

Panoramic: FPGA Image Stitching System

Davut Muhammetgulyyev

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

davutm@mit.edu

Alianis Lopez

Department of EECS

Massachusetts Institute of Technology

Cambridge, MA, USA

alianisl@mit.edu

Lucas Boo

Department of Mathematics

Massachusetts Institute of Technology

Cambridge, MA, USA

lucasboo@mit.edu

Abstract—We present **Panoramic**, an FPGA-based image stitching system capable of detecting, describing, and matching features to generate panoramic images in hardware. Our design implements a full computer vision pipeline, including multi-scale Harris corner detection with non-maximum suppression, steered BRIEF descriptor, brute-force Hamming distance matching, RANSAC-based homography, inverse homography computation, and bilinear-resampled backward warping followed by overlap blending. The system is aimed to achieve lower latency and power consumption compared to CPU implementations. The architecture can be extended to embedded applications that require real-time panoramic stitching, such as robotics, drones, and augmented reality headsets.

Index Terms—FPGA, image stitching, Harris Corner Detection, BRIEF, CORDIC, RANSAC

I. OVERVIEW

Panoramic image stitching requires combining multiple overlapping frames into a single wide-field view. Traditional stitching pipelines are based on software running on CPUs or GPUs. However, these computation structures introduce latency, power consumption, and memory challenges in real-time embedded applications. The goal of this project is to implement a hardware-accelerated feature-based stitching system on a FPGA.

Our system takes two consecutive camera frames as input and performs three main tasks, which form subsystems:

- *Feature Detection*
- *Feature Description and Matching*
- *Image Transformation and Mixing*

Due to multiple modules needing to write to and read from SDRAM memory, we created a new module `memrequest_handler` to deal with the traffic. The module is created once in `top_level` and takes all the memory requests of the relevant modules at once through 1-D registers (the position correlates to the module) separated by write requests from `clk_camera`, write requests from `clk_pixel`, and read requests from `clk_pixel`. These inputs are then put in a FIFO within the handler until they are ready to be sent out to the high definition frame buffer. The responses from the frame buffer become inputs that handler then needs to output to the relevant modules with more 1-D registers. Implementing this module allows us to keep most of the logic of the high definition frame buffer intact (including the traffic generator) with some slight rearranging and moving.

II. FEATURE DETECTION (LUCAS)

In this stage, the goal is to take the raw image and detect feature points, which are points in the image which are relatively resilient against translation and rotation. Here, we use corners as feature points, as detected via Harris Corner Detection described below.

A. Memory Usage

Throughout the feature detection process, there are several stages in which we need the grayscale values in a small window around a pixel. To achieve this, we generally use a sliding window architecture, where BRAM is used to store slices of the previous rows, and LUTRAM then provides a ‘window’ which can be used for processing.

B. Pre-Processing

Before the image is processed, we downscale it by a factor of 2 twice (for an overall 4x downscale), applying Gaussian blur filters in each stage to minimize aliasing effects. We also convert the image to grayscale by combining the red, green and blue channels in a 1 : 2 : 1 proportion. The resulting image is independently saved into DRAM as feature description will rely on the same image.

C. Harris Score

To achieve feature detection, we use Harris Corner Detection to identify feature points. The method uses a 7 by 7 window around each pixel (pixels near the edge are not considered) to determine if the pixel corresponds to a corner. The Harris Score computation for a given neighborhood is computed as follows:

- For each pixel in a 5 by 5 neighborhood around the pixel, we compute the (signed) difference I_x between the pixels immediately above and below it, as well as the difference I_y between the pixels immediately to the left and right of it.
- For each pixel, we compute $I_x^2, I_y^2, I_x I_y$. These values are then summed over all 25 pixels to give $I_{xx} = \sum I_x I_x, I_{yy} = \sum I_y I_y, I_{xy} = \sum I_x I_y$.
- We can now form the matrix $M = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{pmatrix}$, and calculate the Harris Score as $\det M - \lambda \operatorname{tr} M$, where $\lambda = \frac{5}{128}$ (λ is generally chosen within the range $[0.04, 0.06]$, we choose

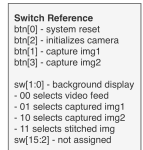
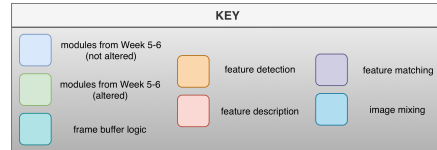
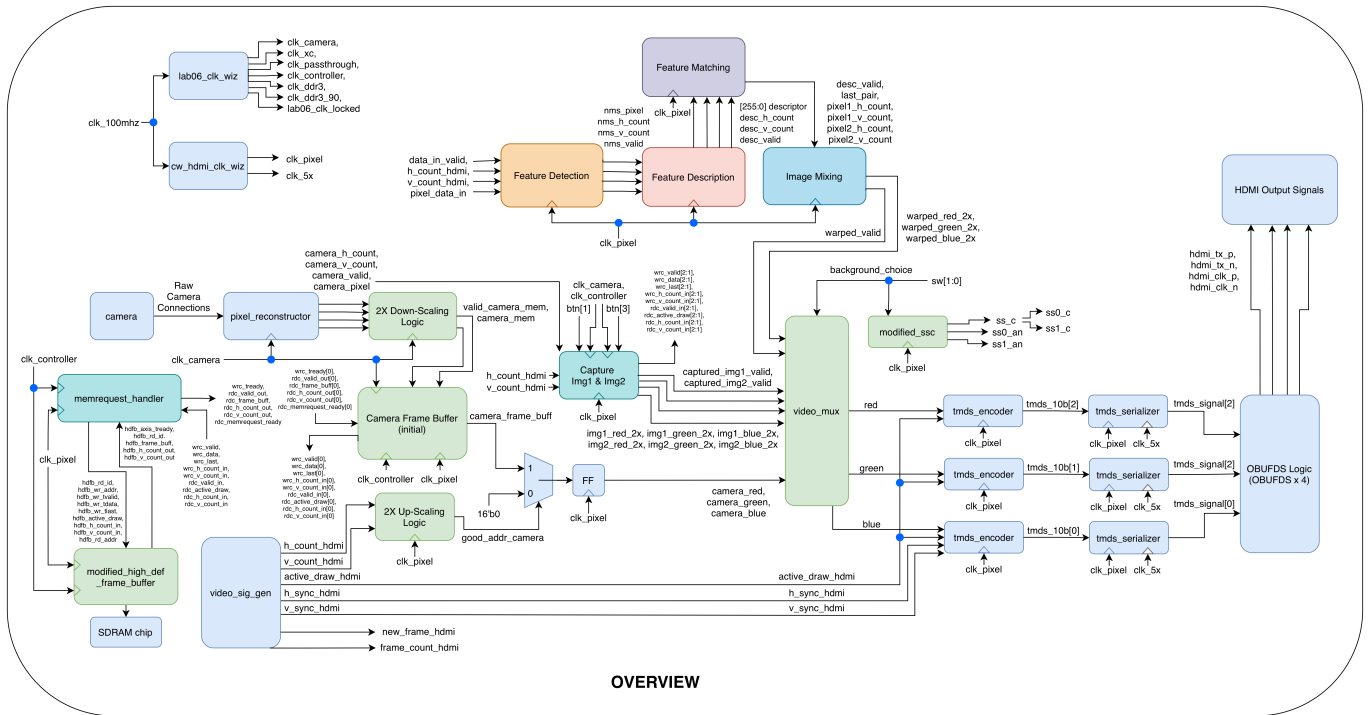


