PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

Report on

OPTICAL FLOW HARDWARE IMPLEMENTATION FOR REAL-TIME APPLICATIONS

Submitted by

Pradyun Kamath (PES1UG22EC196)
Pragati Ramesh (PES1UG22EC197)
Pusshya Jagadish (PES1UG22EC221)

August 2024 - December 2025

under the guidance of

Dr. Rajeshwari B
Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

PES
University
Bengaluru
- 560085

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

PROGRAM B. TECH

CERTIFICATE

This is to certify that the Report entitled

OPTICAL FLOW HARDWARE IMPLEMENTATION FOR REAL-TIME APPLICATIONS

is a bonafide work carried out by

Pradyun Kamath (PES1UG22EC196)
Pragati Ramesh (PES1UG22EC197)
Pusshya Jagadish (PES1UG22EC221)

In partial fulfillment for the completion of 7th semester course work in the Program of Study B.Tech in Electronics and Communication Engineering under rules and regulations of PES University, Bengaluru during the period Aug 2024 – Dec 2025. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The report has been approved as
it satisfies the academic requirements in respect of project work.

Signature with date Dr. Rajeshwari B	Signature with date & Seal Dr. Shikha Tripathi	Signature with date & Seal
Internal Guide	Chairperson	Dean -Faculty of Engg. and Technology

Name and Signature of the Examiners:

Names	Signature

DECLARATION

We, Pradyun Kamath, Pragati Ramesh, Pusshya Jagadish, hereby declare that the report entitled, 'OPTICAL FLOW HARDWARE IMPLEMENTATION FOR REAL-TIME APPLICATIONS', is an original work done by us under the guidance of Dr. Rajeshwari B, Professor, Department of ECE, is being submitted in partial fulfillment of the requirements for completion of project work in the Program of Study B.Tech in Electronics and Communication Engineering.

PLACE:
DATE:

NAME AND SIGNATURE OF THE CANDIDATES

S. No.	Names	Signature
1.	Pradyun Kamath	
2.	Pragati Ramesh	
3.	Pusshya Jagadish	

ACKNOWLEDGEMENT

We would like to express our gratitude towards our guide prof. Rajeshwari B, for her unwavering support and guidance throughout the entirety of this project. Her insightful suggestions and motivation has been crucial in shaping this project as a whole. It has been nothing but an absolute privilege to work under her guidance and mentorship.

We extend our heartfelt thanks to Dr. J Suryaprasad, Vice-Chancellor, PES University for his support. We are also grateful to Dr. Shikha Tripathi, Chairperson, Department of Electronics and communication, PES University for her support in facilitating the completion of this project.

Finally, we thank our parents, friends and to all the people who were directly or indirectly involved, for their cooperation and support for guiding and
supporting us in completion of this project work.

Pradyun Kamath (PES1UG22EC196)
Pragati Ramesh (PES1UG22EC197)
Pusshya Jagadish (PES1UG22EC2

ABSTRACT

In contemporary computer vision, real-time motion estimation is still a major challenge, especially for embedded and robotics applications where low latency and resource efficiency are crucial. Both large and small displacements can be tracked with the dal Lucas-Kanade algorithm's reliable optical flow estimation across several spatial scales. However, memory, computational throughput, and representational precision limitations make it rare to translate such algorithms from software to hardware. With an emphasis on resource efficiency and scalable throughput for deployment in real-time systems, this work presents a low-latency and accurate implementation of the pyramidal Lucas-Kanade optical flow algorithm designed for real-time vision systems. Convolution, spatial derivative computation, bilinear interpolation, and pseudo-pyramid construction centred on specific features are just a few of the core modules that were specially created and integrated. Systolic array architectures maximize throughput for embedded and resource-constrained platforms by further speeding up convolution operations. MATLAB pre- and postprocessing were used for verification and error analysis, and the average end point error (EPE) was calculated against the MPI Sintel ground truth. In addition to highlighting design trade-offs in hardware complexity, feature detectability at deep pyramid levels, and the balance between processing speed and resource use, experimental results show the system achieves accurate, low-latency optical flow appropriate for real time systems. The practical requirements of hardware-accelerated real-time vision platforms are bridged by this work with high-quality optical flow algorithms.

The system, which has a measured latency of 1.5 ms per pyramid transition for a 33x33 patch and a hardware footprint of approximately 13.8k LUTs without BRAM, was evaluated on the MPI Sintel dataset. It obtained an Average Endpoint Error (AEE) of 2.5852 pixels when using Shi–Tomasi feature extraction. These results indicate that the architecture is very resource-efficient and is able to maintain a level of accuracy that is compatible with the competition, thus it is suitable for embedding implementation where power, area, and throughput requirements are more important than absolute precision. Consequently, the proposed hardware accelerator is therefore very suitable for real-time scenarios, where low latency and a limited compute budget are stringent requirements, e.g., edge-deployed vision modules, UAV navigation, and small-form-factor robotics.

TABLE OF CONTENTS

TABLE OF DIAGRAMS

LIST OF TABLES

1. INTRODUCTION

Robot and computer vision applications heavily depend on accurately understanding the motion of objects in the environment. Detecting and interpreting such changes are the foundation of functions like autonomous navigation, intelligent surveillance, visual odometry, video stabilization, and interactive multimedia systems. In these areas low latency decision-making, path comprehension, and behaviour prediction require highly precise motion estimates.

One notable method of motion estimation is the use of optical flow which looks at the difference in pixel brightness between the two frames of the video to generate the velocity of each pixel. Optical flow yields very detailed motion fields that have an accuracy higher than one pixel. The optical flow algorithm, however, also makes some assumptions.

The brightness constancy constraint is the major concept which is common to most of the optical flow algorithms. The assumption states that the intensity value of a point in the scene remains almost constant when it is followed from one image frame to another, provided that the frames are taken one after another and no sudden changes in lighting occur. What brightness constancy essentially means is the intensity observed will be almost the same even if the pixel that stands for the physical point in the frame moves to a different location. This is the basis of finding and following the pixels which is a major step in motion estimation.

We can write the optical flow constraint mathematically as follows: the intensity at a given space and time will be the same as the intensity at the corresponding position in the next frame. This relation allows us to link pixel positions through time and thus find the motion that is only apparent in the image sequence.

Yet, when we convert this into a system of equations to find the exact motion (the optical flow vectors), we face the so-called aperture problem. The constraint provides one equation that links two unknown motion components (horizontal and vertical) for each pixel. Due to this underdetermined system, a unique determination is not possible since there are infinitely many solutions for the motion vector at each pixel solely based on intensity constancy.

Additional limitations must be imposed to answer this question. Most differential optical flow methods either:

- Global smoothness constraints, which guarantee that the general flow field does not change abruptly, or
- Local spatial coherence, which assumes that the neighboring pixels have the similar movement.

This supplement enables the robust estimation of the optical flow field throughout the image, thus algorithms like Lucas-Kanade and Horn-Schunck can produce reliable results in practical computer vision applications.

Within the large class of optical flow methods, Lucas-Kanade (LK) algorithm is probably the most cited local method to estimate optical flow. LK applies ordinary least squares minimization to obtain flow vectors under the assumption of constant motion within small image patches. Some of its contributions are:

- Computational efficiency: The linear algebraic formulation is friendly to hardware implementation.
- Local robustness: Patch-based averaging diminishes noise and sensor artifacts.
- Versatility: Works well for both dense and sparse tracking scenarios.

Firstly, large motions between frames can lead to tracking failures, as they contradict the local linear approximation, which is at the core of LK. The pyramidal Lucas-Kanade method is employed to enlarge Lucy-Kanade's capacity, particularly for scenes where objects move rapidly or far. This technique generates a hierarchical series of downsampled images, or "pyramids," with progressively smaller pixel displacements at the upper pyramid levels. After the estimation is done at the coarsest scale, where large movements are feasible, the flows are perfected at the progressively finer resolutions:

- Image pyramid creation: To represent different spatial resolutions, Gaussian or pseudo pyramids are built.
- Coarse-to-fine motion tracking: Flow is determined on the smallest images first, then it is updated and interpolated in larger images.

• Iterative refinement: At each scale new gradient data is incorporated into patch-wise least squares to locally improve flow solutions.

Even though the pyramidal method is more robust, it significantly increases the computational demands due to the processing of multiple scaled versions of the image and the iterative refinement of the flow estimates at all pyramid levels. Software-only implementations cannot easily achieve the low latency and high throughput performance required for real-time operation when image resolutions and frame rates continue to increase in modern applications like autonomous vehicles and robotics. Therefore, there is a need to find hardware acceleration solutions that can deliver the required computational efficiency and predictable performance. FPGAs, because of their inherent parallelism and configurable datapaths, have become a viable platform for real-time pyramidal LucasKanade optical flow with an implementation that is possible in embedded vision systems. Latency and throughput are the main challenges that the authors have to face due to increasing real-time requirements from applications like robotics, autonomous cars, assisted surveillance, and embedded imaging. The platforms have to be capable of processing data at frame rates higher than 30 frames per second, handling high-resolution video streams, and meeting strict power and memory constraints.

In the case of pyramidal optical flow, software solutions are easy to prototype but they rapidly consume computational resources and cannot meet the real-time requirements, in particular, when large images or multiple input streams are involved. A convincing replacement is hardware acceleration via Field Programmable Gate Arrays (FPGAs). FPGAs are very attractive to the vision problems since they provide massive parallelization, pipelined data processing, deterministic execution, and easy integration of custom modules.

By introducing systolic array architectures that provide high throughput and scalability, the efficiency of FPGA for tasks like convolution and matrix computations is being further improved. Nevertheless, the hardware implementation of the resource utilization, memory bandwidth, and modular interfacing is the new trade-off that are taken into account.

This Work's Contributions and Scope addresses the problem of a unified, efficient hardware-oriented implementation of the pyramidal Lucas-Kanade optical flow algorithm. Convolution (Gaussian/local smoothing), central difference operator, bilinear interpolation, and targeted pseudo-pyramid construction focused on specific features are all combined in one high throughput system architecture. The other contributions include:

- A MATLAB-driven evaluation: using average end point error (EPE) as the main metric, pre- and post-processing, error analysis, and benchmarking against the MPI Sintel dataset.
- Resource-conscious feature selection: by adjusting window sizes and limiting pyramid depth so as to achieve the best hardware-software balance.
- Comprehensive evaluation: Demonstrating robust, scalable optical flow under the constraints of embedded real-time platforms.

Organization(to do)

The algorithmic formulation, hardware design methodology, implementation strategy, experimental evaluation, and a critical discussion of tradeoffs and future directions are all covered in detail in the remaining portion of this report.

2. LITERATURE SURVEY

The comprehension of optical flow and how it is calculated is a big part of the perception systems that work in real-time in embedded vision, robotics, and autonomous navigation. In the last 20 years, there have been different differential formulations, robust multi-scale tracking frameworks, deep learning-based dense estimators, and hardware-accelerated architectures proposed that can work under strict latency constraints. Different products reveal different sides; some show the algorithmic advantages of the old methods like LucasKanade, while others indicate the drawbacks of such methods in cases of large displacements or illuminations changes. Simultaneously, the enhancement of FPGA and event-based accelerators reflects the direct connection of architectural decisions to memory usage, energy efficiency, and throughput.

