may result in a bad estimation of $\mathbf{H}$. To mitigate this problem, we modified the score by taking into consideration the distribution of inliers. Specifically, we use a score that is a function of both the number of inliers and the standard deviations of their locations:

$$score = N_{inliers}\,\sigma_x\,\sigma_y \tag{7}$$

This score can be efficiently computed on the GPU during inlier mask generation. The RANSAC iteration with the highest score under this new metric was used to provide the inliers for the final $\mathbf{H}$ estimation.

3.4 Optimizing Video Stitching Quality

Given the set of inliers identified by RANSAC, we can proceed to compute the homography matrix $\mathbf{H}$ using least squares and then composite the two images based on $\mathbf{H}$ [26]. However, even though the set of inliers appears to be the best that we can find, it is possible for the resulting $\mathbf{H}$ to be ill-formed and to produce a stitched image with some perceptual artefacts. In particular, part of a stitched video frame may suddenly "jerk" when compared to the video frames before and after it.

There are two common causes for such "stitching failures". One is that the overlapping area between the two images is too small, and the resulting matched features are not sufficient to produce a good estimate of $\mathbf{H}$. Our experiments showed that the stitched images generally start exhibiting a certain degree of perceptual distortion when the number of inliers is smaller than 30. An example is shown in Figure 7(a). The other cause is that the original set of inliers selected by RANSAC are concentrated in a small region, and the resultant $\mathbf{H}$ becomes less stable, causing distant regions of images to be misaligned. An example is shown in Figure 7(b).

As human vision is sensitive to sudden scene change, we need to detect ill-formed $\mathbf{H}$ and prevent them from affecting the perceptual experience. In particular, we first determine whether there are sufficient inliers produced by $\mathbf{H}$. If the number of inliers is smaller than 30 (which we had determined from empirical experiments), we consider it as a failure and discard the current frame.

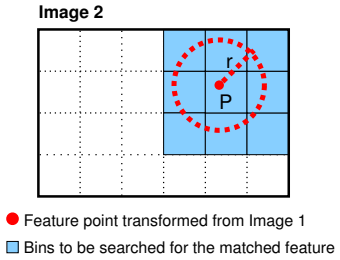


Figure 5: Search region reduction for feature matching.

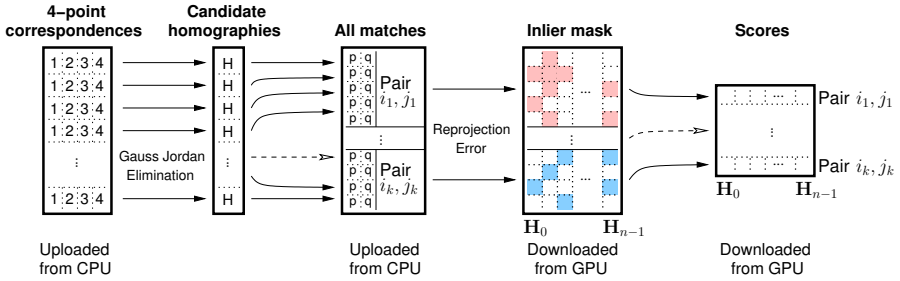
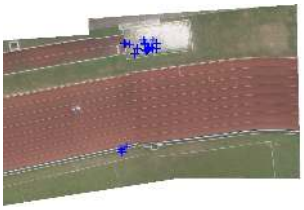
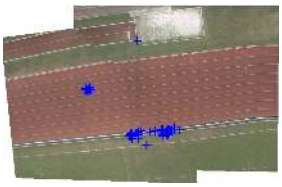


Figure 6: RANSAC GPU Pipeline



(a) Inaccurate rotation. Notice the broken and bent touchline of the football field, below the running track.



(b) Perspective distortion. Notice that the left part of the image is erroneously warped and cannot be perfectly aligned with the right image.

Figure 7: Examples of perceptual distortion. Each blue cross represents a feature point.

When there are sufficient inliers, we also check whether $\mathbf{H}$ badly warps the images and introduces perspective distortion. Let θ be the angle between the two image planes after orthorectification using $\mathbf{M}$. Ideally, θ should be zero, but in practice it could be a small non-zero value due to the errors in the readings of our on-board sensors. To find θ we do homography decomposition on $\mathbf{H}$ [24]

$$\mathbf{H} = \mathbf{R} + \mathbf{t}\,\mathbf{n}^T \tag{8}$$

$\mathbf{R}$ is a 3×3 rotation matrix that transforms camera from the reference image to the target image. We are concerned about the angle between the reference xy-plane and the target xy-plane, which is simply the angle between the reference z-axis and the target z-axis:

$$cos(\theta) = r_{33} \tag{9}$$

By analyzing 300 stitched frames that do not exhibit noticeable perspective distortion (i.e., $\mathbf{H}$ is well-formed), we found that 90% of them have $|\theta| \leq 5^\circ$. Thus, we use θ of 5° as the threshold to determine whether $\mathbf{H}$ is erroneously estimated.