Fig. 1: Overall block diagram of the system

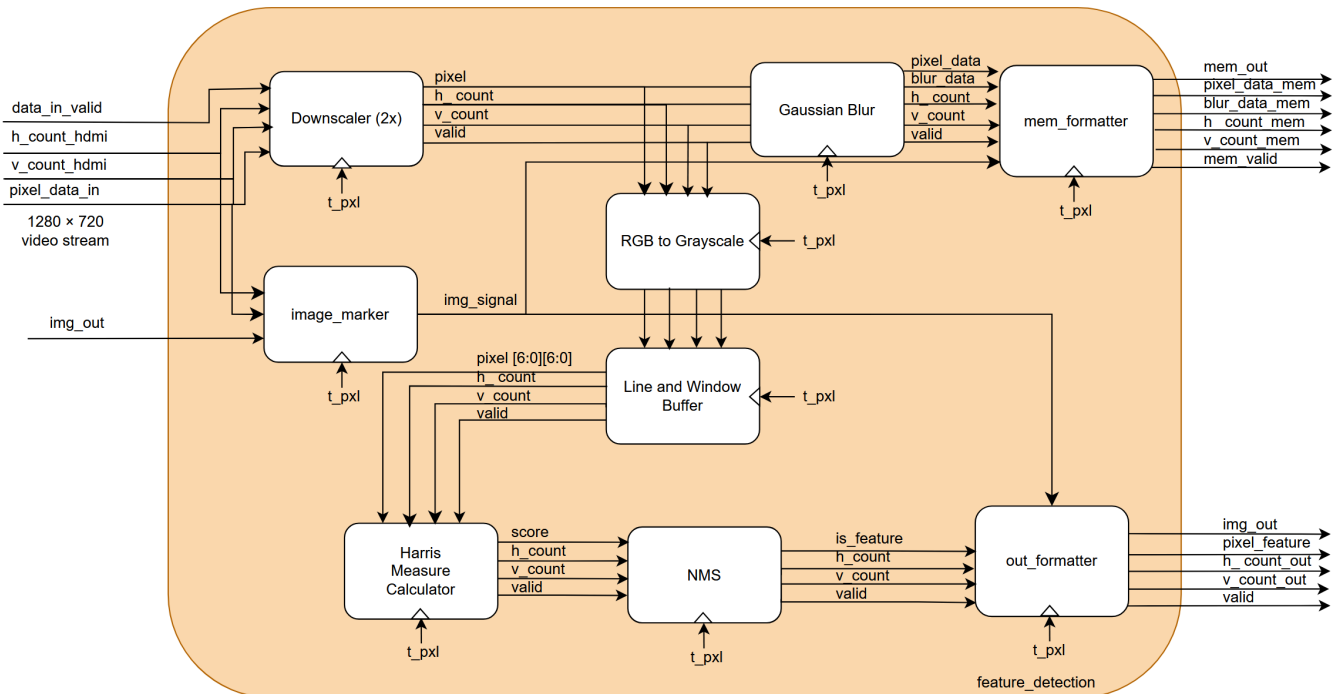


Fig. 2: Block diagram of the detection module

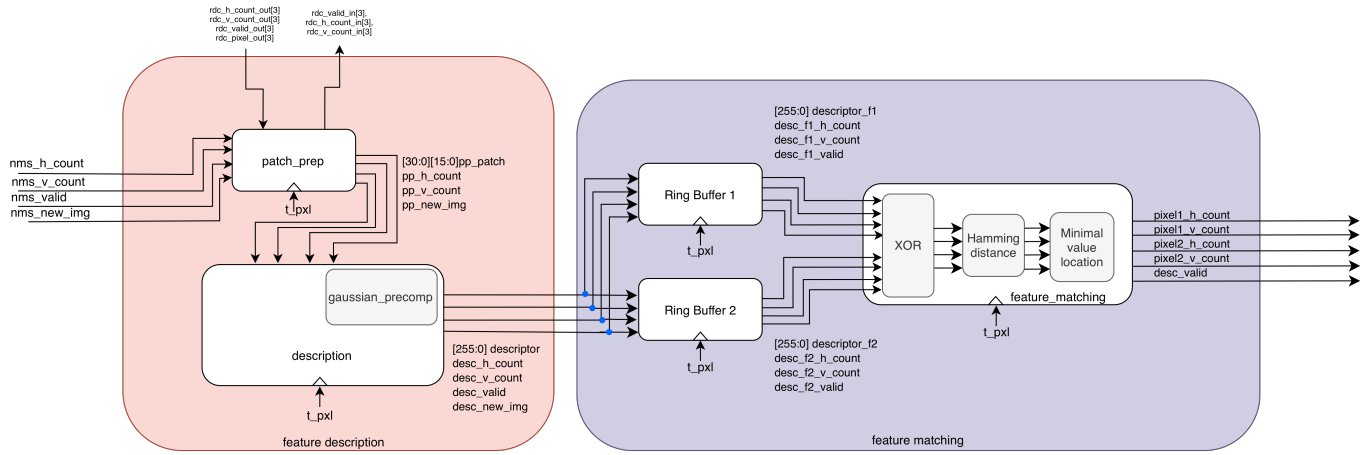


Fig. 3: Block diagram of the description module

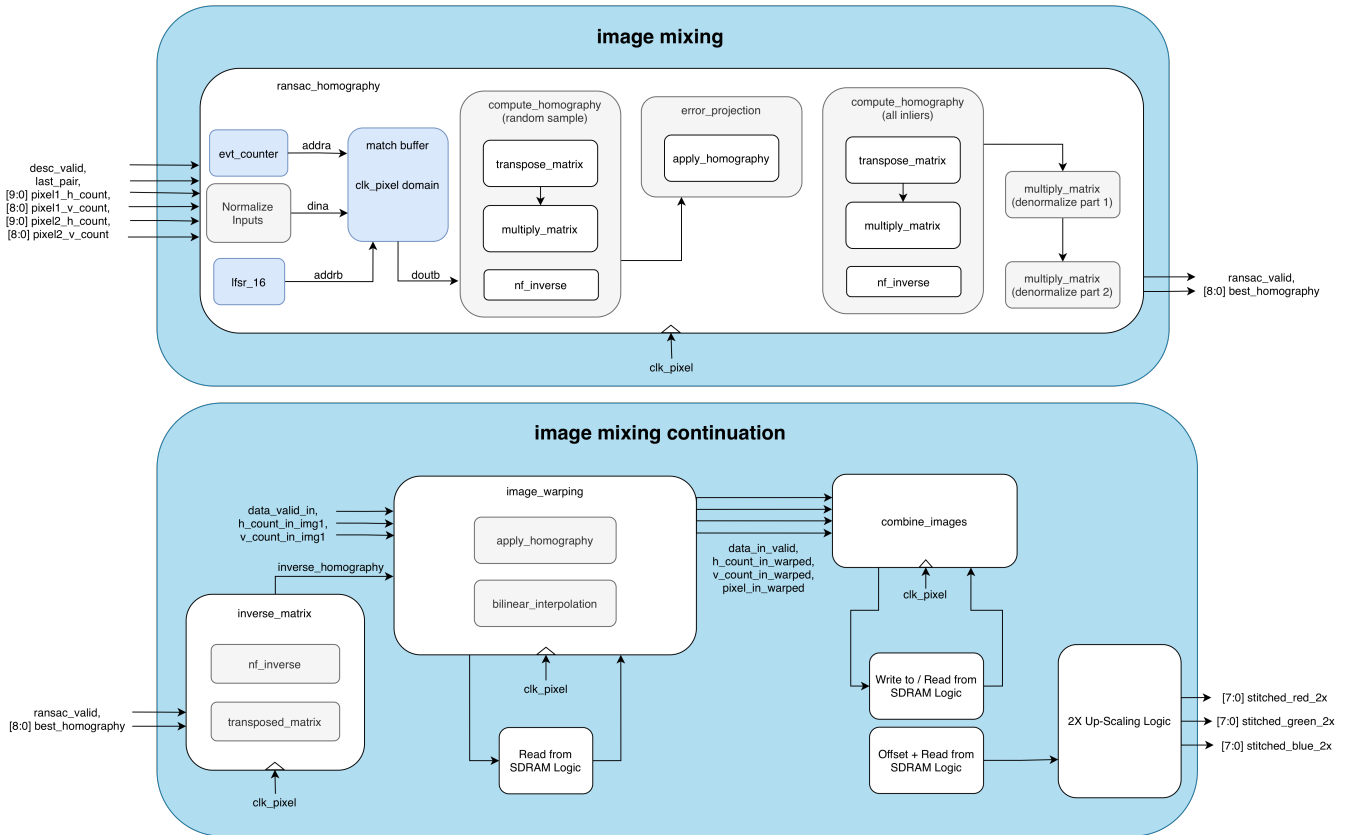


Fig. 4: Block diagram of the Image Mixing

this value since it is easy to multiply by via bit shifting). A high positive Harris score corresponds to a corner feature.

D. Thresholding

The Harris score is compared to a threshold to determine whether it is sufficiently positive to register as a corner. Ideally, we want the number of features output to be roughly 200, however, depending on factors such as the contrast and composition of the input image we cannot achieve this in general with a fixed threshold value. Hence, we take advantage of continuous camera input into the detection module to adjust the threshold continuously: by counting the number of feature points obtained on each frame, we can increase or decrease the threshold if the feature count lies outside of our desired range.

E. Non-Maximum Suppression

Due to the window size of the Harris Corner detection method, a single corner feature will often register in a cluster of points around the feature. Hence, in order to avoid excessive duplicated features, we implement a form of non-maximal suppression (NMS). However, in contrast to regular NMS, our goal is primarily to ensure that the feature pixels are at least some distance apart. We achieve this by keeping a sliding window