CSEE4840 Design Document: Uncalibrated Stereo Motion Tracking

Kuan Zhang and Leen Alshorafa

April 2026

1 Introduction

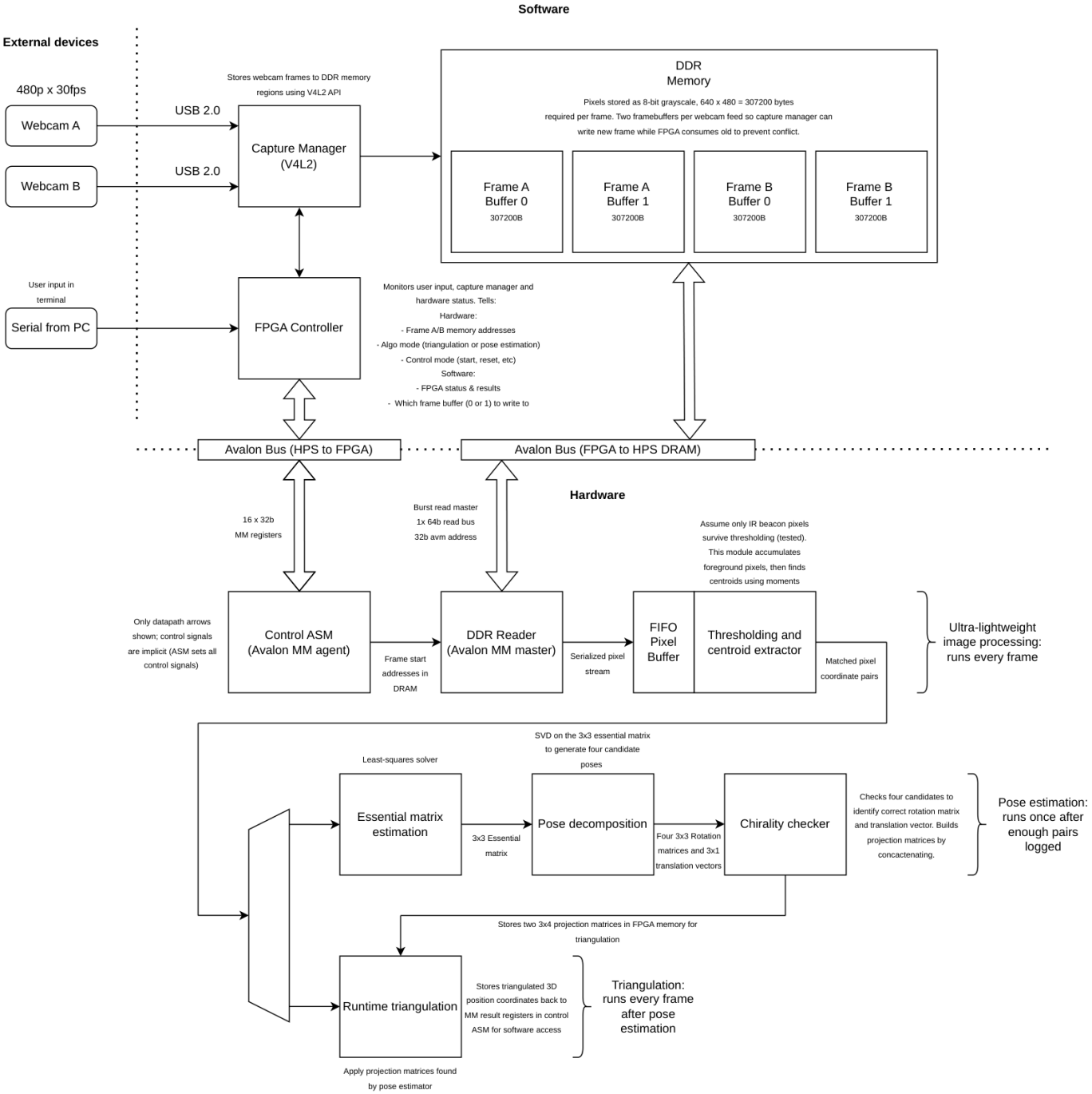
We want to demonstrate a low-cost optical position tracking system for applications such as indoor drone navigation by building an uncalibrated stereo vision processing system using the DE1-SoC. Two USB 2.0 webcams have been modified to see in the infrared band, admitting only light from the IR LED beacon on our custom drone, which will serve as the tracking target. By identifying common points on both webcam feeds, we will use uncalibrated stereo techniques to triangulate the position of the beacon in 3D space. Software on the HPS will handle video capture and user interfacing. FPGA hardware modules will handle thresholding, keypoint detection, pose recovery, and stereo projection.