3.4.1 Recovering from Failed Stitching

While discarding ill-formed $\mathbf{H}$ alleviates jerkiness in the stitched video frames, it also leads to frame drops and viewers may notice discontinuity in the video. To reduce frame drops, we use an alternative method to "predict" $\mathbf{H}$ when the original method fails, by utilizing the optical flow of adjacent frames.

This prediction method is illustrated in Figure 8. $\mathbf{F}$ is the 3×3 homography matrix of optical flow, which can be accurately and efficiently computed on each quadcopter as adjacent frames usually have large overlap. $\mathbf{F}_{n+1}$ is transmitted together with frame $n+1$ to the ground station, which will predict $\mathbf{H}_{n+1}$ as follows:

$$\mathbf{H}_{n+1} = \mathbf{M}_{2,n+1}\,\mathbf{F}_{2,n+1}\,\mathbf{M}_{2,n}^{-1}\,\mathbf{H}_n\,\mathbf{M}_{1,n}\,\mathbf{F}_{1,n+1}^{-1}\,\mathbf{M}_{1,n+1}^{-1} \tag{10}$$

where $\mathbf{M}$ is the orthorectification matrix described in Section 3.2.1 and $\mathbf{H}_n$ is the transformation matrix of the previous stitching which

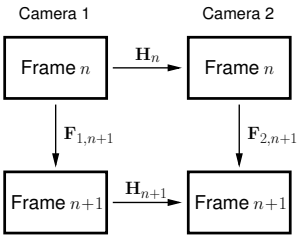


Figure 8: Predict $\mathbf{H}_{n+1}$ using optical flow when $\mathbf{H}_{n+1}$ is ill-formed.

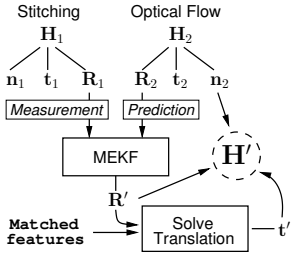


Figure 9: Fusing information from both stitching and optical flow.

was not ill-formed. We check that $\mathbf{H}_{n+1}$ is not ill-formed according to the criteria described in the previous section.

3.4.2 Fusing Stitching and Prediction

In cases where stitching succeeds, we have two homography solutions: one is from pairwise stitching and the other is from prediction by optical flow. Stitching is independently performed for each pair of frames so it does not take temporal information into consideration. Optical flow makes use of previous results to estimate current homography but it could suffer from drift errors in the long term. The property of stitching is like an accelerometer: stable in long term but noisy in short term. Optical flow is like a gyroscope: accurate in the short term but prone to drift over time. This implies that sensor filtering techniques can be applied to achieve a better estimate by considering both stitching and prediction results.

The fusion procedure is shown in Figure 9. First, the two homography matrices are decomposed into rotations and translations (eq. 8). The rotation matrices $\mathbf{R}_1$ and $\mathbf{R}_2$ are converted to quaternions and fed into a Multiplicative Extended Kalman Filter (MEKF) [28]. The quaternion produced by stitching acts as a measurement vector and the quaternion produced by optical flow is used to generate a state transition matrix. We chose not to filter both the rotation and translation at the same time because the filtered homography would generate noticeable misalignment. Instead, the filtered rotation $\mathbf{R}'$ is used to compute a least-squares solution of translation $\mathbf{t}'$ using matched features. Finally, the filtered rotation $\mathbf{R}'$ and recomputed translation $\mathbf{t}'$ are combined to form a new homography matrix $\mathbf{H}'$.

3.4.3 GPS Drift Compensation

We explained in Section 3.2.2 that we can improve the speed of feature matching by focusing on a small search region with radius r. The purpose of r is to tolerate the potential errors when mapping pixels between two images since there are likely errors in the flight status information from on-board sensors, i.e., GPS drift errors.

We observed that when two images are successfully stitched, we will know the ground truth of the transformation. By comparing the ground truth with the transformation which relies on GPS locations

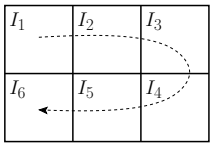


Figure 10: A loop sequence of stitching 6 images.

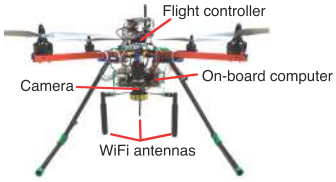


Figure 11: Our self-assembled quadcopter.

(Section 3.2.2), we can estimate the GPS error. Subsequently, we feed this estimated GPS error to feature matching and reduce r by subtracting the GPS error for the next round of stitching. Experimental results show that r can be reduced to smaller than 30 pixels, which makes the search area less than $^1/_{300}$ of total image size. If stitching fails, we reset r to the original conservative value.

3.5 Multiple Video Sources

Stitching multiple video frames can be considered as iteratively performing pairwise stitching. One problem is to decide which image pairs can be stitched. Previous approaches perform pairwise feature matching for all the images to identify the pairs of overlapping images based on matched features [13]. Such exhaustive search is too costly for SkyStitch. Instead, we make use of available flight status information to identify overlapping images and find a sequence of images for stitching. If we assume the quadcopters hover in a grid-like pattern, a simple stitching sequence would be a loop that traverses all the images as shown in Figure 10.