of the previous four rows, tracking which pixels were previously determined as feature points. A new candidate point is required to not have any previous feature points in a half-square above it, or to the left on its current row, to be regarded as a feature point itself. Since we only store one bit of data per pixel for a few thousand pixels (corresponding to the previous four rows), we can use LUTRAM as the total load is relatively low.

However, a limitation of this method is that we lose out on the benefit of NMS of retaining local maxima which more accurately correspond to the feature point.

F. Inputs to SDRAM

The feature detection module is also responsible for generating downscaled images for later use. Two images are saved: The first is the 2x downscaled image (for eventual use in image stitching), and the second has a Gaussian blur applied for use in the description module.

III. FEATURE DESCRIPTION (DAVUT)

We implement **Binary Robust Independent Elementary Features (BRIFT)** algorithm to generate descriptors that will be used in matching stage.

A. Buffering NMS points

We queue the incoming points of interest (POI) from NMS into FIFO, keeping track of `nms_h_count`, `nms_v_count`, and `nms_new_img` signals, to avoid losing POI as the later stages are blocking and will ignore the inputs while ready signal is set to low.

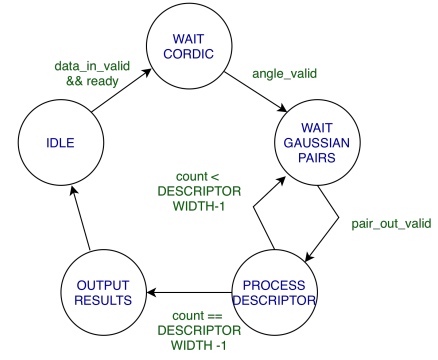


Fig. 5: Overall FSM of description. Note that rst signals are omitted and assumed every state transitions to IDLE in such case.