(a) Modified USB infrared webcam

(b) Custom drone with IR beacons

Figure 1: Custom hardware

2 Block diagram



3 Algorithms

3.1 Problem Statement: Uncalibrated Stereo Vision

Stereo vision seeks to recover the three-dimensional location of a scene point from its two-dimensional projections in a pair of images. Each image measurement loses one dimension of information, since many distinct 3D points lying along a camera ray can project to the same image pixel. The mathematical problem of stereo vision is therefore to recover the original 3D point by combining two such rays from different viewpoints.

In the uncalibrated stereo problem, the relative geometry of the two cameras is not known in advance. More specifically, the rotation and translation between the cameras are unknown, so the two projection matrices cannot be written down directly. Instead, they must be inferred from corresponding image points. The geometric relationship between the two images is encoded by the fundamental matrix F, which satisfies

$$x_2^T F x_1 = 0$$

for every correct correspondence $x_1 \leftrightarrow x_2$. The fundamental matrix depends only on image coordinates and camera geometry; it does not require prior knowledge of the scene depth.

If the intrinsic calibration matrices (the internal projective properties of the camera lens itself) K_1 and K_2 are known, then the problem becomes easier because the normalized image coordinates

$$\tilde{x}_1 = K_1^{-1} x_1, \qquad \tilde{x}_2 = K_2^{-1} x_2$$

can be formed. As such, we have used OpenCV to determine the intrinsic matrices of our webcams. In this case the relation between the two cameras is represented by the essential matrix

$$E = K_2^T F K_1,$$

which satisfies

$$\tilde{x}_2^T E \tilde{x}_1 = 0.$$

The essential matrix contains the relative pose information of the stereo pair and can be decomposed into a rotation R and translation direction t. Once R and t are known, the projection matrices of the two cameras can be constructed, and standard triangulation can be used to recover the 3D point.

Thus the mathematical recovery process used in this project is:

1. Detect the beacon in both images and extract its centroid coordinates
2. Collect corresponding image points from the two cameras during calibration
3. Estimate the epipolar geometry, represented here by the essential matrix E

4. Decompose E into candidate camera poses (R, t)
5. Apply a chirality check to choose the physically correct pose
6. Construct the stereo projection matrices from the recovered pose
7. Triangulate each new centroid pair to recover the beacon's 3D coordinates

Therefore, the central mathematical challenge is not simply finding a bright point in two images, but first recovering the relative geometry of the two cameras well enough that image correspondences can be lifted back into 3D space.

3.2 Image Processing: Thresholding and Centroid Extraction

The runtime image-processing problem is intentionally simplified. The scene is arranged so that, after grayscale conversion and thresholding, only the bright IR beacon remains as foreground. Therefore, the hardware does not need a general-purpose computer vision pipeline. Instead, it only needs to scan the thresholded image and accumulate the centroid of the surviving foreground pixels. This matches the block diagram, which assumes only IR beacon pixels survive thresholding and that the hardware blob detector only needs to accumulate foreground pixels and find centroids.

Let the thresholded image be

$$B(x, y) \in \{0, 1\},$$

where $B(x, y) = 1$ indicates a foreground beacon pixel. For each frame, the hardware computes:

$$A = \sum_{x,y} B(x, y),$$

$$S_x = \sum_{x,y} x\, B(x, y), \qquad S_y = \sum_{x,y} y\, B(x, y).$$

If $A > 0$, the centroid is

$$c_x = \frac{S_x}{A}, \qquad c_y = \frac{S_y}{A}.$$

Thus, for each camera, the thresholding/centroid module outputs one image point (c_x, c_y) per frame. These two image points are the only values needed by the geometric reconstruction stage.

This algorithm is well-suited to a streaming FPGA implementation. Pixels are read sequentially from DDR through the Avalon-MM master reader, passed through a FIFO, thresholded, and accumulated into running sums. No full-frame random-access storage is required beyond the input framebuffers already stored in HPS DDR. This task is performed by the thresholding and centroid extraction module in the block diagram, receiving FIFO-buffered pixels from the DDR reader.