While such loop-based stitching is simple, stitching errors will gradually accumulate along the loop. Consequently, there could be significant distortion between the first and last images, e.g., between images I_1 and I_6 in Figure 10. To address this problem, our solution is to perform adjustment on every pairwise stitching along the loop so that the distortion can be gradually corrected.

Suppose we want to stitch four images in Figure 10, according to the sequence I_1, I_2, I_5 and I_6. Let $\mathbf{H}_1$, $\mathbf{H}_2$, $\mathbf{H}_3$ and $\mathbf{H}_4$ be the homography matrices of image pairs $\langle I_1, I_2\rangle$, $\langle I_2, I_5\rangle$, $\langle I_5, I_6\rangle$ and $\langle I_6, I_1\rangle$ respectively. Ideally, we should have

$$\mathbf{M}_1\mathbf{p} \sim \mathbf{H}_4\mathbf{H}_3\mathbf{H}_2\mathbf{H}_1\mathbf{M}_1\mathbf{p}$$

where $\mathbf{p}$ is a feature point in image I_1 and $\mathbf{M}_1$ is the orthorectification matrix of I_1. However, in practice, $\mathbf{H}_4\mathbf{H}_3\mathbf{H}_2\mathbf{H}_1$ is often not an identity matrix (even after scaling) because of errors in the homography matrices. To make the above relationship hold, we can introduce a correction matrix $\mathbf{H}$ such that:

$$\mathbf{M}_1\mathbf{p} \sim \mathbf{H}\mathbf{H}_4\mathbf{H}_3\mathbf{H}_2\mathbf{H}_1\mathbf{M}_1\mathbf{p}$$

Then we decompose $\mathbf{H}$ into four "equal" components and distribute them among the original four homography matrices. One way to decompose $\mathbf{H}$ is RQ decomposition:

$$\mathbf{H} = \mathbf{A}\mathbf{R}$$

Here $\mathbf{A}$ is an upper triangular matrix (affine) and $\mathbf{R}$ is a 3D rotation matrix. $\mathbf{A}$ can be further decomposed as follows:

$$\mathbf{A} = \mathbf{T}\mathbf{S}\mathbf{D}$$

where $\mathbf{T}$ is a 2D translation matrix, $\mathbf{S}$ is a scaling matrix and $\mathbf{D}$ is a shearing matrix along x-axis. $\mathbf{T}$, $\mathbf{S}$, $\mathbf{D}$ can be taken nth real root where n is number of images in the loop. We can write $\mathbf{H}$ as

$$\mathbf{H} = \mathbf{T}'^4\mathbf{S}'^4\mathbf{D}'^4\mathbf{R}'^4$$

Since the reference image is well orthorectified, $\mathbf{A}$ and $\mathbf{R}$ have geometric meanings. They represent how a camera should be rotated and translated to match the reference camera frame. We move each element from $\mathbf{H}$ into $\mathbf{H}_1, \cdots, \mathbf{H}_4$ to slightly adjust each camera's rotation and translation. For example, we move one $\mathbf{R}'$ into $\mathbf{H}_1$ so that the rotational error is reduced by $^1/_4$. In this way, we can partially correct the errors at each stitching step along the loop to mitigate the distortion between images I_1 and I_6. Note that for a 6-image stitching loop as in Figure 10, we can also perform a similar adjustment to correct the distortion between images I_2 and I_5.

Comparison with Bundle Adjustment. The classical method to close loops is to do bundle adjustment [13]. We tested bundle adjustment in SkyStitch and noticed that it has a few limitations. First, bundle adjustment requires pairwise matching information to be available. This is not always possible because pairwise stitching may fail and have to be predicted by optical flow, in which case bundle adjustment is unusable. Second, bundle adjustment does not take any temporal information into consideration so perspective jerkiness can be introduced between consecutive frames. Our loop closing approach works even when pairwise feature matching information is missing; there is very little jerkiness since cameras are smoothly and coherently adjusted. Even though we cannot prove that our approach will always reduce reprojection errors, we found that in practice, it introduces less perspective jerkiness and is much faster, while visually, it performs as well as bundle adjustment for small loops.

Table 1: Major components used in our quadcopter. The total weight (including components not listed) is 2.1 kilograms, and the total cost is about USD1,200.

Components	Model
Flight controller	APM 2.6
GPS receiver	uBlox LEA-6H
Motor	SunnySky X3108S
Electronic speed controller	HobbyWing 30A OPTO
Propeller	APC 12×3.8
Airframe	HobbyKing X550
On-board computer	NVIDIA Jetson TK1
Camera	PointGrey BlackFly 14S2C
Lens	Computar H0514-MP2
WiFi adapter	Compex WLE350NX

4. SYSTEM IMPLEMENTATION

The implementation of the SkyStitch system involves the integration of UAV design, computer vision and wireless communication. In this section, we describe the key implementation details of SkyStitch.

While there were commercial quadcopters available in the market, we chose to assemble our own quadcopter because it offered the greatest flexibility for system design and implementation. The components used in our quadcopter assembly are listed in Table 1, and our self-assembled quadcopter is shown in Figure 11. We built two identical quadcopters of the same design.