B. Patch Building and Momentum

When the description module raises ready signal to high, we pull one buffered NMS point from the FIFO and start assembling 31x31 patch of Gaussian smoothed pixels centered at that point. The smoothed image was saved to SDRAM in the detection stage, thus we issue a read request to SDRAM and collect the pixels one by one in unpacked array. We also compute the central moments of the patch (m_{10} and m_{01}), which are used in the later stages of the pipeline to determine the dominant orientation, on the fly by the formula [1]:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \quad (1)$$

where $p, q \in \{0, 1\}$ and $I(x, y)$ is the intensity of the pixel at (x, y) , while center is assumed to be $(0, 0)$.

Once we build 31x31 patch, we pass most importantly m_{10} , m_{01} , patch, `h_count`, `v_count` and other signals to the description module, which works based on FSM (see Fig. 5).

C. CORDIC

After we receive valid patch, all of the inputs gets saved, and the FSM transitions to WAIT_CORDIC and start calculating the dominant orientation angle.

To compute dominant orientation of the patch, we use **COordinate Rotation Digital Computer (CORDIC)** algorithm in vectoring mode [2] [3]. The objective is to compute $\text{atan2}(m_{01}, m_{10})$ and return 8 bit circular fixed point format, where values $0 \rightarrow 255$ uniformly map onto $0 \rightarrow 2\pi$. We initialize a lookup table for $\arctan(2^{-i})$ for $i \in [0, 7]$.

Our implementation uses 10 stage pipeline: in the first stage, we store the quadrant information and take absolute values of input vector (x_0, y_0) , afterwards normalizing it by 2^8 . This scaling value was chosen such that it is highest possible power of two that will not overflow. Later at each stage i , a micro-rotation of $\arctan(2^{-i})$ is applied using only addition and arithmetic shifts:

$$x_{i+1} = x_i - \sigma_i 2^{-i} y_i \quad (2)$$

$$y_{i+1} = y_i + \sigma_i 2^{-i} x_i \quad (3)$$

$$z_{i+1} = z_i - \sigma_i \arctan(2^{-i}) \quad (4)$$

$$\sigma_i = \begin{cases} 1 & \text{if } y_i < 0 \\ -1 & \text{if } y_i \geq 0 \end{cases} \quad (5)$$

In the last stage of the pipeline, using the saved quadrant information, we remap the angle to be in the range 0 to 255.

Once we have valid angle, we raise `angle_valid` one cycle high, and our FSM state transitions to `WAIT_GAUSSIAN_PAIRS`.

D. Gaussian Pairs Calculation

BRIEF requires to sample and test pairs of `DESCRIPTOR_WIDTH`, which is 128, 256, or 512, number of points in 31x31 patch from Gaussian distribution. However, computing Gaussian distribution in FPGA is expensive, thus, we pre-compute 256 possible pairs, and for each of 256 pairs, we also further pre-compute 32 different rotations of them to support steered-BRIEF, expanding our descriptors to be moderately rotation invariant, and store everything in Xilinx RAM, using the IP introduced in Lab4.

Now as we have dominant orientation from CORDIC submodule, we can select the rotated pair. Each line in the mem file contains concatenated binaries in format $(row_1, col_1, row_2, col_2)$. This submodule has two cycle latency posed by usage of high-performance mode of Xilinx IP.

Once we have valid Gaussian pairs, the state transitions to `PROCESSOR_DESCRIPTOR`.

E. steered-BRIEF (rBRIEF)

In standard BRIEF description, given a sampling pair, a classification test is performed according to:

$$\tau(p; a, b) = \begin{cases} 1 & \text{if } p(a) < p(b) \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

where, p is a patch, and $p(a)$ is an intensity of pixel at point a in the patch.

This repeats 256 times, and ultimately returned as a concatenated 256 bit descriptor:

$$f_{n_d}(p) := \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(p; a_i, b_i) \quad (7)$$

where $n_d = 256$ is descriptor width.

Steered-BRIEF uses:

$$g_{n_d}(p, \theta) := f_{n_d}(p) | (a, b) \in R_\theta \quad (8)$$

where g is new description function, R_θ is Gaussian pairs rotated by θ [1].

We perform one steered classification at a time and use the shift buffer to store the test results; after, if we haven't yet reached 256 tests, we transition to `WAIT_GAUSSIAN_PAIRS`

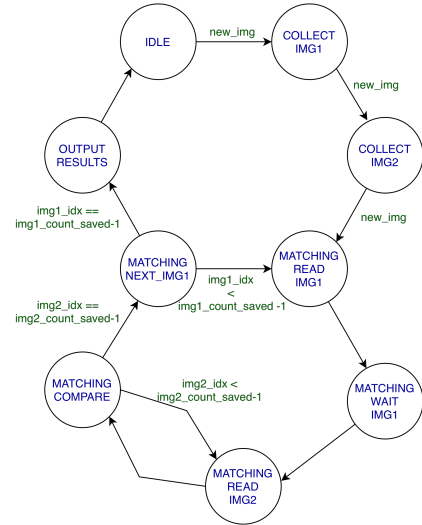


Fig. 6: Overall FSM of matching. Note that `rst` signals are omitted and assumed every state transitions to `IDLE` in such case.

to read the next Gaussian pair and continue description, or if all 256 tests were done, then transition to `OUTPUT_RESULTS`, raising `descrip_valid` one cycle high. `OUTPUT_RESULTS` then transitions to the `IDLE` state clearing all registers and simply acts as reset.

IV. FEATURE MATCHING (DAVUT)

At this stage, we collect and match the described points of two images by brute-force approach based on FSM (see Fig. 6).

A. Buffering Descriptors

As we are using a single instance of description module, we will need to keep track of descriptors, including which image they belong. In addition, we need a support for brute force over descriptors of both images. To achieve that, we implemented `ring_buffer` submodule, which mostly resembles FIFO: this module is intended to be write only first and then read only, cycling indefinitely once the end is reached. It supports single cycle reads, enabling BRAM synthesis rather than combinational reads, which would otherwise use LUTs.

When the first image starts arriving, the FSM transitions to `COLLECT_IMG1` state. In this state, incoming valid descriptors are stored in `ring_buffer_1` along with their `h_count`, `v_count`, and `new_img` signals. When `new_img` goes high for the second time, FSM transitions to `COLLECT_IMG2` and now, similarly to before, valid inputs are collected into `ring_buffer_2`. Once descriptors from both images are fully stored, the FSM moves to the `MATCHING_READ_IMG1`.

This part of the system consumes a significant amount of BRAMs, mainly for ring buffers, specifically $(256 + 11 + 10 + 1) * 300 * 2 = 166.8$ Kbits.