3.3 Epipolar Geometry and the Essential Matrix

Suppose a 3D point X is observed by two cameras. In homogeneous image coordinates (two dimensions and a scaling factor), the corresponding image points are x_1 and x_2. After accounting for projection through the camera's lens using the camera matrices K_1 and K_2, the normalized coordinates are

$$\tilde{x}_1 = K_1^{-1} x_1, \qquad \tilde{x}_2 = K_2^{-1} x_2.$$

If the second camera has rotation R and translation t relative to the first, the two views satisfy the epipolar constraint

$$\tilde{x}_2^T E \tilde{x}_1 = 0,$$

where

$$E = [t]_\times R$$

is the essential matrix, and

$$[t]_\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

is the matrix that represents cross product by t.

The essential matrix is a 3×3 matrix that compactly encodes the relative pose between the two cameras, up to an overall unknown scale on t. Its role is central: once E is estimated from matched point pairs, the camera pose can be recovered.

3.4 Essential Matrix Estimation with the Eight-Point Algorithm

During calibration, the system collects many corresponding image points from both cameras. Because the intrinsic matrices are known, each image point is first converted to normalized coordinates. The essential matrix is then estimated using the normalized eight-point algorithm.

For a correspondence

$$\tilde{x}_1 = (x_1, y_1, 1)^T, \qquad \tilde{x}_2 = (x_2, y_2, 1)^T,$$

the epipolar constraint

$$\tilde{x}_2^T E \tilde{x}_1 = 0$$

expands into one linear equation in the nine unknown entries of E:

$$[x_2x_1 \;\; x_2y_1 \;\; x_2 \;\; y_2x_1 \;\; y_2y_1 \;\; y_2 \;\; x_1 \;\; y_1 \;\; 1] \begin{bmatrix} e_{11} \\ e_{12} \\ e_{13} \\ e_{21} \\ e_{22} \\ e_{23} \\ e_{31} \\ e_{32} \\ e_{33} \end{bmatrix} = 0.$$

Stacking $N \geq 8$ correspondences yields

$$Ae = 0,$$

where A is an $N \times 9$ matrix and e is the vectorized form of E. When more than eight points are used, this becomes an overdetermined homogeneous least-squares problem. Rather than solving it directly with a full rectangular SVD of A, the hardware first forms the normal matrix

$$A^T A.$$

This is a 9×9 symmetric matrix, which is easier to process in hardware.

Implementation-wise, the estimator operates row-by-row. Each matched point pair produces one row of A. For each such row a_k, the hardware computes the outer product

$$a_k^T a_k$$

and accumulates it into a running 9×9 matrix:

$$A^T A = \sum_{k=1}^{N} a_k^T a_k.$$

Because $A^T A$ is symmetric, only one triangular half needs to be stored and updated. This stage is therefore implemented as a multiply-accumulate datapath.After all point pairs have been loaded, the system finds the eigenvector of $A^T A$ corresponding to its smallest eigenvalue. This eigenvector is the least-squares solution for e. In hardware, this eigendecomposition is planned as an iterative small-matrix linear-algebra block rather than a closed-form solver. The resulting 9-element vector is then reshaped into the raw essential matrix

$$E_{\text{raw}}.$$

Thus, the essential matrix estimation block is implemented as three stages:

1. Build each row of A from one correspondence
2. Accumulate the symmetric matrix $A^T A$

3. Solve for the minimum-eigenvalue eigenvector and reshape it into E_{raw}

Pseudocode:

```
EstimateEssentialMatrix(point_pairs, N):
    clear AtA[9][9]

    for k = 0 to N-1:
        build row a[9] from normalized point pair k

        for i = 0 to 8:
            for j = 0 to 8:
                AtA[i][j] = AtA[i][j] + a[i] * a[j]

    [eigvals, eigvecs] = JacobiEigenSymmetric(AtA, 9)

    e = eigenvector with smallest eigenvalue
    Eraw = reshape e into 3x3

    return Eraw

JacobiEigenSymmetric(M, n):
    Q = identity(n)

    repeat for several sweeps:
        for p = 0 to n-2:
            for q = p+1 to n-1:
                if abs(M[p][q]) is not small:
                    tau = (M[q][q] - M[p][p]) / (2 * M[p][q])

                    if tau >= 0:
                        t = 1 / (tau + sqrt(1 + tau*tau))
                    else:
                        t = -1 / (-tau + sqrt(1 + tau*tau))

                    c = 1 / sqrt(1 + t*t)
                    s = t * c

                    for k = 0 to n-1:
                        temp1 = M[k][p]
                        temp2 = M[k][q]
                        M[k][p] = c * temp1 - s * temp2
                        M[k][q] = s * temp1 + c * temp2

                    for k = 0 to n-1:
                        temp1 = M[p][k]
                        temp2 = M[q][k]
```

```
                M[p][k] = c * temp1 - s * temp2
                M[q][k] = s * temp1 + c * temp2

            for k = 0 to n-1:
                temp1 = Q[k][p]
                temp2 = Q[k][q]
                Q[k][p] = c * temp1 - s * temp2
                Q[k][q] = s * temp1 + c * temp2

    eigvals = diagonal of M
    eigvecs = columns of Q
    return [eigvals, eigvecs]
```

3.5 Singular Value Decomposition and Pose Recovery

The raw essential matrix does not in general satisfy the algebraic constraints of a valid essential matrix, so it must be projected back onto the essential-matrix manifold. This is done using singular value decomposition:

$$E_{\text{raw}} = U\Sigma V^T.$$

For a valid essential matrix, the singular values should ideally have the form

$$\Sigma = \text{diag}(s, s, 0).$$

Accordingly, after decomposition the system replaces the singular values with this constrained form and reconstructs

$$E = U \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T.$$

At the implementation level, this decomposition is carried out by a dedicated 3×3 matrix-decomposition block. Since the matrix is small, the design does not require a large general-purpose linear algebra engine. Instead, it can use an iterative rotation-based method such as a Jacobi-style decomposition core. Such a block repeatedly applies plane rotations to drive the matrix toward diagonal form while simultaneously accumulating the left and right singular-vector matrices U and V. The FPGA implementation will use a small control state machine, matrix storage registers, and multiply-add datapaths.

Once U and V have been recovered, the same decomposition is used for pose recovery. Define

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Then the two candidate rotations are

$$R_1 = UWV^T, \qquad R_2 = UW^TV^T,$$

and the translation direction is taken from the third column of U:

$$t = \pm U(:, 3).$$

This yields four candidate poses:

$$(R_1, +t), \quad (R_1, -t), \quad (R_2, +t), \quad (R_2, -t).$$

In hardware, this stage is naturally split into several substeps. First, the SVD block outputs U and V. Next, a small matrix-multiply block computes R_1 and R_2. Then a vector-extraction block copies the third column of U into a translation register and generates both signs. Finally, determinant checks are performed on the candidate rotation matrices; if a candidate has determinant -1, the design flips the signs of both R and t so that the resulting rotation is proper.

Thus, the pose-recovery stage consists of:

1. Decomposing E_{raw} into U, Σ, and V^T
2. Enforcing the singular-value constraint to reconstruct a valid E
3. Computing R_1, R_2, and $\pm t$
4. Generating four candidate camera poses for the chirality checker