We used ArduPilot Mega (APM) 2.6 [1] as the flight controller. APM 2.6 has built-in gyroscope, accelerometer and barometer, and both its hardware and software are open source. The on-board computer is a 120-gram NVIDIA Jetson TK1, featuring a 192-core GPU and a quad-core ARM Cortex A15 CPU. The GPU is dedicated for feature extraction, while the CPU is mainly used for data transmission and optical flow computation. Jetson TK1 also has a built-in hardware video encoder, which we used for H.264 encoding. The Jetson TK1 board runs Ubuntu 14.04.

Each quadcopter is equipped with a PointGrey BlackFly camera, which has a $^1/_3$ inch sensor with 1280×1024 resolution. The external lens we added to the camera has a 62.3° diagonal field of view.

To avoid seam artefacts caused by the imaging process, we applied the same exposure parameters to the cameras and each quadcopter periodically broadcasts its white balance values to neighbors so that videos from both quadcopters have consistent colors.

The ground station consists of an Advantech MIO-2260 board, which acts as a wireless router, and a commodity computer that performs video stitching. The computer, which runs Ubuntu 12.04, has a quad-core 3.4 GHz Intel CPU and is connected to the router through Ethernet. Our implementation uses only a single thread to do video stitching, while the rest of the CPU resources are used for other tasks such as video decoding. The computer also has a GTX 670 graphics card which has 1,344 CUDA cores, much higher than the 192 CUDA cores in the Jetson TK1. We also implemented seam finding and image blending on the GPU. The seam finder is based on Voronoi diagram and is implemented in OpenCL. We adopted the classical multi-resolution approach for image blending and implemented it in OpenGL shaders. We optimized these two stages so that it takes 6 ms to composite two HD images and 30 ms to composite 12 HD images.

Video Synchronization. An implicit requirement for high-quality video stitching is that the video frames to be stitched should be captured at exactly the same instant. Without accurate video synchronization, a mobile object on the ground would appear at different locations from different cameras, which is commonly known as "ghosting" artefacts. Software-based synchronization methods are intrinsically inaccurate since they do not consider the non-negligible delay from the time when photons hit the sensor to the time when the host computer receives the video frame [19].

In SkyStitch, we adopt a hardware-based approach for video synchronization. Our key observation is that each quadcopter is equipped with a GPS receiver for flight control, which also provides an accurate time source. In particular, we connected the GPS receiver's time pulse pin to the external shutter trigger of the camera, and set the GPS receiver to issue a triggering pulse at desired frame rate. When issuing the very first pulse, the GPS receiver also sends the corresponding UTC time to the on-board computer through the flight controller. In this way, the on-board computer can associate every video frame with an accurate UTC time, which will be transmitted to the ground station along with the corresponding video frame. Offline ground truth measurement using a logic analyzer showed that our hardware-based approach could reduce video synchronization errors down to smaller than 10 μs.

5. EVALUATION

In this section, we evaluate the performance of SkyStitch on both stitching time and stitching quality. We conducted our experiments over two types of ground with different distributions of features. One venue was a grass field in a park, covered by yellow leaves and a few wooden blocks ("Grass field"). Due to the scattered yellow leaves, the images taken over this field were rich in features. These features were similar to each other and could lead to many feature mismatches, thereby degrading stitching quality. The other venue was a running track, which did not have many good features ("Running track").

In our experiments, the two quadcopters hovered at an altitude of 20 m, equivalent to the height of a 6-storey building. We did not typically fly higher for safety reasons. That said, we did some simple experiments to verify that altitude did not have a significant effect on performance. The benchmarks we used for comparison are the popular OpenCV implementations [7] of commonly-used stitching algorithms [26].

5.1 Stitching Time

The time required to stitch a pair or batch of frames directly determines the maximum frame rate (or stitching rate) that a stitching algorithm can support. A higher frame rate generally results in better user experience. On the other hand, given a particular frame rate, a shorter stitching time also allows a stitching algorithm to support more video sources. In this section, we show that SkyStitch achieves shorter stitching times than existing methods.