B. Popcount and Matching

We match BRIEF descriptors using Hamming distance, specifically, two descriptors are match if number of the differing bits is the lowest over all:

$$\text{Hamming}(d_1, d_2) = \text{popcount}(d_1 \oplus d_2) \quad (9)$$

$$\text{match}(i) = \arg \min_j \text{Hamming}(d_{A_i}, d_{B_j}) \quad (10)$$

In the FSM, `MATCHING_READ_IMG1` issues read request to the `ring_buffer_1` and transitions to the `MATCHING_WAIT_IMG1`.

During `MATCHING_WAIT_IMG1`, the `best_distance` is initialized to 256, which is the worst possible Hamming distance, where all bits differ, and `img2_idx` is set to 0 to track progress through image 2 descriptors. The current anchor descriptor from image 1 is then compared against every descriptor in image 2. The FSM transitions to `MATCHING_READ_IMG2` state, which reads from `ring_buffer_2`, and then to the `MATCHING_COMPARE`, where the comparison is performed. If the current descriptor yields a smaller Hamming distance than `best_distance`, both `best_distance` and the corresponding image 2 index is updated as now we have better matching pair.

For each anchor descriptor from image 1, if not all image 2 descriptors have been compared, the FSM returns back to `MATCHING_READ_IMG2` to continue. Once all comparisons are done, it moves to the `MATCHING_NEXT_IMG1`, in which we transtion to `MATCHING_READ_IMG1` if we haven't explored every anchor or to `OUTPUT_RESULTS` if we did.

In `OUTPUT_RESULTS`, `matches_out_valid` is raised for one cycle and the FSM returns to `IDLE`.

V. IMAGE MIXING (ALIANIS)

The mixing subsystem handles calculating the transformation from image 1 to image 2, applying said transformation to warp image 1, and then combining the warped image 1 with the original image 2 to form a singular stitched image. The transformation is represented as a 3x3 matrix called a homography, which is used to map one plane to another in computer vision. This particular subsystem deals heavily with helper modules, matrix manipulation, fixed-point arithmetic, and signed numbers.

A. RANSAC

To calculate the geometric transformation, we used the **Random Sample Consensus (RANSAC)** algorithm. This algorithm iteratively samples a small data set to enlarge the set of valid data (inliers), essentially blocking out noisy data (outliers). Specifically, it iteratively samples 4 matched pairs of features to compute a homography and compares how successful the homography is with the previous iterations' best homography's success. In this scenario, success is the amount of matched pairs that are within an acceptable error threshold once the homography has been applied and these

pairs are the inliers of the data. Once all the iterations have been completed, the homography is calculated one final time using all the inlier pairs to improve on the accuracy of the translation. There needs to be at least 4 pairs to calculate the homography since there are 8 unknowns that need to be solved when computing homography. Although there can be more than 4 pairs randomly sampled, increasing the sample size also increases the number of iterations needed, significantly increasing the cycle count. Since RANSAC thrives on smaller sample sizes, selecting 4 pairs is a good compromise in terms of data size, iterations, and accuracy. This module also normalizes the matched pairs before computing the homography. Using image-fixed normalization centered on (1,1) that can be calculated manually and shared between the 2 images due to the shared resolution, the range of values reduce from [0,639] for `h_count` and [0,339] for `v_count` to [0,2] for both. This reduction saves memory and makes it easier to do matrix calculations and scaling.

$$H_SCALE = 2/640 = 1/320 \quad (11)$$

$$V_SCALE = 2/360 = 1/180 \quad (12)$$

$$h_{\text{count_norm}} = h_{\text{count}} \cdot H_SCALE \quad (13)$$

$$v_{\text{count_norm}} = v_{\text{count}} \cdot V_SCALE \quad (14)$$

The final homography has to be denormalized before it can be considered a valid output. Denormalizing the homography involves multiplying the inverse normalizing matrix $T^{(-1)}$ by the normalized homography H_{norm} and then multiplying the resulting matrix by the normalizing matrix T .

$$T_{\text{norm}} = \begin{bmatrix} H_SCALE & 0 & 0 \\ 0 & V_SCALE & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (15)$$

$$T_{\text{norm}}^{-1} = \begin{bmatrix} 640 \cdot 2 & 0 & 640 \cdot 2 \\ 0 & 360 \cdot 2 & 360 \cdot 2 \\ 0 & 0 & 1 \end{bmatrix} \quad (16)$$

$$H = T^{-1} H_{\text{norm}} T \quad (17)$$

FSM (see Fig. 7) states and explanations:

- **COLLECT**: Starting state that normalizes valid input points and saves them to local memory until the last pair

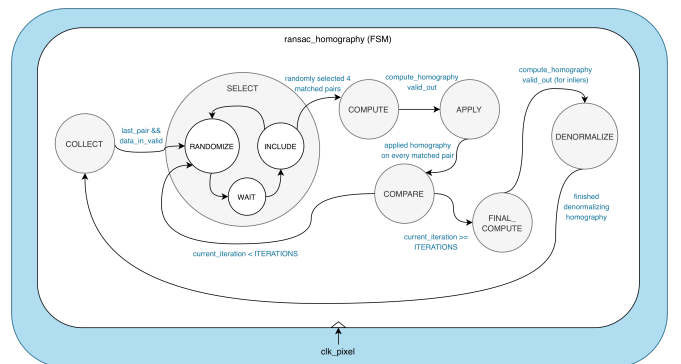


Fig. 7: FSM of RANSAC

of matched pairs have been received. The state moves to SELECT when the last pair has been found.

- **SELECT**: 4 random pairs are selected to use as a sample. This state has an inner FSM for getting the values from memory.
 - **RANDOMIZE**: Starting inner state that randomizes the pair to request from RAM. The inner state immediately moves on to WAIT.
 - **WAIT**: Deliberately has a 1 delay to account for read from RAM before the inner state moves to INCLUDE.
 - **INCLUDE**: Saves the newly read pair as input for computing the homography and keeps track of how many pairs have been requested so far. Once 4 pairs have been selected and read, the inner state resets to RANDOMIZE and the outer state moves to COMPUTE.
- **COMPUTE**: The homography is computed using the random sample. Once the output from the computing helper module is valid, the state moves to APPLY.
- **APPLY**: In this state, every matched pair of features is read from memory and inputted to the error projection module. This helper module applies the homography to the pair and calculates the error of this projection with the euclidean distance formula that is the addition of the squared difference from the projected coordinate (found with the apply homography helper module using the point from image 1) and the point from image 2. If the error found is less than the tolerance threshold, the matched pair is considered a success and becomes an inlier for the current homography. Once all the pairs have been read and applied, the state moves on to COMPARE.