The output of this stage is therefore not yet a single camera pose, but a set of four candidates that are passed to the chirality-check block, which determines which one is physically valid.
Pseudocode:

```
RecoverPoseFromEssential(Eraw):
    [U, S, V] = JacobiSVD3x3(Eraw)

    s = (S[0][0] + S[1][1]) / 2
    Sfix = [[s, 0, 0],
            [0, s, 0],
            [0, 0, 0]]

    E = U * Sfix * transpose(V)

    W = [[0, -1, 0],
         [1,  0, 0],
         [0,  0, 1]]

    R1 = U * W  * transpose(V)
    R2 = U * transpose(W) * transpose(V)

    t = third_column(U)

    return (R1, R2, +t, -t)
```

```
JacobiSVD3x3(A):
    U = identity3x3()
    V = identity3x3()

    repeat for several sweeps:
        JacobiSVDStep(A, U, V, 0, 1)
        JacobiSVDStep(A, U, V, 0, 2)
        JacobiSVDStep(A, U, V, 1, 2)

    S = A
    return [U, S, V]

JacobiSVDStep(A, U, V, p, q):
    alpha = 0
    beta  = 0
    gamma = 0

    for k = 0 to 2:
        alpha = alpha + A[k][p] * A[k][p]
        beta  = beta  + A[k][q] * A[k][q]
        gamma = gamma + A[k][p] * A[k][q]

    if abs(gamma) is not small:
        tau = (beta - alpha) / (2 * gamma)

        if tau >= 0:
            t = 1 / (tau + sqrt(1 + tau*tau))
        else:
            t = -1 / (-tau + sqrt(1 + tau*tau))

        c = 1 / sqrt(1 + t*t)
        s = t * c

        for k = 0 to 2:
            temp1 = A[k][p]
            temp2 = A[k][q]
            A[k][p] = c * temp1 - s * temp2
            A[k][q] = s * temp1 + c * temp2

        for k = 0 to 2:
            temp1 = V[k][p]
            temp2 = V[k][q]
            V[k][p] = c * temp1 - s * temp2
            V[k][q] = s * temp1 + c * temp2

    alpha = 0
```

```
beta  = 0
gamma = 0

for k = 0 to 2:
    alpha = alpha + A[p][k] * A[p][k]
    beta  = beta  + A[q][k] * A[q][k]
    gamma = gamma + A[p][k] * A[q][k]

if abs(gamma) is not small:
    tau = (beta - alpha) / (2 * gamma)

    if tau >= 0:
        t = 1 / (tau + sqrt(1 + tau*tau))
    else:
        t = -1 / (-tau + sqrt(1 + tau*tau))

    c = 1 / sqrt(1 + t*t)
    s = t * c

    for k = 0 to 2:
        temp1 = A[p][k]
        temp2 = A[q][k]
        A[p][k] = c * temp1 - s * temp2
        A[q][k] = s * temp1 + c * temp2

    for k = 0 to 2:
        temp1 = U[k][p]
        temp2 = U[k][q]
        U[k][p] = c * temp1 - s * temp2
        U[k][q] = s * temp1 + c * temp2
```

3.6 Chirality Check

Only one of the four candidate poses is physically correct. The correct one is selected using the chirality condition: after triangulating a matched point under a candidate pose, the reconstructed 3D point must lie in front of both cameras.

If the first camera uses the canonical pose

$$P_1 = [I \mid 0],$$

and the second uses candidate pose

$$P_2 = [R \mid t],$$

then a point is triangulated under each candidate pair. The correct pose is the one for which the reconstructed point has positive depth in both camera coordinate systems.

The chirality checker therefore serves as the final decision stage in calibration. In the design, this block examines four candidates, identifies the correct rotation and translation, and then forms the projection matrices used during runtime.

3.7 Projection Matrices and Runtime Triangulation

A projection matrix maps a homogeneous 3D point X into a homogeneous image point x. In normalized coordinates, the two camera projection matrices are

$$P_1 = [I \mid 0], \qquad P_2 = [R \mid t].$$

If image-space projection is desired, intrinsics are included:

$$P_1 = K_1[I \mid 0], \qquad P_2 = K_2[R \mid t].$$

These matrices represent the full camera models used to project a 3D point into each image. Once P_1 and P_2 are known, runtime reconstruction becomes a triangulation problem: given live centroids (u_1, v_1) and (u_2, v_2), solve for the 3D point X that best satisfies

$$x_1 \sim P_1 X, \qquad x_2 \sim P_2 X.$$

In practice, this is written as a homogeneous linear least-squares problem:

$$AX = 0,$$

where A is built from the rows of P_1, P_2, and the measured image points. The solution is the right singular vector associated with the smallest singular value of A, followed by dehomogenization.

This is the final runtime stage of the system. After calibration has been completed once, every new pair of centroids is triangulated using the stored projection matrices, and the resulting 3D coordinates are written back into memory-mapped result registers for software access. The block diagram shows this in the runtime triangulation block.

4 Resource Budget

This design is dominated by frame storage, not by the calibration or triangulation matrices. Full image frames are stored in HPS DDR, while the FPGA only stores small working data structures such as FIFOs, accumulators, and geometry matrices. This follows the intended accelerator model, where large image data is kept off-chip and only compact state is kept on the FPGA.