This literature review is a means to combine these different research directions, recognize the factors that can be helpful for embedded real-time systems, and pinpoint the particular insights which resulted in the hardware-accelerated Lucas–Kanade implementation decisions. By taking into account theory, application-specific modifications, robustness, and hardware-realization aspects from sixteen significant papers, we present a powerful case for a multi-scale LK pipeline implementation, aimed at FPGA.

1. Lucas-Kanade and Multi-Scale Processing as the Hardware-Friendly Baseline:

The single most widely acknowledged conclusion in the literature is that pyramidal LucasKanade is the best optical flow method for real-time systems. Bouguet's formulation [15] is used as the theoretical basis for multi-scale tracking, and the results in robotic navigation [5] and drone motion estimation [6] show LK's superiority in the accuracy and throughput balance over HS or Farnebäck. Visual odometry frameworks [12] illustrate how LK can be used perfectly with feature-based modules, and large-displacement extensions [11] show how pyramids and warping can extend LK's range. Local window structure and sparse gradient computations in FPGA accelerators [1], [4], and [16] also let LK be naturally hardware- efficient.

Lucas-Kanade, pyramidal version in particular, is the most trustworthy, scalable, and hardware-friendly optical flow method for real-time embedded systems, as demonstrated by our combined results: works [5], [6], [11], [12], [15], and [1], [4], [16].

2. Addressing Core Limitations: Illumination, Motion, and Dynamic Scenes:

The removal of only three typical troubles of LK - static illumination, small motion, and spatial coherence - is the second main learning point. Paper [7] becomes illuminationindependent by applying energy terms and local color transformations. While [10] makes LK into dynamic SLAM by incorporating multi-view geometric consistency for dynamicobject rejection, [11] and [15] get over small-displacement assumptions through pyramidal multi-scale refinement and warping. The survey [13] treats these issues as the foundation of a larger research area, which means that most of the optical flow methods are still vulnerable to illumination changes, occlusions, and large motion.

Our joint learning: Research works [7], [11], [13], and [10] illustrate that illumination models, pyramidal refinement, and geometric constraints work together to increase the robustness of the real-world scenarios and expand the understanding of the basic limitations of optical flow.

3. Dataflow architectures, memory optimization, and hardware pipelines:

Much of the research points out that memory-efficient, deeply pipelined hardware architectures are a must for high-throughput optical flow. The ultra-high-performance accelerator [4] introduces direction prediction and scalable PE arrays for massive parallelism, while pseudo-pyramidal LK designs [1] reduce memory footprint. At the same time, LK-specific memory optimizations like interleaved downsampling [16] emphasize the importance of limiting external memory bandwidth, while RAFT accelerators [8] draw attention to the resource challenges of deep-learning-based flow. Event-based hARMS [3] gives architectural ideas in the form of fully asynchronous pipelines that focus on localized data processing and lower latency. The requirement of hardware-software co-design for robotics navigation systems is proved by SLAM accelerators [14].

Our consolidated learning: The combined findings of the papers [1], [3], [4],[8], [14], and [16] suggest that the major tasks in efficient hardware design are to lessen memory access, perfect data reuse, parallelize gradient computations, and take advantage of the predictable LK window structures. These ideas are a direct guide to our FPGA implementation.

3. PROBLEM STATEMENT AND OBJECTIVES

Our problem statement is to implement the Pyramidal Lucas Kanade algorithm for object tracking in real-time systems. The focus of the entire report is on latency and resource utilisation.

Our primary objectives are as follows:

1. Reduce Latency with Hardware Acceleration: Create an FPGA-based architecture that, when compared to software (MATLAB/Python/C++) implementations, lowers the endto-end latency. This comprises of the following:
    - utilizing pipeline and spatial parallelism for window summations, gradient computation, and iterative refinement.
    - ensuring deterministic processing time per frame to satisfy real-time requirements (e.g., 30–60 FPS).
    - reducing memory access overhead by using effective dataflow scheduling and buffering.

2. Maintain or Improve Tracking Accuracy

Ensure that the hardware implementation produces optical flow estimates that are comparable to, or better than, established software implementations. Key accuracy considerations are:

- Correct handling of sub-pixel interpolation using fixed-point arithmetic.
- Maintaining numerical stability across pyramid levels.
- Preserving robustness in low-texture or noisy regions through precise gradient and structure tensor calculations.
- Ensuring that the fixed-point design does not introduce unacceptable quantization errors.

3. Optimize for Energy Efficiency

Develop the architecture with a focus on the constraints of embedded and battery-powered systems. Energy-saving strategies include:

- Using fixed-point arithmetic to reduce computational cost.
- Leveraging FPGA-specific optimizations such as DSP slices, BRAM usage, and clock gating.
- Minimizing off-chip memory transfers, as these dominate power consumption.

4. Provide a Flexible and Scalable Hardware Design

Ensure that the architecture can adapt to multiple deployment scenarios and input data formats. This involves:

- Supporting various image resolutions (e.g., 240p, 480p, 720p), pyramid depths, and window sizes.
- Allowing the number of tracked features to be configurable depending on the application.
- Designing modular hardware blocks (gradients, solver, interpolation) that can be reused or expanded.

4. PROPOSED METHODOLOGY

This section describes the entire process used to set up a hardware-software co-processing system for feature extraction and tracking, as well as its expansion to object tracking with YOLOv4. The method combines software for image pre-processing, hardware for estimating optical flow, and software again for tracker re-initialisation, evaluation and visualisation. The next few sections will explain each part of the pipeline, how they work together, and the reasoning behind each of their designs.

4.1 Dataset Preparation and Frame Management

The pipeline begins by acquiring video sequences from the MPI Sintel dataset. This dataset is a well-known optical flow benchmark that has densely annotated ground-truth motion fields for synthetic scenes with near-real features. There are 50 to 100 frames in each video sequence, and they were all made with complicated lighting, motion blur, and environmental effects. This makes them good for testing the robustness of optical flow. To enable frame-by-frame processing, each sequence is decomposed into individual RGB images. The software environment loads these frames and presents them as consecutive image pairs (I_t, I_{t+1}). Each pair forms a single transition whose flow vectors are estimated by the hardware.

4.1.1 Image Pre-processing in Software

Before feature extraction or hardware dispatch, both images in each pair undergo preprocessing designed to improve gradient quality and suppress artifacts that degrade optical flow estimation.

4.1.2 RGB-to-Grayscale Conversion

Optical flow relies primarily on intensity variations rather than color information. Therefore, each image is converted from RGB to grayscale using a perceptually weighted transformation:

$$Igray = 0.299R + 0.587G + 0.114B \qquad \text{Eq 4.1}$$

This ensures consistency of gradient computation while reducing memory transfer overheads to the hardware.

4.1.3 Gaussian Smoothing for Anti-Aliasing

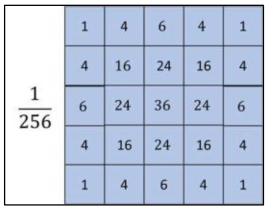
A 5×5 Gaussian low-pass filter is applied to both grayscale frames to suppress highfrequency noise and artifact patterns. The presence of high-frequency artefacts affect the following:

- spatial image gradients
- temporal gradients
- eigenvalue stability in the Shi–Tomasi matrix
- large displacement motion consistency during downscaling

The 5×5 kernel is chosen because we are downsampling only twice to obtain the image pyramid, which involves the removal of adjacent rows and columns. To retain only the information of the nearest rows and columns, and no other parts, we restrain ourselves to a 5X5 kernel, which cover the data that is expected to be lost.

This pre-processed frame pair is then used for feature extraction. 2D convolution of a kernel with an image is explained in detail in the following section

4.1.4 2D Convolution



	1	4	6	4	1
	4	16	24	16	4
$\frac{1}{256}$	6	24	36	24	6
	4	16	24	16	4
	1	4	6	4	1

Fig1: The 5X5 Gaussian kernel with σ = 1 and μ = 0

2D convolution is a commonly used procedure in signal processing to combine two signals and portray how one signal influences the other. It involves sliding one signal over another and multiplying and accumulating the overlapped values of both the signals. In image processing specialised matrices called kernels, are used to exert its properties on the image using convolution. This process is called filtering.

One such kernel is the Gaussian kernel. A 5X5 Gaussian kernel with standard deviation = 1 and mean = 0 is illustrated in Fig1. We are using a 5X5 Gaussian kernel to filter the image before any other process, to reduce the high frequency artefacts and to retain local neighbourhood information before downsampling. This window was selected due to its symmetric nature eliminating the need for flipping the kernel before sliding as is done in conventional convolution. Its radial nature also helps retain an ideal amount of information of the neighbouring pixel values based on its distance from the pixel of interest, making it the best kernel for our use.

4.2 Shi-Tomasi Feature Extraction

The Shi-Tomasi feature extraction method is derived directly from the mathematics of feature tracking, and is hence one of the most commonly used feature extraction technique for a tracking algorithm. The method starts from the observation that reliable tracking of a window requires the associated image motion parameters to be solvable in a stable and noise-robust manner. It employs a symmetric positive semi-definite 2×2 matrix, which encodes the second-moment structure of the gradient distribution and captures the directional observability of motion. The matrix is described as follow:

$$Z = \begin{bmatrix} \sum_w g_x^2 & \sum_w g_x g_y \\ \sum_w g_x g_y & \sum_w g_y^2 \end{bmatrix}$$

Eq 4.2

Where, w is the window within which the gradients are summed over, g_x is the image gradient in x direction and g_y is the image gradient in y direction.

Shi and Tomasi[1] show that accurate and well-conditioned estimation of displacement of features requires both eigenvalues of Z to be sufficiently large. Listed below are the findings summarised:

- If both eigenvalues are small, the window contains nearly uniform intensity; motion is unobservable.
- If one eigenvalue is large and the other small, the window contains only unidirectional gradients; motion is ambiguous along the gradient direction (aperture problem).
- If both eigenvalues are large, the window contains intensity variations in at least two orthogonal directions, making it suitable for accurate 2D motion estimation.

Thus, the quality of a candidate feature is determined by the minimum eigenvalue:

$$\lambda_{\min} = \min(\lambda_1, \lambda_2)$$

Eq 4.3

A point is accepted as a “good feature to track” if:

$$\lambda_{\min} > \lambda_{\text{threshold}}$$

Eq 4.4

Practically, this means Shi-Tomasi features correspond to corners, junctions, textured intersections, and other image patches where intensity varies strongly in two independent directions. Importantly, the selection criterion is derived to maximize the stability of the Newton-Raphson optimization used in optical flow, rather than to heuristically detect corners. Therefore, Shi-Tomasi features are inherently robust to noise, deformable motion within a small window, and the limitations imposed by the aperture problem.

The effectiveness of the Shi-Tomasi criterion has been repeatedly validated through its integration into feature-based optical flow algorithms, most notably the Lucas-Kanade tracker. Bouguet's[2] pyramidal implementation explicitly adopts the minimum eigenvalue test as the core feature-selection mechanism, noting that the same 2×2 gradient matrix (identical to the Z matrix in Shi-To$\overset{G}{\text{m}}$asi) governs the numerical stability of the iterative

Lucas-Kanade update. Thus, features that satisfy the Shi-Tomasi condition $\lambda_{\min} > \lambda_{\text{threshold}}$ are precisely those for which the Lucas-Kanade linear system remains invertible and wellconditioned across pyramid levels. This direct correspondence between the feature criterion and the tracking equations has led to its widespread acceptance as the standard featureselection method for Lucas-Kanade based optical flow.