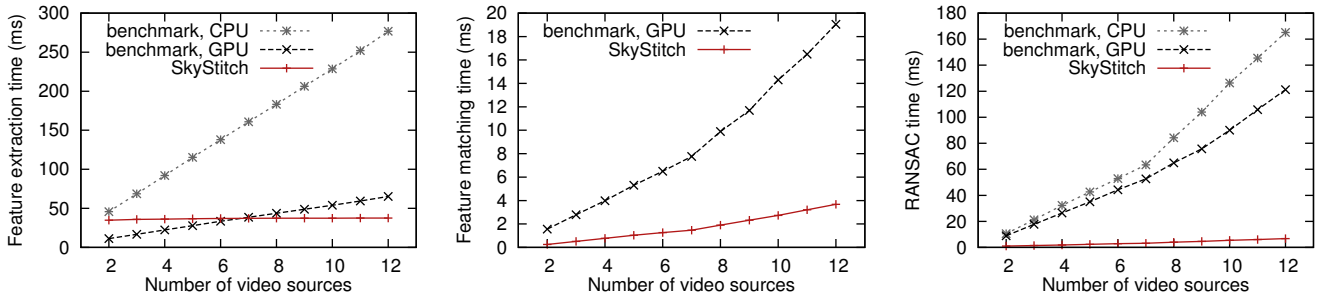
The 12 video traces used in our evaluation were collected by our quadcopters in a 2×6 grid pattern (for the Grass field venue). The resolution of the traces is 1280×1024 and the overlap between adjacent traces is about 80%. The duration of each video trace is 15 seconds at a frame rate of 20 fps. We ran both the CPU-based and GPU-based implementations of the benchmarks at the ground station. One exception is that we used the GPU-based implementation in [20] to benchmark RANSAC, for which OpenCV lacks the GPU-based implementation. We selected OpenCV's BruteForceMatcher method (instead of the KD-tree method) as a benchmark for feature matching, as we found the KD-tree method has little speed advantage when feature database is small. The number of iterations in all the RANSAC implementations was set to 512. The CPU in the ground station is a 3.4 GHz quad-core CPU, and the GPU is a GTX 670 graphics card with 1,344 cores (see Section 4).

Execution time of individual step. We varied the number of video sources N from 2 to 12 and measured the execution time of each individual step. Figure 12(a) compares the average execution time of feature extraction for SkyStitch to the two benchmarks (i.e., the CPU-based and GPU-based implementations). The standard deviation of the results are not included as they are very small. As expected, we find that SkyStitch is scalable to many video sources because feature extraction is offloaded to each quadcopter. On the other hand, the execution time of benchmarks is proportional to the number of video sources. Even though the GPU benchmark uses a more advanced GPU (1,344 cores) than SkyStitch (192 cores), it has a worse execution time than SkyStitch when the number of video sources is large ($N > 6$).

Figure 12(b) illustrates the execution time of feature matching. SkyStitch is approximately 4 times faster than the GPU-accelerated benchmark and up to 70 times faster than the CPU benchmark (not shown in the figure as it is too slow). Such large gain of SkyStitch over CPU benchmark is mostly due to the reduction of the search space, as feature matching in SkyStitch runs in CPU as well. We believe that a GPU implementation of feature matching in SkyStitch could further reduce its execution time. Note that when $N > 7$, the slopes of both methods become steeper. The reason is that our 12 video traces are in a 2×6 grid pattern and more pairwise stitchings are needed when processing the second row of video sources.

The execution time of RANSAC is shown in Figure 12(c). SkyStitch is up to 18 times faster than the GPU benchmark and 24 times faster than the CPU benchmark. Surprisingly, the gap between the GPU benchmark and the CPU benchmark is not large. One reason is that a typical SVD solver (e.g., Jacobian SVD) does not run efficiently on GPU cores and the speedup due to high parallelism is partially offset by its poor performance per core. Another reason is that sequentially performing RANSAC for each image pair requires many CPU-GPU memory transactions and thus introduces significant IO overhead. In contrast, SkyStitch's RANSAC GPU pipeline is much more efficient. Even with 12 video sources, its execution time is below 10 ms.

Stitching rate. We calculate stitching rate by taking the reciprocal of the total execution time for stitching a batch of frames. Let $T_{extract}$ denote feature extraction time and T_{others} denote the total



(a) Feature extraction. (b) Feature matching. (c) RANSAC.

Figure 12: Execution time of each step of video stitching. The number of features extracted from every frame is 1,000.

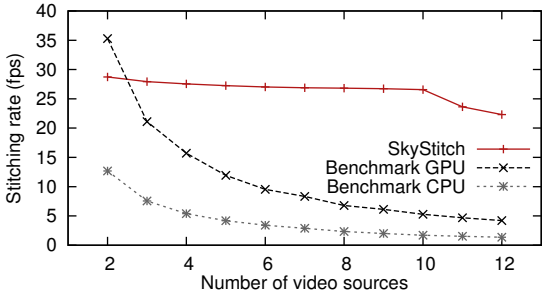
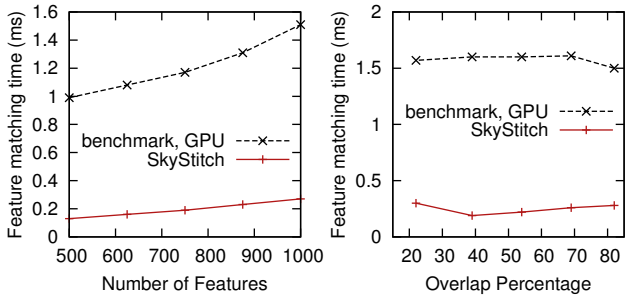


Figure 13: Stitching rate (in terms of frames per second) with respect to the number of video sources.



(a) Impact of feature count. (b) Impact of overlap.

Figure 14: Impact of feature count and overlap percentage on the execution time of feature matching.

time of the remaining steps (i.e., feature matching, RANSAC and image compositing). Then the execution time per stitching is equal to $max(T_{extract}, T_{others})$ for SkyStitch and $T_{extract} + T_{others}$ for the benchmarks.