$$\text{projected point} = (x'_1, y'_1) \quad (18)$$

$$\text{image 2 point} = (x_2, y_2) \quad (19)$$

$$\text{error} = (x'_1 - x_2)^2 + (y'_1 - y_2)^2 \quad (20)$$

- **COMPARE**: The current homography is compared to the best homography. If the current one has more inliers than the best one, the best is replaced with the current homography, inlier number, and inliers list. If the current iteration count is less than the max iterations set by the parameter, the state goes back to SELECT; otherwise, it moves on to FINAL_COMPUTE.
- **FINAL_COMPUTE**: Computes the final homography with only the inliers found by best homography of all the iterations. The state moves on to DENORMALIZE once this homography has been calculated.
- **DENORMALIZE**: This state undergoes 2 matrix multiplications to denormalize the homography. Once completed, the now denormalized homography is ready for output and the state goes back to the beginning with COLLECT.

B. Computing Homography

This helper module estimates the homography matrix using the Direct Linear Transformation (DLT) algorithm with a mod-

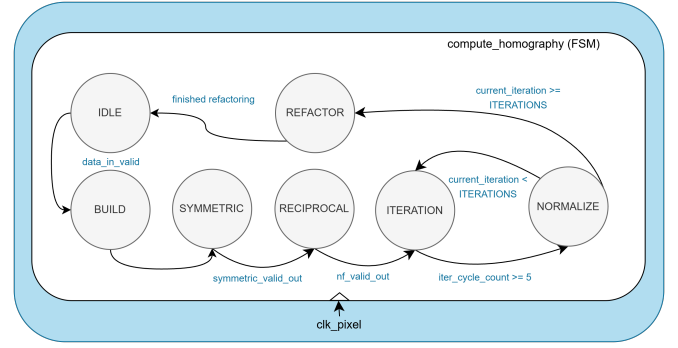


Fig. 8: FSM of Compute Homography

ified linear Jacobi iteration for inverse iterating. DLT is able to estimate a projection matrix by setting up a homogeneous linear system, where each pair of features become two linear equations in matrix A , and solving for $A \cdot h = 0$. With N pairs, A is an $N \times 2 \times 9$ matrix.

$$\text{pair } i : (x_1, y_1) \text{ and } (x_2, y_2) \quad (21)$$

$$\text{row } i \cdot 2 : [-x_1, -y_1, -1, 0, 0, 0, x_2 \cdot x_1, x_2 \cdot y_1, x_2] \quad (22)$$

$$\text{row } i \cdot 2 + 1 : [0, 0, 0, -x_1, -y_1, -1, y_2 \cdot x_1, y_2 \cdot y_1, y_2] \quad (23)$$

based on:

$$-h_1x_1 - h_2y_1 - h_3 + h_7x_2x_1 + h_8x_2y_1 + h_9x_2 = 0 \quad (24)$$

$$-h_4x_1 - h_5y_1 - h_6 + h_7y_2x_1 + h_8y_2y_1 + h_9y_2 = 0 \quad (25)$$

Traditionally, DLT uses Singular Value Decomposition (SVD) to solve the linear equations to get h (9-element vector that refactors to homography H); however, since the only part of SVD that is vital for getting h is the eigenvector corresponding to the smallest eigenvalue, we are choosing a different solving method that prioritizes that calculation solely. For our cases, the method is linear Jacobi iteration that solves for $A^T A h = 0$ by making a guess and iterating until the guess converges. This method has been modified with inverse power iteration to solve for the smallest eigenvalue's eigenvector directly. Without this modification the linear Jacobi iteration would solve for every eigenvector. Since this method can solve for all 9 rows in $A^T A$ without needing the previous row's answer, the rows can be calculated in parallel making the method perfect for FPGA despite taking more iterations to converge than other methods.

$$i \in (0, 8), k \in (0, \text{ITERATIONS}) \quad (26)$$

$$x_{\text{guess}} = x' = x_{k+1}, \quad x_{\text{new}} = x = x_k \quad (27)$$

$$x'_i = \frac{x_i - \sum_{j=1}^8 M[i][j] x_j}{M[i][i]}, \quad j \neq i \quad (28)$$

FSM (see Fig. 8) states and explanations:

- **IDLE**: Starting state that creates copy of input values (ensures values are stable) if inputs are valid.

- **BUILD:** Creates matrix A with all the matched pair inputs. If the number of valid matched pairs are less than the parameter `PAIR_COUNT`, those rows are build with zeroes. Once the matrix has been build, the state moves to **SYMMETRIC**.
- **SYMMETRIC:** First, the transposed matrix of A is calculated. Then, A^T and A are multiplied to form a 9×9 matrix. Once the multiplication have been calculated, the state goes to **RECIPROCAL** and the initial guess for the eigenvector is set as 1.
- **RECIPROCAL:** To prevent from calculating the reciprocal of the diagonals of $A^T A$ within every iteration, the reciprocals are calculated here to later multiply by in the **ITERATION** state.
- **ITERATION:** This state contains all the computations for Jacobi iteration pipelined for finding x_{new} using matrix M and x_{guess} . After 5 cycles, the state moves on to **NORMALIZE**.
- **NORMALIZE:** To minimize error accumulation from all the iterations, x_{new} is normalized by multiplying the reciprocal of $x[8]$ by every element in x_{new} such that $x[8]$ becomes 1. Additionally, x_{guess} is updated using the normalized x_{new} . If the current iteration count is less than the max iterations set by the parameter, the state goes back to **ITERATION**; otherwise, it moves on to **REFACTOR**.
- **REFACTOR:** The output homography is set equal to x_{guess} , making it ready for use. The state goes back to **IDLE**.

C. Inverse Homography

Once the best (most optimal) homography has been calculated through RANSAC, it needs to be inverted so that it maps every destination point to a pixel from the original image rather than mapping every pixel in the image to a destination. Using the inverse prevents visual gaps since every destination point is guaranteed to have a pixel. Since the inverse homography will remain the same when applying it to every destination point, it is calculated only once prior to the warping stage. Computing the inverse of a homography is the same logic as finding the inverse of a 3×3 matrix. We start by finding the determinant. If the determinant is not zero, an inverse exists and we can proceed to find the adjugate matrix of the homography, which is the transposed cofactor matrix. The inverse matrix is equal to the adjugate matrix divided by the determinant, but to avoid division, our code computes the inverse matrix as the adjugate matrix times the reciprocal of the determinant. The reciprocal is found using a helper module that uses the Newton-Raphson method to guess what the reciprocal will be and iteratively improve on the guess, ideally until it converges.

$$x_{n+1} = x_n(2 - x_n \cdot \text{denominator_in}), \quad (29)$$

where n is iterations.