4.1 Basic sizes

The following estimates are based on:

- Grayscale pixels stored as 8-bit values

- Image dimensions of 640×480
- Calibration and geometry values stored as 32-bit quantities
- The least-squares accumulator $A^T A$ stored as 64-bit values

4.2 Off-Chip Memory (HPS DDR)

The largest storage requirement is the set of four framebuffers shown in the block diagram: two buffers per webcam, with each grayscale frame requiring

$$640 \times 480 = 307200 \text{ bytes.}$$

Therefore, total framebuffer storage is

$$4 \times 307200 = 1{,}228{,}800 \text{ bytes.}$$

4.3 On-Chip FPGA Memory

The runtime image-processing path requires very little on-chip storage. For each camera, centroid extraction only needs running accumulators:

$$\texttt{sumX}, \quad \texttt{sumY}, \quad \texttt{count},$$

plus optional bounding-box values. Even with 32-bit accumulators, this is only a few tens of bytes total for both cameras.

The calibration-point log is also small. If each point pair stores two normalized homogeneous image points, then each correspondence requires only a few tens of bytes, so even dozens of calibration pairs fit comfortably within a few kilobytes.

The largest on-chip memory structure in the runtime path is the FIFO between the DDR reader and the threshold/centroid stage. For a 64-bit FIFO, a typical depth of 512 entries requires

$$512 \times 8 = 4096 \text{ bytes.}$$

4.4 Calibration and Geometry Storage

The essential-matrix solver does not store the full A matrix. Instead, it accumulates the 9×9 symmetric normal matrix

$$A^T A.$$

If stored fully at 64-bit precision, this requires

$$9 \times 9 \times 8 = 648 \text{ bytes.}$$

Because the matrix is symmetric, only one triangular half is strictly necessary, reducing this to

$$45 \times 8 = 360 \text{ bytes.}$$

The Jacobi eigensolver also stores an accumulated 9×9 eigenvector matrix Q, which requires

$$9 \times 9 \times 4 = 324 \text{ bytes}$$

at 32-bit precision.

All remaining geometry matrices are very small:

- Raw and corrected essential matrices E_{raw}, E: $2 \times 3 \times 3$
- SVD outputs U, Σ, and V: three 3×3 matrices
- Four candidate rotations and four candidate translations
- Two 3×4 projection matrices for runtime triangulation
- One 4×4 matrix for linear triangulation

Together these require well under 1 KB.

4.5 Summary

The memory budget is therefore dominated by the four grayscale framebuffers in HPS DDR:

$$1{,}228{,}800 \text{ bytes total.}$$

By contrast, the on-chip FPGA storage required for the FIFO, centroid accumulators, least-squares accumulation, eigendecomposition, SVD, pose storage, and triangulation is only on the order of a few kilobytes. Thus, the design is feasible as long as full frame storage remains in HPS DDR and only compact working state is kept on-chip.

5 Hardware/Software Interface

The hardware/software interface consists of a 16-register Avalon-MM slave used for control, status, debug, and result reporting, along with a separate Avalon-MM master used by the FPGA DDR reader. All slave registers are 32 bits wide, so their byte offsets increment by 4.

5.1 Avalon-MM Slave Register Map

Offset	Name	Purpose
0x00	CONTROL	Start/reset/clear control bits
0x04	STATUS	Busy/done/calibrated/error/result-valid flags
0x08	MODE	Calibration/runtime mode and algorithm select
0x0C	FRAME_A0_ADDR	Camera A buffer 0 DDR base address
0x10	FRAME_A1_ADDR	Camera A buffer 1 DDR base address
0x14	FRAME_B0_ADDR	Camera B buffer 0 DDR base address
0x18	FRAME_B1_ADDR	Camera B buffer 1 DDR base address
0x1C	ACTIVE_BUF	Which buffer per camera the FPGA should consume
0x20	THRESHOLD	Threshold value for centroid extraction
0x24	AREA_LIMS	Maximum and minimum valid foreground area
0x28	RESULT_X	Triangulated X coordinate
0x2C	RESULT_Y	Triangulated Y coordinate
0x30	RESULT_Z	Triangulated Z coordinate
0x34	DEBUG0	General debug register
0x38	DEBUG1	General debug register
0x3C	DEBUG2	General debug register