4.3 Feature Tracking using the Optical Flow Method

Optical Flow is a technique used in Computer Vision, to track the apparent motion of features between successive frames to generate a motion field. The 3 basic assumptions of this technique are as follows:

1. Brightness Constancy: The Intensity level of the feature is assumed to be same or near similar between 2 consecutive frames. This can be formulated as below:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

Where, $I(x, y, t)$ is the intensity of pixel (x, y) at frame t, and $I(x + \Delta x, y + \Delta y, t + \Delta t)$ is the intensity of the corresponding pixel at frame $t + \Delta t$. Eq 4.5

2. Small Displacement: It is assumed that the feature pixel moves only small distances between frames.

$$I(x + u, y + v, t + 1) \approx I(x, y, t) + I_x u + I_y v + I_t$$

Eq 4.6

Where, I_x and I_y are the spatial gradients about the feature pixel at (x, y), and I_t is the temporal gradient.

3. Spatial coherence: The feature along with its neighbouring pixels all move the same between the frames, that is, the feature pixel and its neighbours move smoothly between frames.

Building upon the above differential optical flow formulation, the Lucas-Kanade (LK) method provides a practical and numerically stable solution for estimating motion by explicitly solving the optical flow constraints within a local image window. While the optical flow equation may be underdetermined for some pixels, LK exploits the spatial coherence assumption to aggregate gradient information over a neighbourhood, enabling a stable least-squares estimation of motion. The section below elaborates on this method in detail.

4.4 Lucas-Kanade Optical Flow and its Pyramidal Extension

The Lucas-Kanade (LK) method provides a local, differential approach for estimating optical flow by solving the brightness-constancy constraint over a small window. To solve the below equation, first order approximation of the Taylor series is applied.

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \quad \text{Eq 4.7}$$

For many motion analysis and tracking systems, optical flow estimation is essential. The Lucas-Kanade (LK) algorithm is one of the most popular techniques currently in use because of its computational effectiveness and resilience to slight inter-frame motions. However, the underlying linearization assumptions of the classical LK method are intrinsically limited. Bouguet[2] developed the Pyramidal Lucas–Kanade framework, which incorporates a coarse-to-fine strategy for enhanced tracking stability, to expand its operability to larger motions.

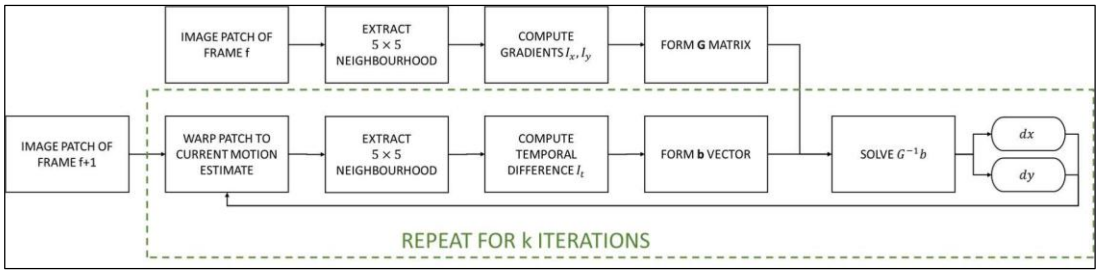


IMAGE PATCH OF FRAME f
EXTRACT 5 × 5 NEIGHBOURHOOD
COMPUTE GRADIENTS I_x, I_y
FORM **G** MATRIX
IMAGE PATCH OF FRAME f+1
WARP PATCH TO CURRENT MOTION ESTIMATE
EXTRACT 5 × 5 NEIGHBOURHOOD
COMPUTE TEMPORAL DIFFERENCE I_t
FORM **b** VECTOR
SOLVE $G^{-1}b$
dx
dy
REPEAT FOR k ITERATIONS

Fig2: Data flow diagram for the pseudo-pyramid Lucas Kanade algorithm.

The LK method's computational procedures, assumptions, limitations, and mathematical underpinnings are described in detail in the sections that follow. A thorough explanation of the pyramidal extension comes next. The idea of a pseudo pyramid (Fig2), which is frequently employed in hardware with limited resources or simplified software implementations, is also explained in a subsection. The integration of pseudo-pyramid with Lucas Kanade is illustrated in Fig2.

4.4.1 Brightness Constancy and Taylor Linearization

The LK method begins with the brightness constancy assumption, which states that the intensity of a point remains unchanged between consecutive frames:

$$I(x, y, t) = I(x + \Delta x,\ y + \Delta y,\ t + \Delta t)$$ Eq 4.8

To make this equation solvable, a first-order Taylor series approximation is applied to the right-hand side:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) \approx I_x \Delta x + I_y \Delta y + I_t + I(x, y, t)$$ Eq 4.9

Rearranging yields the optical flow constraint equation (OFCE):

$$I_x u + I_y v + I_t = 0$$ Eq 4.10

where u is the change in position along x direction and v is the change in position along the y direction. However, to approximate the equation to the first order, the below 2 assumptions are made, which is in line with the optical flow assumptions.

- Small distances: LK assumes the motion of pixels between frames is sufficiently small to assume Taylor's first order approximation. Small Motion Assumption.

  The Taylor expansion is valid only when $(\Delta x, \Delta y)$ is small. Large motion violates
- the linearization, producing inaccurate flow estimates.
- Spatial Coherence: All pixels within the integration window are assumed to share a common displacement between frames. LK operates on a small window (e.g., 5×5, 7×7), assuming all pixels within that region share the same motion vector:

$$u(x, y) \approx u(x + \delta x, y + \delta y) \forall (\delta x, \delta y) \in \text{window}$$ Eq 4.11

These assumptions transform an underdetermined system (one equation, two unknowns) into an overdetermined least-squares solvable system.

4.4.2 Gradient Computations

1/2
−1
0
1
1/2
−1
0
1

(a) (b)

Fig3: Row and Column kernels of central difference operation

We are using the central difference method to determine the gradients of the image along the 2 orthogonal directions, x and y. The 2 kernels used to perform this operation is illustrated in Fig 3. The row kernel (Fig 3(a)) is used to compute the central difference along the x axis and the column kernel (Fig 3(b)) is used to compute the central difference along the y axis.

The reason we use central difference to find the gradients as opposed to other operators like Sobel, Prewitt, Robert etc. is because the kernels are symmetric and small. The operator provides a second-order accurate, centre-aligned estimate of the spatial derivatives, which is better suited for the small-motion assumption at each pyramid level. In a hardware the kernels can easily be approximated to simple bit shifts and subtraction, eliminating the need for complex multipliers and adders, as would be required for other kernels.

The other kernels have built-in smoothing, portraying edge affinity, which is undesirable for the small-motion assumption and Taylor's approximation used in LK.

4.4.3 Least-Squares Solution

For all pixels i inside the window, we obtain:

$$I_x(i)u + I_y(i)v + I_t(i) = 0 \qquad \text{Eq 4.12}$$

Stacking all equations in matrix form:

$$\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_x^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_w I_xI_t \\ \sum_w I_yI_t \end{bmatrix} \qquad \text{Eq 4.13}$$

$$Gv = b \qquad \text{Eq 4.14}$$

The matrix G is the spatial gradient matrix (or structure tensor). A stable solution exists only if:

G is well-conditioned. Both eigenvalues of G are large enough (Shi–Tomasi criterion).
Thus, LK is typically applied at corner points, which provide robust gradient information.

4.4.4 Limitations of classical LK

Classical LK fails under the following conditions:

- Large inter-frame displacement
- Fast object motion
- Low texture regions
- Motion blur

The linearization error increases rapidly with displacement, making the small-motion assumption a major bottleneck. Enlarging the window would reduce this problem but at the cost of:

- Smoothing out motion boundaries
- Increasing computational load
- Violating the spatial coherence assumption

This limitation motivated the Pyramidal Lucas-Kanade approach.

4.4.5 Gaussian Pyramid Construction

The pyramidal approach constructs a sequence of progressively downsampled images. It involves first convolving the image with a low-pass gaussian filter followed by removal of alternate rows and columns to get the image at half its original resolution. This process is repeated on the previous layer to obtain the next layer, thus forming the pyramid.

The reason this overcomes the fast object motion constraint of classical LK is because, A large motion in level 0 is reduced to a small motion higher (coarser) levels, due to the scaling down by 2 at each layer to match the layer resolution.
To prevent anti-aliasing during subsampling to create the pyramid, a Gaussian or other low pass filter is required. The Gaussian kernel is perfect for our implementation since its filter coefficients are all powers of two, which makes it simple to implement as a left or right shift in hardware.

4.4.6 The Pyramidal Lucas-Kanade Flow

The pyramidal LK algorithm follows a top-down refinement procedure. This is described in the following steps:

Step 1: Start at the Coarsest Level

At the highest pyramid level L, motion is smallest. Classical LK is applied to estimate coarse displacement (u_L, v_L).

Step 2: Upsample the Motion Estimate

The displacement is scaled by 2 (or the pyramid factor) to serve as an initial guess for level $L-1$:

$$u_{L-1}^{(0)} = 2u_L, v_{L-1}^{(0)} = 2v_L \quad \text{Eq 4.15}$$

Step 3: Patch Warping

At each level, the second image patch is warped according to the current motion estimate:

$$J_w(x, y) = J(x + u^{(k)}, y + v^{(k)}) \quad \text{Eq 4.16}$$

Warping ensures that the residual motion becomes small enough for a valid Taylor approximation.

Step 4: Iterative Refinement

Each level applies multiple LK iterations:

- Compute temporal residual:

$$I_t = J_w - I \quad \text{Eq 4.17}$$

- Reuse the precomputed gradient matrix G.
- Solve:

$$\Delta v = G^{-1} b \quad \text{Eq 4.18}$$

- Update

$$v^{(k+1)} = v^{(k)} + \Delta v \qquad \text{Eq 4.19}$$

- Iterations continue until:
  - $|\Delta v| < \varepsilon$, or
  - A maximum iteration count is reached.

Step 5: Final Subpixel Flow

The process continues down to level 0, yielding a robust subpixel-accurate flow vector (u_0, v_0).