Figure 13 shows that SkyStitch achieves a stitching rate up to 4 times faster than the GPU benchmark and up to 16 times faster than the CPU benchmark. The stitching rates of both benchmarks drop sharply as N increases due to the fact that all computations are performed at the ground station. SkyStitch, however, offloads feature extraction to the UAVs to greatly reduce the workload at the ground station. When $N \leq 10$, SkyStitch's stitching rate is solely determined by the feature extraction time, which is around 28 fps. When $N > 10$, the bottleneck of SkyStitch shifts to the ground station and the stitching rate starts to drop. Nevertheless, we estimate that SkyStitch will be able to support a stitching rate of 22 fps when $N = 12$ while the two benchmarks could not maintain such stitching rate when $N > 3$.

Impact of feature count and overlap percentage. In the above experiments, the number of extracted features is fixed at 1,000 and the overlap between two adjacent video sources is about 80%. We found that these two factors do not affect stitching time except the time of feature matching. Figure 14(a) shows the impact of feature count on the time of feature matching, for the case of two video sources. We can see that the feature matching time of both SkyStitch and the GPU benchmark increases with feature count, but the rate of increase for SkyStitch is slower. This is expected since the GPU benchmark requires pairwise comparison for all the features while SkyStitch only needs to search in a small area for the potential match.

The impact of overlap percentage on feature matching time is shown in Figure 14(b). Generally the time of SkyStitch increases with overlap since more features in the overlapping area need to be matched. The exception is at the overlap of 20%, where feature matching time does not follow the trend. Upon closer inspection, we found that it is because when the overlap is small there are more stitching failures which lead to a larger search radius r (see Section 3.4.3). The feature matching time of the GPU benchmark appears to be independent of the overlapping area and is much longer than SkyStitch.

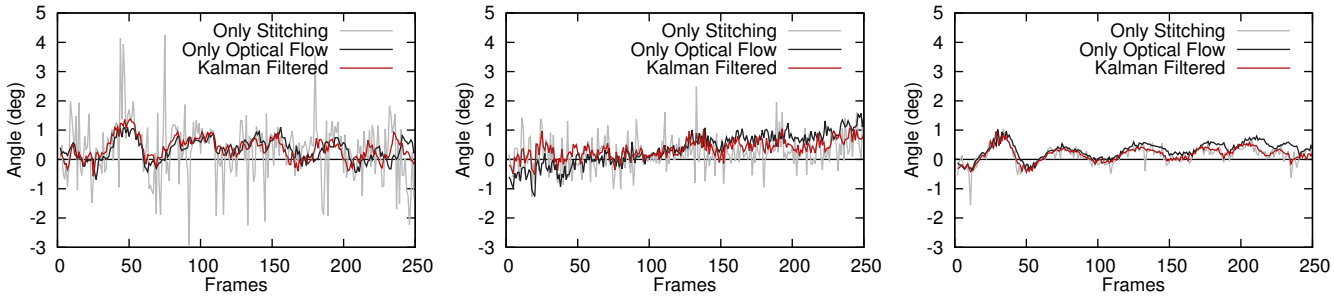
5.2 Stitching Quality

In this section, we show that SkyStitch produces high-quality stitching results. In particular, we evaluate the number of inliers, the stitching success rate and the degree of frame jerkiness and also the incidents of ghosting artefacts. Sample videos of stitching can be found in [4].

Number of inliers. The number of inliers is a basic measure of goodness for stitching algorithms. More inliers generally leads to a better $\mathbf{H}$ estimate. Figure 16 shows the number of inliers at different degrees of overlap. For both types of ground, SkyStitch generates more inliers than the benchmark. The reason is that the feature matching step in the benchmark uses pairwise comparison among all the features and the probability of false matching is high. In contrast, the feature matching in SkyStitch only searches in a small target area and thus it is less likely to have false matching. From Figure 16 we can also see that the slope of Grass field is steeper than that of Running track. This is likely because these two types of ground have different feature distributions.

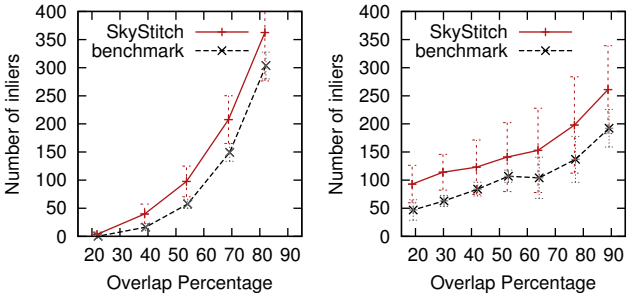
Success rate of stitching. In Section 3.4, we define the failure of stitching (or ill-formed $\mathbf{H}$) based on the number of inliers and the angle θ. In SkyStitch, failed stitching can be potentially recovered using optical flow. We calculate the success rate of stitching for the above experiment, as shown in Figure 17. For both types of ground, we can see that SkyStitch has higher success rate than the benchmark, because of the recovery using optical flow.