5.2 Register Bit Definitions

0x00 CONTROL (software write, hardware read)

- bit 0: `start` — begin operation using current mode
- bit 1: `reset` — reset control FSM and internal pipeline
- bit 2: `clear_done` — clear done flag
- bit 3: `clear_error` — clear error flag
- bit 4: `clear_result` — clear result-valid flag
- bits 31:5: reserved

0x04 STATUS (hardware write, software read)

- bit 0: `busy` — hardware currently running
- bit 1: `done` — current operation completed
- bit 2: `calibrated` — valid pose/projection matrices available
- bit 3: `result_valid` — `RESULT_X/Y/Z` contain valid output
- bit 4: `error` — error condition occurred
- bit 5: `frame_ready` — frame has been consumed successfully
- bits 31:6: reserved

0x08 MODE (software write, hardware read)

- bit 0: `calibration_mode` — 1 for pose estimation, 0 for runtime triangulation
- bits 2:1: `algorithm_select` — stereo algorithm select
- bits 31:3: reserved

0x0C–0x18 Frame Buffer Address Registers

These registers store the 32-bit base byte addresses of the four grayscale framebuffers in HPS DDR:

- `FRAME_A0_ADDR`
- `FRAME_A1_ADDR`
- `FRAME_B0_ADDR`
- `FRAME_B1_ADDR`

Software initializes these once and updates them only if the buffer layout changes.

0x1C ACTIVE_BUF

This register tells the FPGA which of the two buffers for each camera should be consumed next.

- bit 0: active buffer for camera A
- bit 1: active buffer for camera B
- bits 31:2: reserved

0x20 THRESHOLD

- bits 7:0: pixel threshold
- bits 31:8: reserved

0x24 AREA_LIMS

This register stores the valid foreground area range for centroid detection. Frames whose foreground pixel count lies outside this range are rejected.

- bits 31:16: maximum valid foreground area
- bits 15:0: minimum valid foreground area

0x28–0x30 RESULT Registers

These registers store the latest triangulated 3D coordinates:

- `RESULT_X`
- `RESULT_Y`
- `RESULT_Z`

Each is a signed 32-bit fixed-point or integer value.

0x34–0x3C DEBUG Registers

These registers expose internal hardware state for debugging. Example uses include:

- Latest centroid coordinates
- Current FSM state
- Number of calibration pairs processed
- Current smallest eigenvalue estimate
- Overflow/error diagnostics

5.3 Typical Software Usage

Software interacts with the accelerator as follows:

1. Capture grayscale frames from both USB webcams into the DDR framebuffers
2. Write the active buffer selection into `ACTIVE_BUF`
3. Write mode, threshold, and valid area range into `MODE THRESHOLD`, and `AREA_LIMS`
4. Start the hardware by setting `CONTROL.start`
5. Poll `STATUS` until `done = 1`
6. If `result_valid = 1`, read `RESULT_X RESULT_Y`, and `RESULT_Z`
7. If needed, inspect `DEBUG0--DEBUG2`

5.4 Avalon-MM Master DDR Reader Interface

The DDR reader is a separate Avalon-MM master used only by the FPGA. It reads image data from HPS DDR using a 32-bit address bus and a 64-bit read-data bus. Its externally visible signals are:

- `avm_address[31:0]` — byte address of the DDR read
- `avm_read` — read request
- `avm_burstcount` — burst length
- `avm_waitrequest` — DDR/interconnect stall signal
- `avm_readdata[63:0]` — returned data
- `avm_readdatavalid` — indicates valid read data

The software does not access this master directly. Instead, software provides frame-buffer base addresses through the slave CSR registers, and the FPGA control FSM passes the selected base address to the DDR reader. The DDR reader then generates sequential addresses:

$$\texttt{avm\_address} = \texttt{frame\_base\_addr} + \texttt{byte\_offset}$$

with `byte_offset` increasing by 8 for each 64-bit read beat.

5.5 Summary

The software-visible interface is intentionally simple: 16 control/status/result registers, each 32 bits wide, are sufficient to configure the accelerator, supply framebuffer addresses, select operating mode, set centroid-detection thresholds, and read back 3D results. Large image data is never passed through the register interface; instead, it is fetched directly from HPS DDR by the FPGA through the separate 64-bit Avalon-MM master reader.