4.4.7 Pseudo-Pyramid

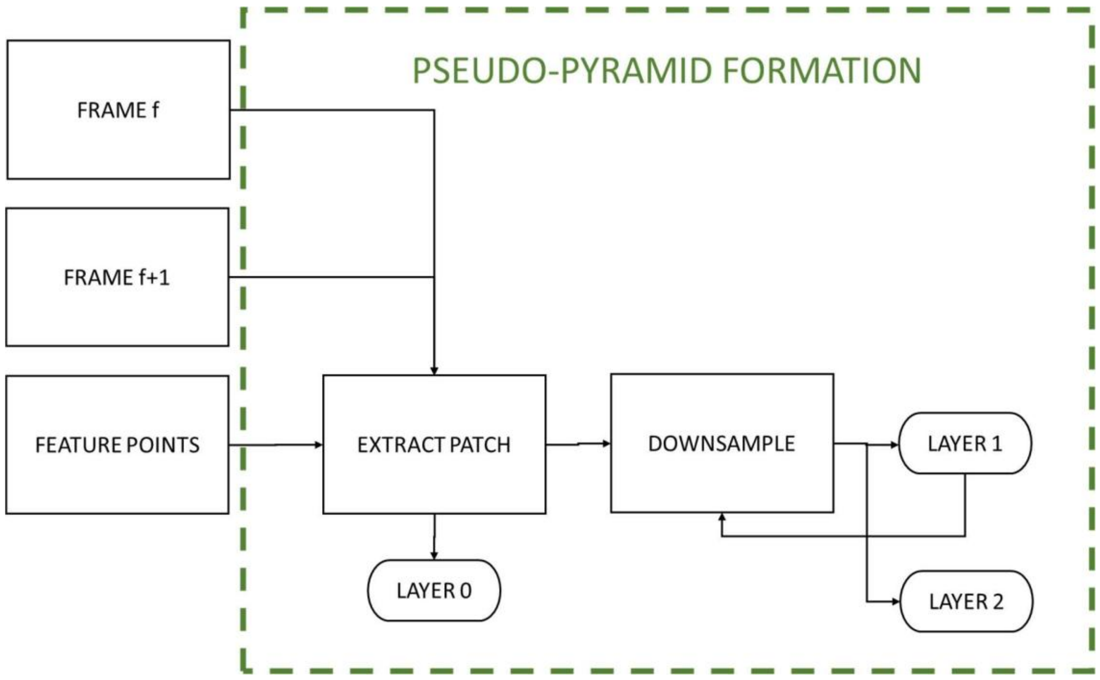


Fig4: Flow diagram for the formation of a pseudo pyramid. The Layer 1 patch is once again downsampled to obtain Layer 2 patch

In some hardware-accelerated or low-memory systems, building a full Gaussian pyramid is expensive. A Gaussian pyramid can be computationally simplified by using a pseudopyramid (Fig4), which avoids true downsampling. Instead, only the patch size is changed to mimic coarse-to-fine behaviour; the image is not resized. Since it mimics the pyramid idea, hence the name "pseudo".

To create a Gaussian pyramid, multiple lower-resolution versions of the image must be stored in memory, which requires substantial resources for large images and embedded
devices. A 436x1024 frame, for instance, has 446,464 pixels; its two downscaled layers, 218x512 and 109x256, have 111,616 and 27,904 pixels, respectively. 4.47 MB of memory would be needed to store all three levels with 64-bit precision; with 8-bit intensity values, the total memory needed for these three layers is approximately 572 KB. In systems with limited on-chip memory or where accessing external DRAM adds latency and power, allocating several megabytes for multiple pyramid levels is not practical. By only storing the original image in memory and simulating coarser scales through larger patches and additional smoothing on the fly, the pseudo-pyramid avoids these expenses.

This method is ideal for real-time, resource-constrained hardware implementations because it significantly reduces memory footprint and external memory bandwidth while maintaining the coarse-to-fine search semantics required for sparse trackers. Reduced fidelity for very large displacements and the need for careful smoothing to prevent aliasing are the trade-offs, but these are typically acceptable compromises for sparse tracking workloads.

4.4.7 Parameters of the proposed implementation

The Lucas–Kanade formulation estimates optical flow by assuming that pixel intensities move slightly and roughly continuously within a local neighbourhood surrounding each feature point. The 5×5 window used to compute the spatial gradient matrix
A fair trade-off between numerical stability, noise robustness, and feature localization is offered by G and the error vector b.

Because there are insufficient gradient samples to create a well-conditioned G, a small window like 3x3 frequently results in an under-constrained or ill-conditioned system, particularly in areas with weak texture. Bigger windows (7x7 or larger) break the localconstant-motion assumption, increase computational complexity, and blur motion boundaries.

A 5x5 window is small enough that motion inside the window can still be assumed to be uniform, but it provides sufficient spatial support to gather gradient information and generate a stable G. The window provides enough redundancy to solve the 2x2 normal equations without over-smoothing fine motion details, averages out image noise, and increases the accuracy of eigenvalue-based corner detection. As a result, in practical Lucas– Kanade implementations, the 5×5 window size is commonly used as the best balance between accuracy, stability, and computational efficiency[15].

4.5 Software Evaluation and Visualization

Once per-feature flow vectors are available, the software computes optical-flow accuracy and prepares visualizations.

For each feature coordinate, End Point Error (EPE) is calculated as follows:

$$EPE_i = \sqrt{(dx_i - dx_i^{GT})^2 + (dy_i - dy_i^{GT})^2}$$

Eq 4.20

dx_i^{GT}, dy_i^{GT} are obtained from the MPI Sintel ground-truth flow fields. where

8. Average Transition Accuracy

For each frame transition:

$$EPE_{\text{avg}} = \frac{1}{N} \sum EPE_i$$

Eq 4.21

This quantifies hardware-accelerated LK performance on a per-transition basis.

9. Quiver Visualization

To interpret the motion field, software overlays:

- The feature points on the original RGB image
- hardware-computed flow vectors on the original image using quiver plots. This visualization confirms both correctness and spatial consistency of the tracked motion.

4.6 Object Tracking Application Using YOLOv4 and Optical Flow

To demonstrate use beyond tracking sparse features the system is expanded to track at the object level specifically following a person, in the MPI Sintel sequence.

We use the YOLOv4 model, to detect a person. YOLOv4 (You Look Once version 4) is an advanced single-pass convolutional neural network detector trained on the MS-COCO (Common Objects in

Context) dataset featuring 80 object categories. We utilize it for object detection to identify a bounding box from which we sample features. YOLOv4 integrates the CSPDarkNet53 backbone, PANet path aggregation and spatial pyramid pooling (SPP) to achieve detection suitable, for real-time use.
The version history of the You Only Look Once CNN is as follows:

- YOLOv1 (2015): The introduction of the YOLO network presenting single-shot detection for the first time
- YOLOv2/3 (2016–2018): improved multi-scale detection and deeper backbones
- YOLOv4 (2020): designed for GPUs and real-time detection greatly enhancing precision through training techniques (Mosaic augmentation, CIoU loss, etc.)
- YOLOv4 analyses the image and outputs bounding boxes along with confidence levels.

4.6.1 Feature Sampling Inside the Object Region

Once YOLOv4 identifies an object, the bounding box serves as the region from which features are extracted, to pass into the Pyramidal Lucas-Kanade algorithm. The features within the bounding box are selected and tracked as follows:

- For each detected bounding box, a central sub-region is defined by scaling the width and height by 0.4 (That is, 40% of the width and 40% of the height of the bounding box make up the central patch).
- Within this central sub-region, an N×N grid of sampling locations is generated using uniformly spaced coordinates along both axes. These coordinates serve as the initial feature points for Lucas-Kanade tracking.
- The individual displacements of all tracked points are then averaged to estimate the net motion of the object.
- This averaged displacement is used to update the position of the bounding box, enabling continuous object tracking even in the absence of repeated detections.
- The YOLOv4 detection is forced to re-detect every 5 frames for smooth running.

4.6.2 Hardware Flow Computation for Object-Level Tracking

Just like in sparse feature tracking, the device outputs 100 flow vectors indicating the movement of the sampled ROI points, between frames.

4.6.3 Bounding Box Motion Estimation and Update

To estimate object motion:

- Compute the average flow vector over all 100 features:

$$(u_{avg}, v_{avg}) = \frac{1}{100}\sum(dx_i, dy_i)$$ Eq 4.22

- Shift the bounding box centroid by this average displacement.
- Use the new bounding box for:
    - o visual tracking display
    - o sampling the next 10×10 features for the next transition

This approach allows smooth, continuous tracking even when YOLOv4 is not executed for every frame. In practice, the detector is invoked once every 5 frames to re-synchronize the box and avoid drift.

4.7 Integrated Hardware–Software Pipeline

The entire process constitutes an integrated cycle:

Software (SW)

- reads MPI Sintel frames
- pre-processes images
- extracts features (Shi–Tomasi) or object ROI features (YOLOv4)
- packages coordinates and images

Hardware (HW)

- extracts local patches
- forms pseudo pyramids
- performs pyramidal LK with iterative refinement
- outputs motion vectors

Software (SW)

- computes EPE vs ground truth
- visualizes flow
- updates bounding box
- prepares next transition

This flexible design allows scalable sparse tracking, object-level tracking, hardware– software partitioning suitable for FPGA/ASIC acceleration and benchmarking against a world-standard optical flow dataset.

5. IMPLEMENTATION

The implementation of this project spans both software and hardware. This section contains a detailed explanation of how the previous section has been implemented.

The core of this project, the pyramidal Lucas Kanade algorithm, has been implemented using Verilog, a hardware description language. HDL is a language used to describe the circuit one wants to design. Use of this has made designing circuits much easier, considering how a few lines of code can describe a complex module, as opposed to making individual connections on a circuit simulator. The purpose of an HDL is to eventually synthesize and make a circuit out of the proposed design or implement it on an FPGA. The software used for this is Xilinx Vivado. Xilinx Vivado is an open-source tool to work on FPGA and related projects. A project creation in this tool starts with selecting an FPGA board. The resource utilization is compared against the available resources of the selected board.

The proposed design cannot be used as an ASIC because we have used '*' (the multiplication operator) wherever necessary and not a multiplier block. The synthesis tool references the DSP blocks to perform the multiplication. This is much faster than implementing a custom multiplier block due to the inherent parallelism in the DSPs. The division operation is done using '/' operator, which is replaced by an AMD IP (intellectual property) block during synthesis. The design is also simulation only, as of now, since the memory elements have not been replaced by synthesizable constructs/IPs.

The entire design is based on fixed point notation to simplify the arithmetic operations and representation. Some rules to note about fixed point arithmetic. Consider a = 0.5 and b = 0.25 with a Q3.2 notation. QM.N means, a number has M bits to represent the integer part and N bits to represent the fractional part. In binary, a = 000.10 and b = 000.01. When representing fixed point in Verilog, decimal point is omitted and now a = 00010 (2) and b = 00001 (1). To get the decimal point back, divide the number by 2^ (number of fractional bits). Note that number of fractional bits in the result changes with the operation performed. Arithmetic operations on a, b are done as in equations 5.1 to 5.6.

1. Addition:
   - a + b = 2 + 1 = 3 → 00011 (3/4 = 0.75) = 0.5+0.25 Eq. 5.1

2. Subtraction:

- a – b = 2 – 1 = 1 → 00001 (1/4 = 0.25) = 0.5 – 0.25 Eq. 5.2

- (unsigned) b – a = 1 – 2 = -1 → 11111 (31/4 = 7.75) != 0.25 – 0.5 Eq. 5.3
- (signed) b – a = 1 – 2 = -1 → 11111 (taking 2's complement: -(00000+1) = -1 and -1/4 = -0.25) = 0.25 – 0.5 Eq. 5.4

3. Multiplication: a*b = 2*1 = 2 → 00010 (2/16 = 0.125) = 0.5*0.25 Eq. 5.5

[When multiplying fixed point numbers, the product's decimal point shifts (p+q) positions to the left, if p, q are number of fractional bits of a , b respectively.]