Frame jerkiness. Stitching every frame independently could introduce perspective jerkiness which degrades visual quality. To quantify perspective jerkiness and show how our approach can reduce jerkiness, we measure the rotation component in the homography matrix, which is directly related to camera perspective. To



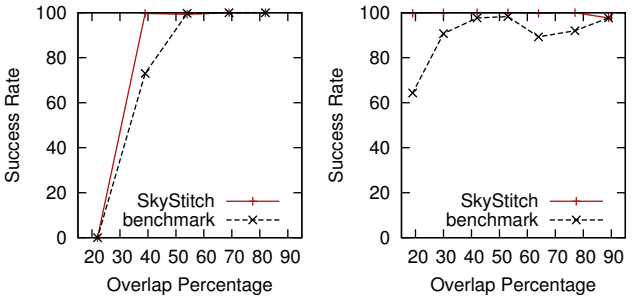
(a) Roll (b) Pitch (c) Yaw

Figure 15: Jerkiness of rotation components in computed homographies



(a) Grass field. (b) Running track.

Figure 16: Impact of overlap percentage on the number of inliers. The number of features extracted from every frame is 1,000.

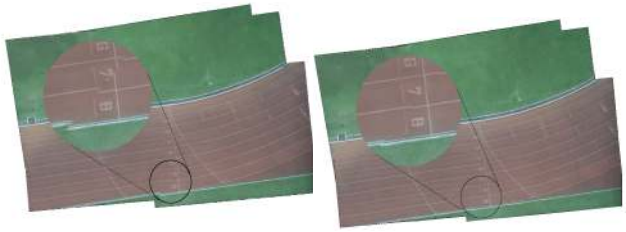


(a) Grass field. (b) Running track.

Figure 17: Impact of overlap percentage on success rate.

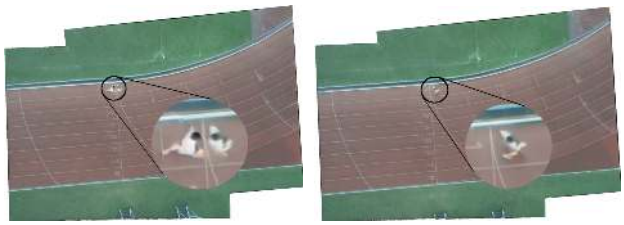
make things more intuitive, we convert the rotation matrix into Euler angles yaw, pitch and roll. We stitched a pair of video sequences (40% overlap) in the Running Track scene. Figure 15 shows the Euler angles extracted in the estimated homographies. Note that "Only stitching" produces very significant jerkiness in pitch and roll angles. In contrast, "Only optical flow" method is much smoother but suffers from drift especially in yaw and pitch angles. The "Kalman filtered" angles are not prone to either of the problems. It follows closely the results of "Only stitching" while mitigating most of the jerkiness. The perceptual difference can be seen from our demo videos [4].

Multiple video sources. In Section 3.5, we proposed a loop closing method for multi-image stitching. Figure 18 illustrates the effect of alleviating the misalignment between the first and the last image during multi-image stitching. The six images were collected from the Running track scene. We can see in Figure 18(a) that there is a discernible misalignment along the stitching seam between the first and the last image. By applying the loop closing method discussed in Section 3.5, we were able to mitigate the misalignment so that the viewers could hardly notice it (see Figure 18(b)). A sample video can be found in [4].



(a) No loop closing. (b) With loop closing.

Figure 18: Effect of loop closing for 6-image stitching.



(a) Unsynchronized frames. (b) Synchronized frames.

Figure 19: Effect of video synchronization on ghosting artefacts.

We follow a similar method to that in Section 3.4 to determine the success rate of multi-image stitching. Specifically, if any pairwise stitching is found to have an ill-formed **H**, we consider the multi-image stitching as a failure. Using the above trace with six video sources, we found that the success rate of SkyStitch is about 97%, while the success rate of the benchmark is only 32%.

Elimination of Ghosting Artefacts. In Section 4, we explained that our hardware-based method for video synchronization is important to ensure good stitching quality when there are mobile objects. Figure 19 illustrates the effectiveness of our synchronization method. The source images were taken from two quadcopters with hardware-based synchronization, while a man was jogging on the running track. Figure 19(a) is stitched from two video images that are 200 ms apart in capturing time. We can see a clear ghosting artefact around the stitching seam, i.e., two running men instead of one. In practice, such ghosting artefact will affect the viewers' assessment of the actual situation on the ground. In Figure 19(b), the image is stitched from two video frames synchronized by GPS. We can see that our hardware-based synchronization method is sufficiently accurate to eliminate distracting ghosting artefacts.

6. CONCLUSION

In this paper, we present the design and implementation of SkyStitch, a multi-UAV-based video surveillance system that supports real-time video stitching. To improve the speed and quality of video stitching, we incorporated several practical and effective techniques

into SkyStitch, including feature extraction offloading, utilization of flight status information, efficient GPU-based RANSAC implementation and jerkiness mitigation by Kalman filtering. By implementing SkyStitch using a pair of quadcopters, we demonstrate that it can achieve stitching rate 4 times faster than state-of-the-art GPU accelerated methods, while ensuring good stitching quality.