D. Warping

Since we're working with destination points from what would be the final stitched image as inputs, the resolution of

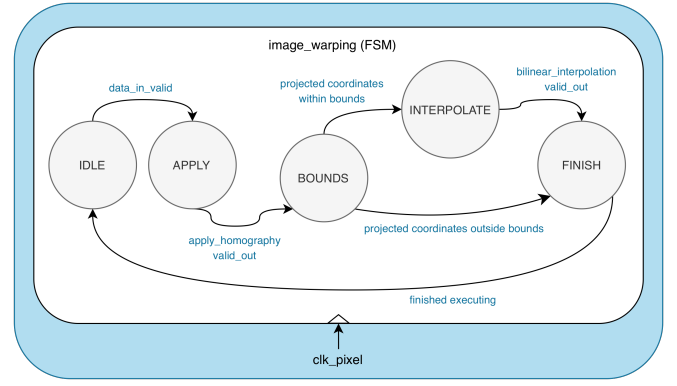


Fig. 9: FSM of Image Warping

the image needs to be large enough to accommodate the image 2 and the warped (translated) image 1 overlapping image 2. As such, the module has parameters in place for adding offsets to the sides of the resolution of image 2. Under the assumption that image 2 will either be translated to down and/or to the right from image 1, we are setting the offset parameters such that image 2 is on the bottom right and there is enough space around it for image 1 to comfortably be even with minimal overlap in the images. To be specific, the total resolution will be 1280×720 , where the offsets are 640 on the left and 360 on the top. FSM (see Fig. 9) states and explanations:

- **IDLE:** Starting state where once the module has a valid input point, it offsets that point by how much image 2 is offset in our destination image. The state then moves on to **APPLY**.
- **APPLY:** The inverse homography is applied to the modified point, mapping the point to a pixel and outputting a projected coordinate for where the pixel can be found in image 1. Once the homography has finished being applied, the state moves on to **BOUNDS**.
- **BOUNDS:** To ensure that the mapped pixel exists in image 1, we check the bounds of the projected coordinate are within image 1's resolution.
 - **Within bounds:** If they are, we request 4 pixel values from the points neighboring the projected coordinate from SDRAM. The top, left neighboring point is the rounded down projected coordinate and the other 3 points are either 1 to the right and/or 1 down from it. Using neighboring points is important since the projected coordinate is not a whole number and doesn't have an exact pixel value it corresponds with. Once the 4 pixel values from the neighbors are returned from memory, we move on to the **INTERPOLATE** state.
 - **Outside bounds:** If they are not, we move on to the **FINISH** state with black as the default pixel value.
- **INTERPOLATE:** We use bilinear interpolation on the 4 neighboring points to estimate the pixel value. This helper module essentially calculates the weighted average of the

pixels (giving preference to the top, left $pixel_1$) through the horizontal interpolation of row 1 and row 2 and then the vertical interpolation of those rows. This calculation is done for each of the color channels r,g,b to ensure the colors are accurately estimated. Once the calculation is done, the state moves to FINISH.

$$row_1 = pixel_1 - pixel_1x + pixel_2x \quad (30)$$

$$row_2 = pixel_3 - pixel_3x + pixel_4x \quad (31)$$

$$\text{vertical interpolation} = row_1 - row_1y + row_2y \quad (32)$$

- FINISH: The final state where we set data_out as valid, output the original point and the newly estimated pixel value, and move the state back to the start at IDLE.

E. Combining

This module directly takes the output points and pixel values from the image warping module as inputs to combine the warped image 1 and image 2 (offset to the bottom right corner) into a single image to store in SDRAM. If the input point is within the bounds of where image 2 is offset, we request the pixel value of that point in image 2 (minus the offset) from memory and wait until we get a valid read back. If the input pixel at that point is not default black, that means that image 1 overlaps and we need to average the pixel values from the input and image 2 in 3 separate calculations to account for each color channel rgb. In the case that there is no overlap, the final pixel value will be from the input or image 2 (depending on if its within the bounds). The outputs of this module are saved to SDRAM as the final stitched image.

VI. DESIGN EVALUATION

A. Performance

Initial simplistic description module (10d) had moderate accuracy where around 80% of the matched points were visually accurate, however it was lacking rotation invariance, giving under 20% match accuracy in rotated images. We added CORDIC implementation from the scratch, which computes dominant orientation angle of the feature point and rotate our sampling tests according to that. This yielded averaged better accuracy however it was much inferior to the simple description method even with more general non-rotated cases. We originally thought we had a bug in CORDIC especially related to the signs, however one observation we made was matched points are either almost a perfect match or completely mismatch. Thus, we decided to change our threshold objective to more discriminative yet something simple that can be done in FPGA, which we ended up in Lowe's ratio test. This helped to single out the mismatching decisions yielding almost 90% accuracy in most of the images.

B. Resource Usage

In the detection module, BRAM was used to generate line buffers of the various sizes needed to facilitate downscaling and Harris measure calculation. Overall, even with the 2x upscaling needed when comparing the standalone detection

build and the desired final resolution, the BRAM usage lies well within the FPGA's capacity. DSP usage was also managed by minimizing the number of multiplication steps performed where possible, mainly by using bit shifts to achieve similar effects where possible.

The main resource usage in description – division and angle computation – is avoided by using CORDIC; to compute the dominant orientation of the patch, we run CORDIC on the vectoring mode. However, the main resource usage especially BRAM is in matching as its a brute force method and we need a support to scan through the descriptors multiple times, in addition to the fact that each descriptor is 256 bits and each image can have up to 400 features.

Within the mixing sub-system, the ransac_homography module uses 1 BRAM block to store all the normalized matched pairs of feature points from the matching stage in computation. All other modules in mixing use SDRAM for memory storage (if applicable). Currently, helper module compute_homography is using the most DSP at a given time. Further optimizations could be made to cycle through the necessary multiplications, but they would need to be taken with caution since it would increase the cycles needed for the module and the significantly impact the timing of ransac_homography which iterates compute_homography several times.