4. Division: a/b = 2/1 = 2 → 00010 (2/1 = 2) = 0.5/0.25 Eq. 5.6

[When dividing fixed point numbers, the result's decimal point shifts (p-q) to the left, if p, q are number of fractional bits of a, b respectively. This operation also strictly demands that p>=q, else p-q is negative, and a negative left shift makes little sense.]

Signs in multiplication and division are handled as in the subtraction operation. In Verilog, a QM.N number is handled as a number with (M+N) bits. So, all the results are scaled up. In order to interpret the results correctly, a MATLAB script is used to convert them to fixed point numbers.
Typically, every complex design is divided into control path and data path. Data path consists of all the modules that operate on given data. Control path controls the flow of data into the data path. The data to be processed is stored in the memory. The next section explains about the memory element used.

5.1 Synchronous First In First Out (FIFO)

The concept of FIFO is that, the first element written into it is the first one accessed. It's a memory element with sequential access, not random access. It has a write pointer which moves by one every time an element is written into it. It has a read pointer that moves by one every time an element is read. A FIFO can be written into, only when write enable is asserted. Similarly, it can be read from, only when a read enable is asserted. A synchronous FIFO follows the clock signal to write or read a value. Unless using the global asynchronous reset, the contents of the FIFO remain unchanged unless written into. When a synchronous local reset is used, both the read and write pointers reset to their initial position, but the contents remain unchanged. For our project, we have used a linear FIFO, which means a FIFO:

- Is full when write pointer = DEPTH -1 (DEPTH is the number of elements a FIFO can accommodate)
- Is empty when read pointer = 0

The reason for this is that, for this project, 8 pixels, two consecutive columns with four consecutive rows, are output at once. This style of output is absolutely necessary for the downstream logic to work as described in the next sections. A typical BRAM has only two ports at max with at least one clock edge read latency from the rising edge of the read clock, which means two clock cycles in total, which is different from our required latency of one clock cycle only, hence a custom FIFO logic is used here.

Streaming means reading one (or more) pixels every clock cycle, sequentially. To stream more multiple pixels at a clock edge, multiple read pointers are used. This doesn't let us use the condition that FIFO is empty when read and write pointers are equal. At the beginning of our algorithm, the read and write pointers are set to zero. Later, read pointers are initialised to certain values to enable faster streaming. This might let read and write pointers coincide, raising false alarms, which is not desired.

When performing operations such as convolution (to find gradients for e.g.) on an image, the pixels are accessed sequentially, which is the main reason using a FIFO is preferred to a RAM. In the synthesis tool, a FIFO module infers a BRAM (if its of a certain size). But due to the sequential access, the address generation logic becomes much simpler for a FIFO, than a BRAM.

5.2 Patch Extraction

The image is first initialised in the testbench using $readmemh command. The system design starts from this point, assuming the image is already in the memory. The block diagram for the address generator is given in Fig.[]. The start address is calculated and given to the address generator by the FSM controller using the formula given by Eq. 5.7.

Start address = (r – pr)*cols + (c - pc) Eq. 5.7

- (r, c) are the pixel coordinates of the feature point, which become the centre pixel of the extracted patch
- cols is the number of columns of image 1
- pr*pc is the patch to be extracted around the centre pixel

Note that the entire process further revolves around the feature point being the centre of the patch. The concept of patch, pseudo image pyramid and reasons of usage have been explained in earlier sections of the report.

While the patch is being extracted, it is down sampled and stored into a layer FIFO. The concept of FIFO is explained in the previous section. The next section explains the pyramid generation process.

5.3. Pseudo Image Pyramid Generation

The process of down sampling the pseudo image/ the patch in parallel, while reading it has been referred from [1]. Since project implements a 3-layer image pyramid, 3 FIFOs are required for each image. They are named as: L0, L1, L2. L2 contains layer 2 of the pyramid and so on, for each image.

Down sampling once means reducing the rows and columns of an image by half. This process is divided into routing to the appropriate FIFO and generating the appropriate write enable signals. The write enable signals for different layers of FIFOs are derived from the base clock through clock division (a clock divide by 2 is implemented using a TFF). The write enable signal is common for a FIFO of same layer. A counter keeps track of the number of pixels read, which is the routing logic. Based on this counter, every second pixel (even) is sent to the layer 1 FIFO and every fourth pixel (mod4) is sent to the layer 2 FIFO. The process can be implemented without the routing logic as well, since writing into a FIFO entirely depends on the write enable signals.

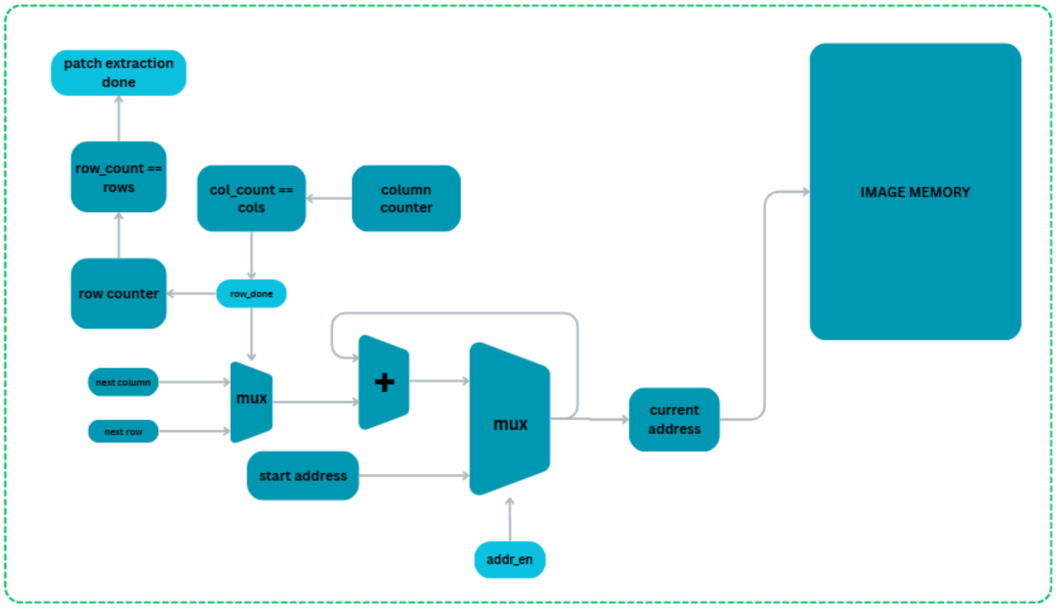


Fig.5. Block diagram of the address generator used to extract a patch of pixels from the image stored in memory.

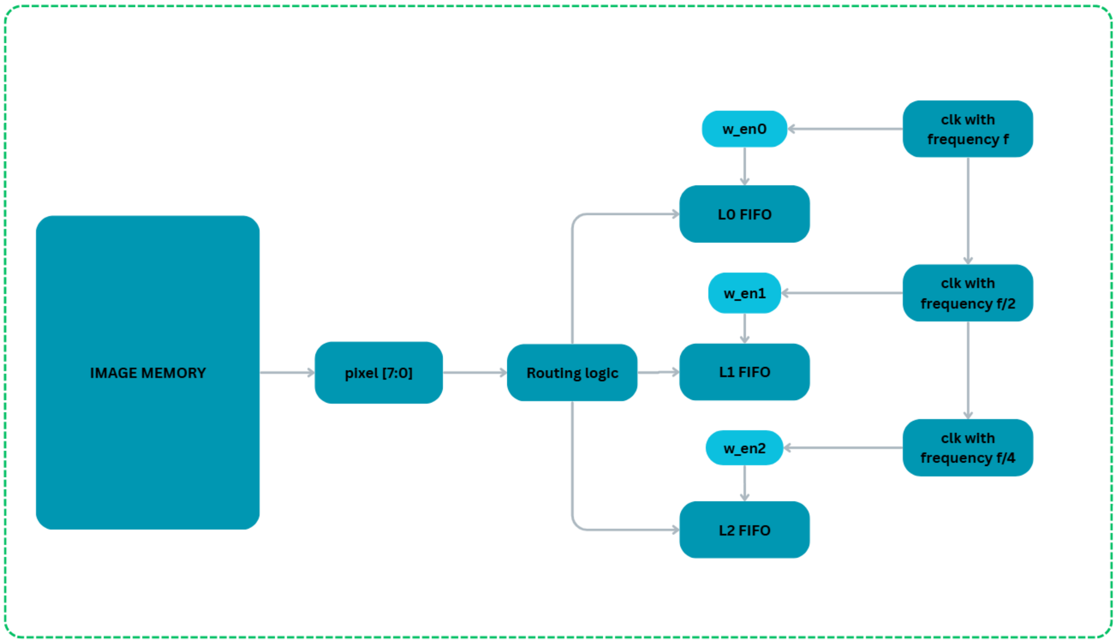


Fig.6. Block diagram to depict the pseudo image pyramid generation process.

The entire system is divided into data path, which performs various operations on the data, and the control path, which controls the data flow to different modules. After the pyramid is generated, the processing is done from layer 2. Each layer uses the same data path, but the controller decides which layer of the pyramid is to be processed. The working of the data path, with block diagrams for each module is given in the next section.

5.4. The Data Path

An MxM patch is written into the FIFOs, where M can vary based on the application. But for processing the LK equation, we need a 5x5 window around the centre pixel, as mentioned in section 4. For this, the read pointer must begin at a particular address, given by Eq.5.8.

row k pointer = 1+((pr-wnr)*cols + (pc-wnc) + (k-1)*cols) Eq. 5.8

- where pr x pc is the patch row and column size
- wnr = wnc = 5 (since we need to extract a 5x5 window)
- k is the row number within the FIFO (since our FIFO can read 4 rows at a time)
- a 1 is added to every address. Every counter or pointer is reset to zero, so when they are enabled, the counting starts from 1. Basically, the write pointer writes into the FIFO from address 1, which makes it one-based indexing.

This address is loaded into the read pointer of the FIFO by the controller. Once loaded, the controller enables the bilinear interpolation module. The FIFO reads eight pixels at a time. These eight pixels, after interpolation, produce three pixels which is given to the image gradient module, explained in the next section. It is important to note that, a window is extracted around the pixel whose coordinates are input. If the coordinates are not integer values, it’s fractional part is extracted and used to calculate the subpixel intensities for every pixel in the window. In other words, the fractional part used to calculate the interpolated intensity for every pixel in the extracted window is constant and equal to the fractional part of the input pixel coordinates (which is the centre pixel). This makes it seem as if, a 5x5 window is extracted around the point at the subpixel coordinate. Hence, this logic makes sense. Refer to Fig.7(a) for a more visual representation of this logic.
Fig.7(a). The figure on the left explains the logic behind using the same fractional part as the centre pixel to calculate the interpolated intensities of all the pixels in the window extracted around that centre pixel. Consider the blue points to be pixels at integer coordinates.