Although SkyStitch is developed for quadcopters, its design principles and techniques can also be applied to other types of hovering aircraft such as blimps [14] which are more vulnerable to wind but have longer flight time. In addition, SkyStitch is currently only a proof-of-concept prototype and there is significant room for future work. For example, we would like to understand how other types of features (like color feature [29] or low-dimensional feature [21]) affect the performance of SkyStitch; it is also interesting to see how SkyStitch performs in challenging environments such as over rough ground with buildings.

7. ACKNOWLEDGMENTS

This work was supported by the Singapore Ministry of Education grant T1 251RES1204.

8. REFERENCES

[1] ArduPilot Mega 2.6. http://ardupilot.com.

[2] Assisting firefighting with a quadcopter. http://tinyurl.com/o5gxvmv.

[3] Civilians with drones lead charge in search and rescue. http://tinyurl.com/oq3zcob.

[4] Demo videos for SkyStitch. http://tinyurl.com/oz8ftc4.

[5] Hugin, panorama photo stitcher. http://hugin.sourceforge.net.

[6] Kolor. http://www.kolor.com.

[7] OpenCV. http://opencv.org.

[8] TCOM's aerostat systems help U.S. border patrol. http://tinyurl.com/ovnwgbr.

[9] M. Adam, C. Jung, S. Roth, and G. Brunnett. Real-time stereo-image stitching using GPU-based belief propagation. In *Proceedings of the Vision, Modeling, and Visualization Workshop, 2009*, Nov. 2009.

[10] D. Anguelov, C. Dulong, D. Filip, C. Frueh, S. Lafon, R. Lyon, A. Ogale, L. Vincent, and J. Weaver. Google street view: Capturing the world at street level. *Computer*, 43(6):32–38, 2010.

[11] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. In *Proceedings of ECCV '06*, May 2006.

[12] A. Brown, C. Gilbert, H. Holland, and Y. Lu. Near real-time dissemination of geo-referenced imagery by an enterprise server. In *Proceedings of the GeoTec Event*, Jun. 2006.

[13] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[14] M. Burri, L. Gasser, M. Kach, M. Krebs, S. Laube, A. Ledergerber, D. Meier, R. Michaud, L. Mosimann, L. Muri, et al. Design and control of a spherical omnidirectional blimp. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '13)*, Nov. 2013.

[15] J. de Villiers. Real-time photogrammetric stitching of high resolution video on COTS hardware. In *Proceedings of International Symposium on Optomechatronic Technologies (ISOT '09)*, Sep. 2009.

[16] A. Eden, M. Uyttendaele, and R. Szeliski. Seamless image stitching of scenes with large motions and exposure differences. In *Proceedings of CVPR '06*, Jun. 2006.

[17] M. El-Saban, M. Izz, and A. Kaheel. Fast stitching of videos captured from freely moving devices by exploiting temporal redundancy. In *Proceedings of ICIP '10*, Sep. 2010.

[18] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[19] R. Hill, C. Madden, A. v. d. Hengel, H. Detmold, and A. Dick. Measuring latency for video surveillance systems. In *Proceedings of Digital Image Computing: Techniques and Applications (DICTA '09)*, Dec. 2009.

[20] N. Ho. GPU RANSAC homography. http://nghiaho.com/?page_id=611.

[21] G. Hua, M. Brown, and S. Winder. Discriminant embedding for local image descriptors. In *Proceedings of ICCV '07*, Oct. 2007.

[22] A. Kaheel, M. El-Saban, M. Refaat, and M. Ezz. Mobicast: a system for collaborative event casting using mobile phones. In *Proceedings of the International Conference on Mobile and Ubiquitous Multimedia (MUM '09)*, Nov. 2009.

[23] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[24] E. Malis and M. Vargas. Deeper understanding of the homography decomposition for vision-based control. 2007.

[25] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: an efficient alternative to SIFT or SURF. In *Proceedings of ICCV '11*, Nov. 2011.

[26] R. Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends in Computer Graphics and Vision*, 2(1):1–104, 2006.

[27] M. Tennoe, E. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, V. R. Gaddam, D. Johansen, C. Griwodz, and P. Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. In *Proceedings of the IEEE International Symposium on Multimedia*, Dec. 2013.

[28] N. Trawny and S. I. Roumeliotis. Indirect Kalman filter for 3D attitude estimation. *University of Minnesota, Dept. of Comp. Sci. & Eng., Tech. Rep*, 2, 2005.

[29] K. E. van de Sande, T. Gevers, and C. G. Snoek. A comparison of color features for visual concept classification. In *Proceedings of the International Conference on Content-based Image and Video Retrieval*, Jul. 2008.

[30] D. Wischounig-Strucl, M. Quartisch, and B. Rinner. Prioritized data transmission in airborne camera networks for wide area surveillance and image mosaicking. In *Proceedings of CVPRW '11*, Jun. 2011.

[31] W. Xu and J. Mulligan. Panoramic video stitching from commodity HDTV cameras. *Multimedia Systems*, 19(5):407–426, 2013.

[32] S. Yahyanejad, D. Wischounig-Strucl, M. Quaritsch, and B. Rinner. Incremental mosaicking of images from autonomous, small-scale UAVs. In *Proceedings of the IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS '10)*, Aug. 2010.