C. Timing and Throughput

In the detection module, the filter-like nature of the computational task lent itself to an approach where the camera data could be processed in real time, allowing for very high throughput (effectively real-time computation of features). However, there could potentially have been room for improvement in that the computationally intensive Harris Measure module could have leveraged the fact that at most every other timestep would constitute a valid signal and potentially sacrifice latency to decrease resource use.

Description module is implemented in a blocking mechanism and thus has low throughput due to the BRAM usage and Gaussian pairs: computing Gaussian pairs is expensive and thus we load the pre-computed pairs from ROM, however for one POI, we need to sample 256 pairs which we largely cannot load from BRAM simultaneously, however, if that could be performed in some way, then one POI can be described in one pass. One obvious way would be to heavily pipeline the description module to 256 stages to increase throughput, however we should note that by doing so we will need to keep track of 31x31 window of 16 bit pixels for each stage (total of $31 \cdot 31 \cdot 16 \cdot 256 \approx 3.9$ Mbits). In addition, currently we use CORDIC only to compute the angle but not rotate our Gaussian pairs - note that CORDIC has a latency of about 10 cycles but for each sampling pair of each detected point, we can not use CORDIC as that will introduce $256 \cdot 10$ cycle overhead per detected point. Rather we decided to compute the angle for each detected point once, and use lookup table for computing rotated Gaussian pairs, which is possible as Gaussian pairs are pre-determined/computed. We divide 360

degrees into 32 batches and pre-compute rotated Gaussian pairs by angles increments of $360/32$.

Matching module uses brute-force approach and hence has low throughput by construction. To match single feature point of one image, we need to scan through every single feature point in the other image, thus overall complexity is $O(N \cdot M)$, where N and M are number of detected feature points.

VII. RETROSPECTIVE

A. Goals

For individually working mini-projects on the checklist:

The downscaling and *feature detection* algorithm has been demonstrated running independently on camera input; see Figure 10a. The module in isolation runs off of camera input in real time on the FPGA.

We simulated the *description* and *matching* modules in real image, using software implementation of feature detection: Figure 10b, 10c, and 10d shows the results. As you can see from the figure, implementation that takes into account rotations have less accurate (less than 50% of matched points do actually match), while the rotation omitted implementation has moderate to high accuracy (more than 80%). We expect there is a likely a bug that is caused by sign introduction by CORDIC/momentum or lookup table values for Gaussian pairs (.mem) file was built wrong. The image used ('catabus') is from [5]

B. Use Cases

As is, the individual submodules may serve as computational accelerators for computer vision tasks, including but not limited to image stitching. Many generic applications of image stitching, such as the generation of panoramic photos, surveillance, or medical imaging, could potentially see benefit from our design.

C. Challenges

One of the main challenges was comparing various candidate algorithms and deciding on what would be most suitable for the FPGA. For instance, we were initially looking into the use of algorithms like SIFT and SURF for feature detection; however, the heavy reliance on rotated features and the need to sample at various scales caused concerns for us.

Introducing signed arithmetic came with enough headache, especially in description stage, where the same test passes on individual module but fails when integrated with another module, due to the bit extension or truncation. This bit width struggle was also the case for the mixing stage since signed arithmetic was used with fixed point arithmetic for many of the modules.

We did not allocate sufficient time and manpower to integrating our modules: in particular, interfacing with SDRAM between all our modules proved to be a significant challenge.

VIII. CODE

The following repository contains the code of our system, including test benches and simulations written in Python:

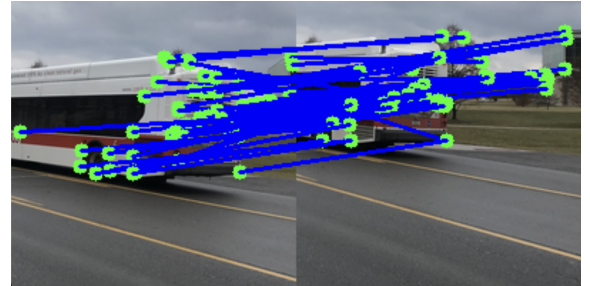
<https://github.com/6205F25/fa25-6205-team64>



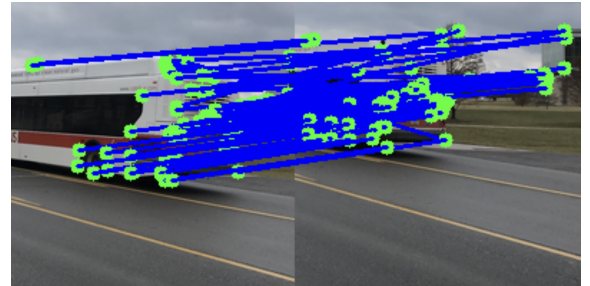
(a) Feature detection results. Green = candidates, orange = after NMS.



(b) Original two images to be stitched.



(c) Rotation-invariant pipeline results.



(d) Results *without* rotation invariance (higher accuracy).

Fig. 10: Mini-project comparison results.



Fig. 11: Rotation invariant – steered BRIEF results

IX. CONTRIBUTIONS

- Davut implemented the feature description and matching modules, including their larger internal submodules like CORDIC, and also their respective test benches. Additionally, worked on SDRAM communication.
- Alianis implemented the image mixing subsystem (and all helper modules found within), modifications to SDRAM communication (including memrequest_handler module), and respective test benches. Additionally, worked on overall block diagram.
- Lucas implemented and tested the feature detection module.

REFERENCES

- [1] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, Barcelona, Spain, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544. keywords: Boats.
- [2] Milos D. Ercegovic and Tomas Lang, "CORDIC ALGORITHM AND IMPLEMENTATIONS", Digital Arithmetic, UCLA, Los Angeles, CA, 2004. Available: https://web.cs.ucla.edu/digital_arithmetic/files/ch11.pdf
- [3] Joe Steinmeyer, "CORDIC and Iterative Algorithms", 6S965, MIT, Fall 2024, Available: https://rfsoc.mit.edu/6S965/_static/F24/lectures/lecture_07.pdf
- [4] R. I. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2004. Available: http://www.r-5.org/files/books/computers/algo-list/image-processing/vision/Richard_Hartley_Andrew_Zisserman-Multiple_View_Geometry_in_Computer_Vision-EN.pdf
- [5] Y. Li, M. Tofighi, and V. Monga, "Robust alignment for panoramic stitching via an exact rank constraint," IEEE Transactions on Image Processing, vol. 28, no. 8, pp. 3833–3848, Aug. 2019, doi: 10.1109/TIP.2019.2909800.