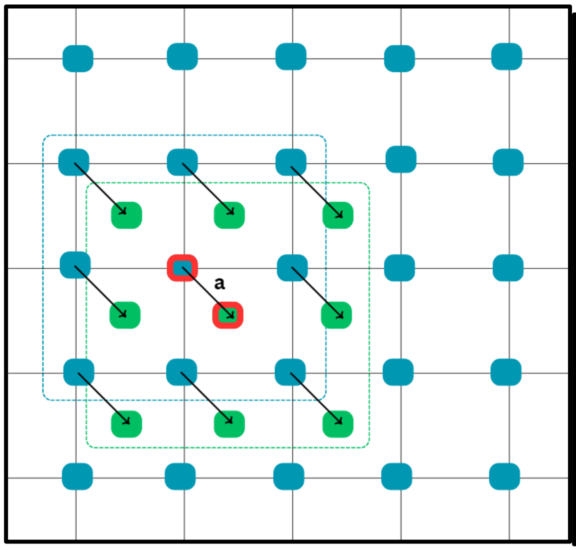


Fig.7(a)

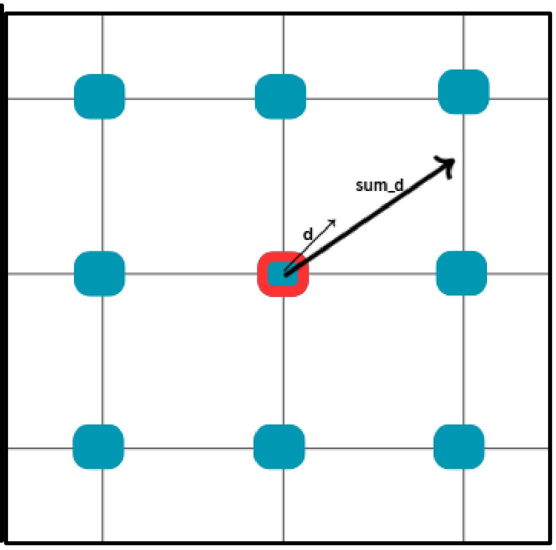


Fig.7(b).

The blue pixel with red border is the centre pixel of the 3x3 window show using blue boundary. After and iteration, the centre blue point is shifted by ‘a’ value, as shown by the arrow. The shifted centre pixel, given by green point with red border is the new centre pixel, around which a 3x3 window must be extracted. To get this window, the other blue points (pixels in

the 3x3 window of the previous centre pixel (blue point)) must be shifted by the same amount. Fig.7(b). In the figure on the right, consider the blue points as the pixels at integer locations of image 2. The centre pixel is the blue point with red boundary. Suppose, ‘d’ is the output of the current iteration and ‘sum_d’ is the total flow vector across all iterations till the current iteration. At the beginning of every iteration, the FIFO read pointer points to the centre pixel. If only ‘d’ is added, then all the information that’s done till now is lost and the windows keeps staying close to the centre pixel, because each time similar window is extracted, and similar output is found, so no progress is made. Hence, the total vector ‘sum_d’ must be used to find the new window, so that the individual shifts made during every iteration till the current one is accounted for.

5.4.1 Image Gradient Calculation using Central Different Operator

There are two image gradient kernels, row and column. The column kernel is simple subtraction, which is done at the same clock edge. The architecture used for convolution, using row kernel, is inspired from the systolic array architecture, but it is simpler.

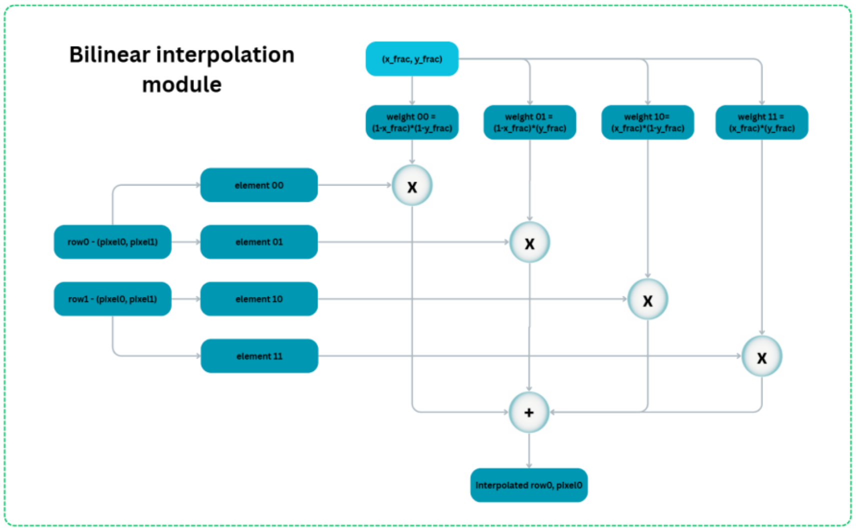

Bilinear interpolation module
(x_frac, y_frac)
weight 00 = (1-x_frac)*(1-y_frac)
weight 01 = (1-x_frac)*(y_frac)
weight 10= (x_frac)*(1-y_frac)
weight 11 = (x_frac)*(y_frac)
element 00
row0 - (pixel0, pixel1)
element 01
row1 - (pixel0, pixel1)
element 10
element 11
X
+
Interpolated row0, pixel0

Fig.8. Block diagram for the bilinear interpolation module. Two pixels from two consecutive rows, so four pixels are required to find the interpolated pixel intensity value. The FIFO reads eight pixels from two columns and four consecutive rows, so a total of three bilinear interpolation modules is required for each FIFO.

The kernel values are noted in section 4. Due to the simple nature of the kernels, the MAC unit can be optimized. A systolic cell is synchronous and gives output at every clock edge.

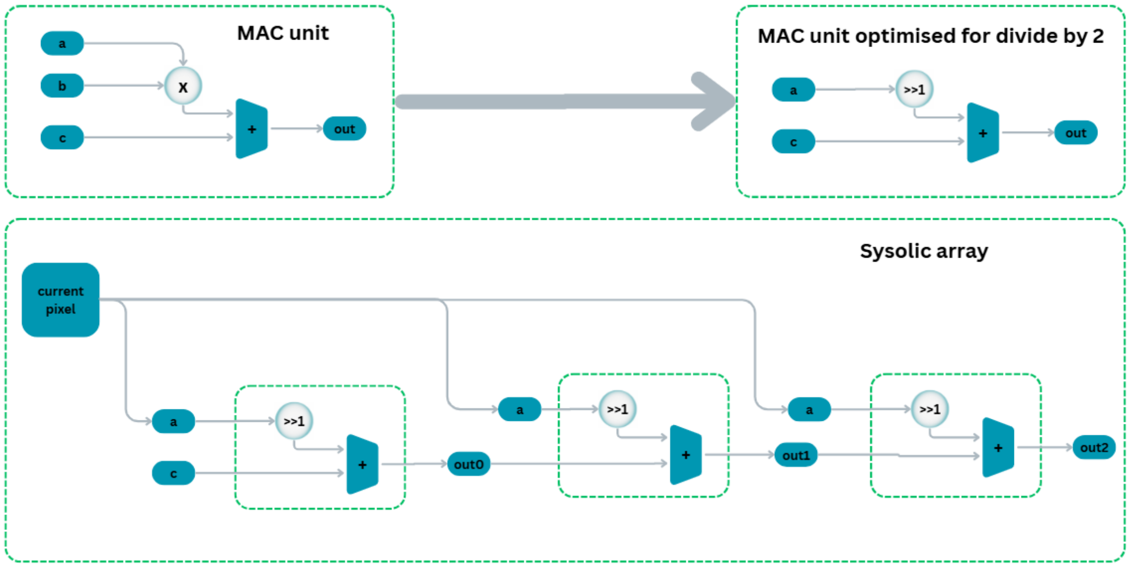


Fig.9. The upper part of the figure shows the MAC unit and how it can be optimized to replace the multiplier with a shifter, which reduces the resources utilised. The block with the '+' sign is an accumulator, that is an adder with a storage element.

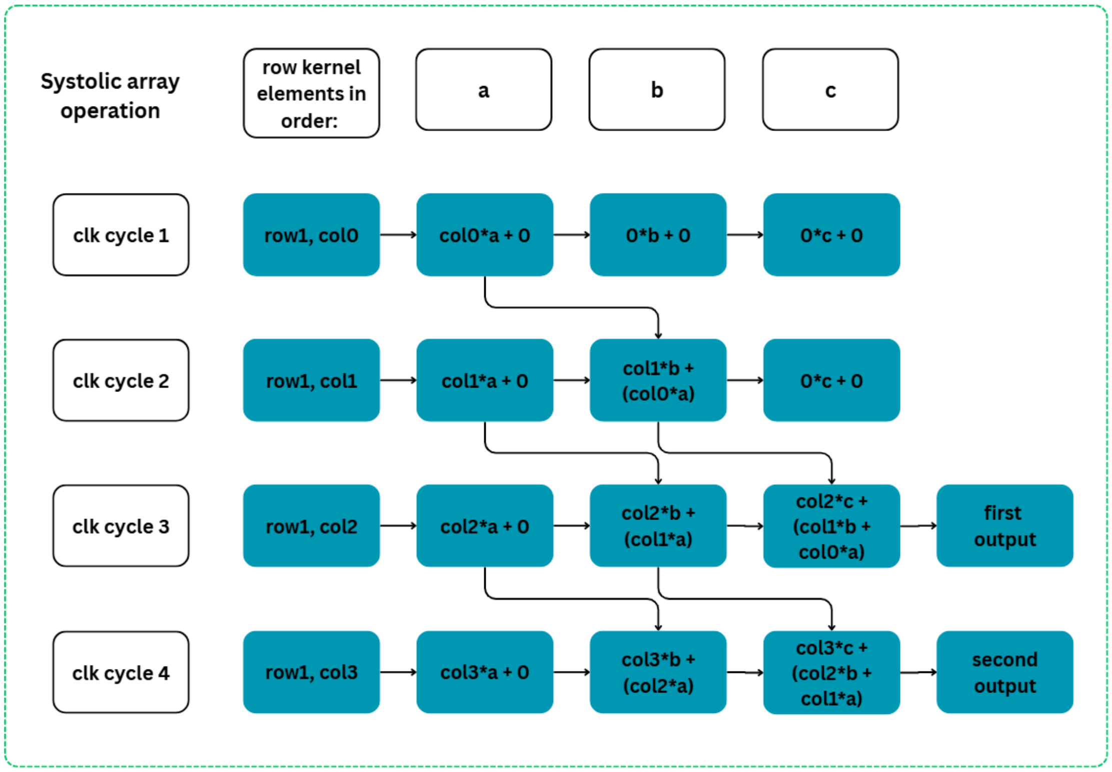


Fig.10(a). This figure shows the operation of a systolic array at every clock edge. For convolution, the kernel [a b c] is fixed (as part of the systolic cells) and the data is flipped before giving as input.

From Fig.10(a), it can be seen that the first output is obtained at the third clock edge, and then one output at every clock edge. This effectively increases the throughput, which is the main reason systolic architecture is used. Without a systolic array, the data path of the convolution module would have 3 multipliers. One clock period would then have to accommodate such a large combinational delay, which decreases the operating frequency. By using systolic arrays, the delay is reduced to 1 multiplier, which enables operation in relatively lower frequencies. The overall delay of a one convolution operation remains the same, which is three clock cycles (with DSPs for the multiplier), but the amount of data that is processed in the same time, which is known as the throughput, is increased.

The column kernel is the difference of the elements in the previous row and next row of the current element. To get both the spatial gradients of a pixel in the same clock cycle, 3 pixels of the same column but three consecutive rows are needed, which is given by the interpolation module. Comparing Fig.10(a) and Fig.10(b), the outputs of row kernel is delayed by 2 clock cycles compared to the column kernel, for the same element (or pixel). But, a data read from the FIFO is delayed by 1 clock cycle, so the first element of the column enters the column kernel operation in clock cycle 2. The entire cycle is consumed for processing and its output at clock cycle 3, which completely matches with the first output of the row kernel. Hence, both spatial gradient values of a pixel are output initially at the third clock cycle, then one output per cycle. This is why no delay element is required in the next module, when the G matrix is being computed. The next section describes this module.

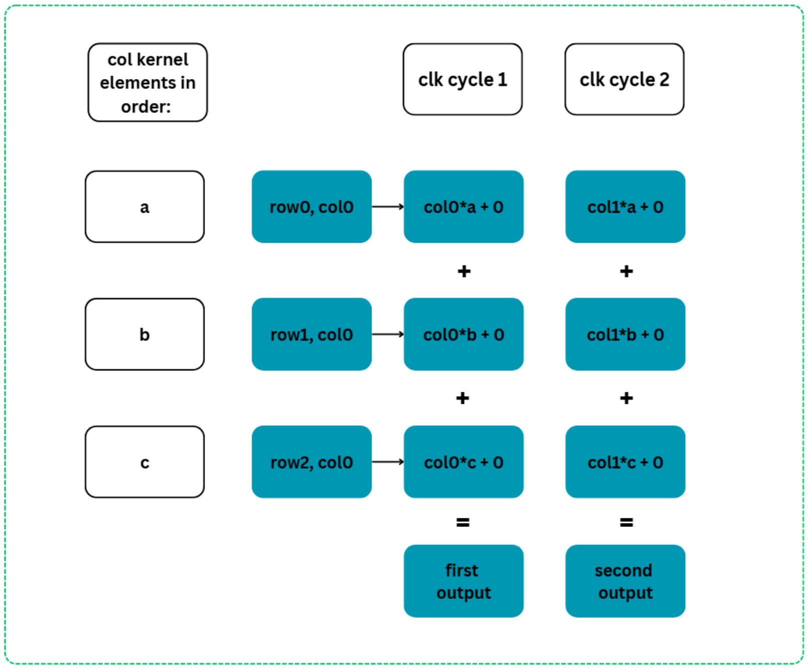


Fig.10(b). The figure depicts the operation of a column kernel in the system when inputs are given instantly, and not read from memory or FIFO. The delay from read is not accounted for, here.

5.4.2. G matrix construction and Temporal Gradient Calculation

While the spatial image gradients are being computed, the controller enables the module responsible for temporal gradient calculation. Hence, both spatial and temporal gradients for a pixel are calculated in parallel, but with some delays. From the last part of the previous section, it is clear that both the spatial gradient values are available two clock cycles after the pixel is input. For proper G matrix and b vector computation, all the three gradient values of a pixel has to be available as the same time. Since the temporal gradient uses a subtractor for the operation, the results of which are given in the same clock cycle, the input pixels must be delayed by 2 clock cycles to be in sync with the spatial gradients of the same pixel. Once all the image gradients are available at the same clock edge for a pixel, they are accumulated over 25 clock cycles, 25 because the project uses a 5x5 window around the feature pixel. For a window size of Wr x Wc, all the elements of the LK equation, that is, G matrix and b vector, can be found at the end of (Wr*Wc)th clock cycle. This is possible only when the pixel streaming logic is as described in section 5.1. For this project, at the end of 25 (5x5) clock cycles, all the elements required to solve the LK equation are available. The solution is given using Eq. 5.9. Once the optical flow vector is available, it is used for iterative refinement, as explained in the next section.

$$\text{Optical flow vector } d = (G^{-1}) * b \qquad \text{Eq. 5.9}$$

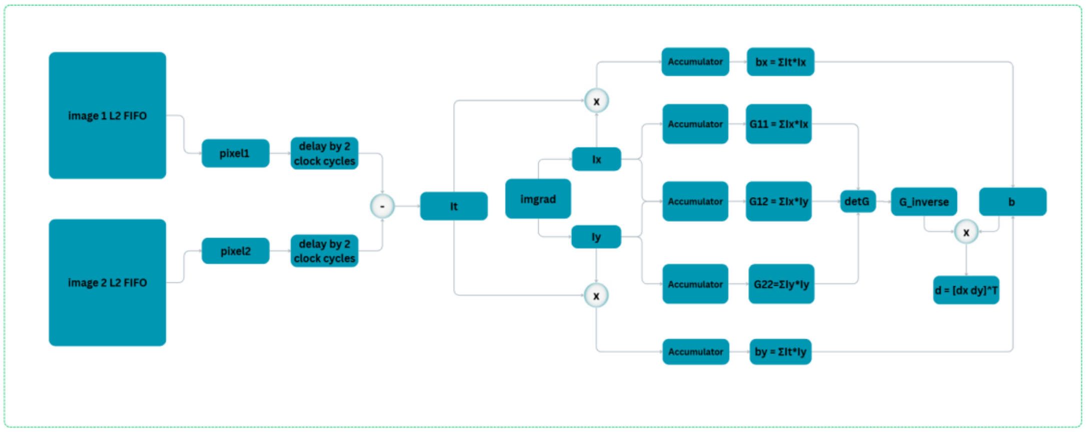


Fig.11. The figure shows the block diagram of the module responsible for finding the temporal gradient, G matrix and b vector construction.

5.4.3. Iterative Refinement

The data path explained in sections 5.4.1 and 5.4.2 is for one iteration. Lucas Kanade optical flow algorithm includes iterative refinement for better accuracy. Once the current iteration, say k, produces optical flow vectors, they need to be added to the accumulated flow vectors from previous iterations. This accumulated value reflects the true optical flow vector that has been calculated within the current layer of the pyramid. This value is further divided into fractional and integer parts. The integer value is used by the L2 FIFO for the second image to access the shifted window. Every time a new ‘d’ is found through an iteration, and added to the existing value of flow vector, the window access from the said FIFO is closer to the feature pixel the algorithm is searching for. The fractional value is used by the bilinear interpolation module to calculate the subpixel intensities of the new window.

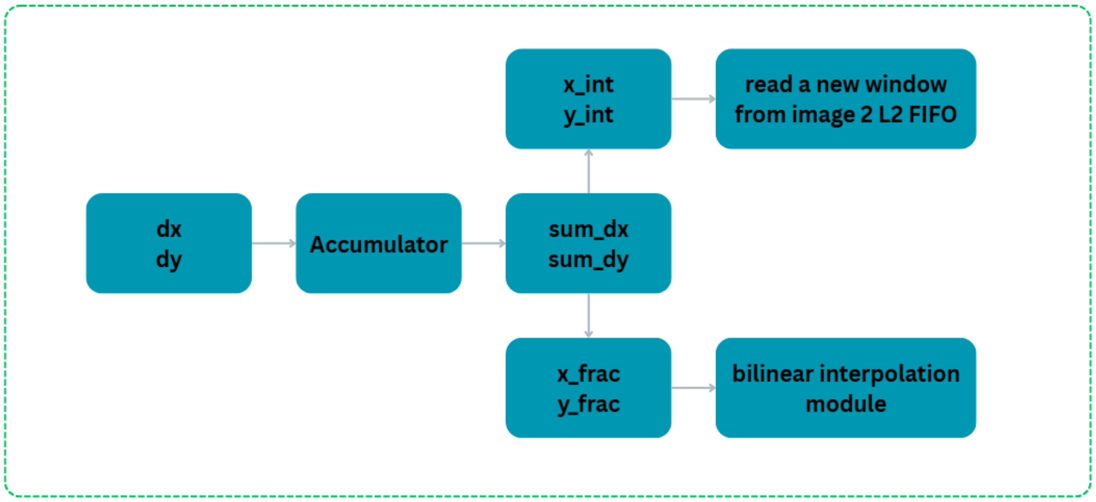

x_int
y_int
read a new window from image 2 L2 FIFO
dx
dy
Accumulator
sum_dx
sum_dy
x_frac
y_frac
bilinear interpolation module

Fig.12. This figure illustrates the connections between modules for iterative refinement as explained in section 5.4.3.

The order of this process must remain as given here, because it matters for the accuracy. Considering only the integer part of the iteration output vectors will give inaccurate results. The FIFO is designed in such a way that, the 7x7 window (two extra rows and columns over the actual 5x5 to get the correct image gradients of border pixels) is always extracted around the centre pixel of the patch that is stored in the FIFO. The centre pixel is considered as an origin. This means, the shift that is calculated as the output 'd' of an iteration is lost with respect to the FIFO. So, every time the window is accessed, the total flow vector calculated till that point in the process must be used to extract the integer part and input to the FIFO to extract the correct window. A visual representation is shown in Fig.7(b).

For the same reason as stated in Fig.7(b), the fractional part must be considered after the total flow vector has been updated. Else, the interpolation module causes accuracy problems. Once this new window is extracted, the next iteration begins, and this process continues till certain number of iterations are completed. The strategy for moving from one layer to another is two-way, either the number of iterations are reached, or the 'd' or the change between one layer to the next is less than a set threshold. These values have been decided using the MATLAB model, based on the parameters that give the least error (EPE). This design works most accurately with k=7 and eta = 0.0001. Once the total flow vector from the current pyramid layer is found, it is multiplied by 2 (left shift by 1) and given as an initial guess to the next layer of the pyramid. The multiplication by 2 is done to negate the effects of down sampling done during the pyramid generation. It is also important to note that, the flow vectors are accumulated in one location only, to account for the result from every iteration across all the layers of the pyramid. This is given as the final output.

The next section describes the control path or the FSM and gives a block diagram of the entire system.

5.5 The Control Path

A control path is typically implemented using an FSM. In this project, the FSM is mainly used to enable and disable the FIFOs, control the flow of data from one layer to the next, keep track of the iterations done within a layer and enable-disable the data path modules as appropriate. The FSM used here has 18 states, distributed as:

- 7 INIT or initialisation states, where no data processing happens but used to set/reset/load data/address into appropriate modules.
- 2 states to mark the start and end of the process.
- 9 states where data processing happens.

TABLE I gives a brief explanation about each state, its purpose, the conditions for transition and the output signals generated in each state. Fig.13. gives the state transition diagram for the FSM described in TABLE I. Fig.14. gives the block diagram after the data path and the control path has been integrated. The next section explains the testing done on the system, module wise and for the system as a whole.

To summarise the system, there are 6 FIFOs, 1 per layer and 3 per image. Two outputs of one FIFO are connected to a bilinear interpolation module. Except for the L0 FIFO of image 1, which does not require interpolation, all the other FIFOs have 3 interpolation modules each, to give a total of 15 such modules. Each layer FIFO of image 1 is connected to a separate module that calculates spatial gradient, so 3 such modules in total. Every layer FIFO of image 2 is connected to one module which calculates temporal gradient and b vector.

TABLE I

State Transition Table

CURRENT STATE AND ITS PURPOSE	NEXT STATE	OUTPUT SIGNALS
START The defaults state of the system, where all modules are reset and no data processing happens.	When the start signal is asserted, FSM transitions into INIT_1	A signal called 'busy' is asserted when the FSM leaves the START state. This signal indicates that the system is enabled. The start address for patch extraction is calculated from the input pixel coordinates and output into the address generator.
INIT_1 An intermediate state to make transition from previous state to next state cleaner.	There are no control signals for transition from this state. In the next clock cycle, the FSM transitions to EXTRACT_PYR_GEN.	No output signals originate from this state.
EXTRACT_PYR_GEN Responsible for patch extraction and pyramid generation. The address generator responsible for patch extraction is enabled.	After the entire patch is extracted, when the patch_done signal goes high, the FSM transitions to INIT_2 state. Till then, the FSM remains in EXTRACT_PYR_GEN state. If the address output by the address generator is invalid (beyond the size of the image memory), the FSM returns to the START state and remains there until started again. Invalid address is prioritised over patch_done, because when the address is invalid it makes no sense to continue extracting the patch. This logic is thus implemented using an if-else ladder.	The addr_gen signal is asserted, which activates the address generator. Within the module, the column counter starts counting when the col_count_en signal is asserted. Without this signal, the counter runs freely when in other states, leading to unnecessary dynamic power dissipation.
INIT_2 Loads the address to start reading the 7x7 window around the centre pixel from L2 FIFO of image1 and image2.	No external control signals for the transition from this state. In the next clock cycle, the FSM transitions from INIT_2 to FIND_G.	The load_addr2 signal is asserted to load the said address into the read pointer of L2 FIFO of image1 and image2. Every time load_addr2 is asserted, the same address is loaded into L2 FIFO of image 1, but the address loaded into L2 FIFO of image 2 is different depending on the flow vector value. The 7x7 window extracted for image 1 remains the same throughout the process because the flow vector doesn't affect it.
FIND_G Starts the LK process from layer 2 of the pyramid. Enables	When the L2 FIFO finishes reading the 7x7 window, as given by the wn_done2 signal, the FSM transitions to	The fifo2_r_en signal is asserted to enable the L2 FIFO. The imgrad_en2 signal is asserted

the L2 FIFO of image 1 and image 2. Enables spatial image gradient calculation as the FIFO stream the pixels	FIND_D_L2. Until then, it remains in FIND_G.	which enables the spatial image gradient calculation. The drdc_en signal, asserted for all states beyond FIND_G, is responsible to direct the flow vector integer part to the image 2 FIFOs to enable extraction of new windows. When this signal is not asserted, the input is zero since the flow vectors have not been computed and aren't relevant.
FIND_D_L2 Finds the flow vector for k=0 iteration for L2. The g inverse calculation module is enabled which considers the G elements calculated in the previous state. The L2 FIFOs are disabled to stop pixel streaming.	Once a valid flow vector is found, given by valid_det signal which is asserted when the determinant of G is zero, the FSM transitions into INIT_3 state, else it stays in FIND_D_L2. The LK equation can be solved only when the determinant of G is non-zero, hence the use of 'valid_det' signal to indicate a valid flow vector is appropriate.	The ginv_en signal is asserted to enable the module that calculates G matrix's inverse.
INIT_3 Resets Image gradient module and the b vector construction module to start calculations anew for the next iteration. Loads the start address of 7x7 window into L2 FIFO of image 1 and image 2 to start pixel streaming.	No external control signals for the transition. In the next clock cycle, the FSM transitions into L2_ITERATE state.	The imgrad_rst2 and b_reset signal asserted to reset image gradient and b construction module. The 'load_addr2' signal asserted to load the address into L2 FIFO.
L2_ITERATE Streams the pixels from L2 FIFO for every iteration. Enables spatial image gradient and b vector calculation.	When wn_done2 is asserted, which denotes that the 7x7 window has been streamed, the FSM transitions into L2_CHECK. Else, it stays in L2_ITERATE.	The imgrad_en2 and fifo2_r_en signals are asserted.
L2_CHECK Enables G matrix inverse calculation for the G matrix calculated in L2_ITERATE state. Keeps track of the iterations within the layer. Provides the two exit strategies to move from layer 2 to layer 1.	If a valid flow vector is not found, as given by valid_det signal, the FSM transitions into STOP state, stopping the tracking for currently input pixel. Else, the exit strategies are checked. If k == k_thresh or eta_thresh_pass_d is asserted (denotes eta < eta_thresh), the FSM transitions into the INIT4_1 state. If valid_det signal is asserted by the exit conditions are not met, the FSM transitions to INIT_3 to start the next iteration.	The ginv_en signal is asserted to enable the flow vector calculation. Every time the valid_det signal is asserted, a valid flow vector is calculated, hence k increments by 1. This is controlled by the k_en signal which is equal to the valid_det signal.

INIT4_1 Resets k counter so that the iteration counter starts anew for the next layer of the pyramid and enables multiplication by 2 for the final flow vector of layer 2 of the pyramid.	No external control signals. In the next clock cycle, the FSM transitions to the INIT_4 state.	The k_reset signal is asserted to reset the k counter, and shift_d signal is asserted to enable multiplication by 2 (which is basically shift left by 1).
INIT_4 Resets Image gradient module and the b vector construction module to start calculations anew for the next iteration. Loads the start address of 7x7 window into L1 FIFO of image 1 and image 2 to start pixel streaming.	No external control signals for the transition. In the next clock cycle, the FSM transitions into L1_ITERATE state.	The imgrad_rst1 and b_reset signal asserted to reset image gradient and b construction module. The 'load_addr1' signal asserted to load the address into L1 FIFO.
L1_ITERATE Streams the pixels from L1 FIFO for every iteration. Enables spatial image gradient and b vector calculation.	When wn_done1 is asserted, the FSM transitions into L1_CHECK. Else, it stays in L1_ITERATE.	The imgrad_en1 signal is asserted to enable the image gradient calculation and fifo1_r_en signal is asserted to allow pixel streaming from L1 FIFO.
L1_CHECK Enables G matrix inverse calculation for the G matrix calculated in L1_ITERATE state. Keeps track of the iterations within the layer. Provides the two exit strategies to move from layer 1 to layer 0.	If a valid flow vector is not found, the FSM transitions into STOP state, stopping the tracking for currently input pixel. Else, the exit strategies are checked. If k == k_thresh or eta_thresh_pass_d is asserted (denotes eta < eta_thresh), the FSM transitions into the INIT5_1 state. If valid_det signal is asserted by the exit conditions are not met, the FSM transitions to INIT_4 to start the next iteration.	The ginv_en signal is asserted to enable the flow vector calculation. The k counter is incremented by 1 every time valid_det is high.
INIT5_1 Resets k counter and enables multiplication by 2 for the final flow vector of layer 1 of the pyramid.	No external control signals. In the next clock cycle, the FSM transitions to the INIT_5 state.	The k_reset signal is asserted to reset the k counter, and shift_d signal is asserted to enable multiplication by 2 (which is basically shift left by 1).
INIT_5 Resets Image gradient module and the b vector construction module. Loads the start address	No external control signals for the transition. In the next clock cycle, the FSM transitions into L0_ITERATE state.	The imgrad_rst0 and b_reset signal asserted to reset image gradient and b construction module. The 'load_addr0' signal asserted to load the address into L1 FIFO.

of 7x7 window into L0 FIFO of image 1 and image 2 to start pixel streaming		
L0_TERATE Streams the pixels from L0 FIFO for every iteration. Enables spatial image gradient and b vector calculation.	When wn_done0 is asserted, the FSM transitions into L0_CHECK. Else, it stays in L0_ITERATE.	The imgrad_en0 signal is asserted to enable the image gradient calculation and fifo0_r_en signal is asserted to allow pixel streaming from L0 FIFO.
L0_CHECK Enables G matrix inverse calculation for the G matrix calculated in L0_ITERATE state. Keeps track of the iterations within the layer. Provides the two exit strategies to end the process and give the final optical flow vector.	If a valid flow vector is not found, the FSM transitions into STOP state, stopping the tracking for currently input pixel. Else, the exit strategies are checked. If k == k_thresh or eta_thresh_pass_d is asserted (denotes eta < eta_thresh), the FSM transitions into the STOP state. If valid_det signal is asserted by the exit conditions are not met, the FSM transitions to INIT_5 to start the next iteration.	The ginv_en signal is asserted to enable the flow vector calculation. The k counter is incremented by 1 every time valid_det is high.
STOP The last state of the FSM. Concludes the entire system.	No external control signals for the transition. In the next clock cycle the FSM transitions to START state and stays there until start signal is asserted the next time.	The busy signal is de-asserted, denoting that the system has completed the process.

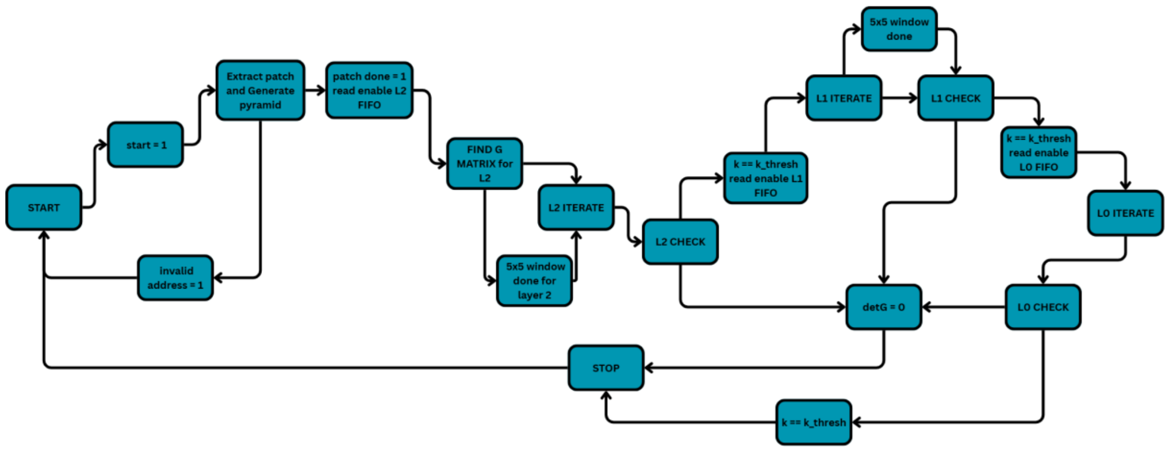

START
start = 1
Extract patch and Generate pyramid
patch done = 1 read enable L2 FIFO
invalid address = 1
FIND G MATRIX for L2
5x5 window done for layer 2
L2 ITERATE
L2 CHECK
k == k_thresh read enable L1 FIFO
L1 ITERATE
5x5 window done
L1 CHECK
k == k_thresh read enable L0 FIFO
L0 ITERATE
L0 CHECK
detG = 0
STOP
k == k_thresh

Fig.13. This figure illustrates the state transition diagram associated with the FSM discussed in section 5.5.

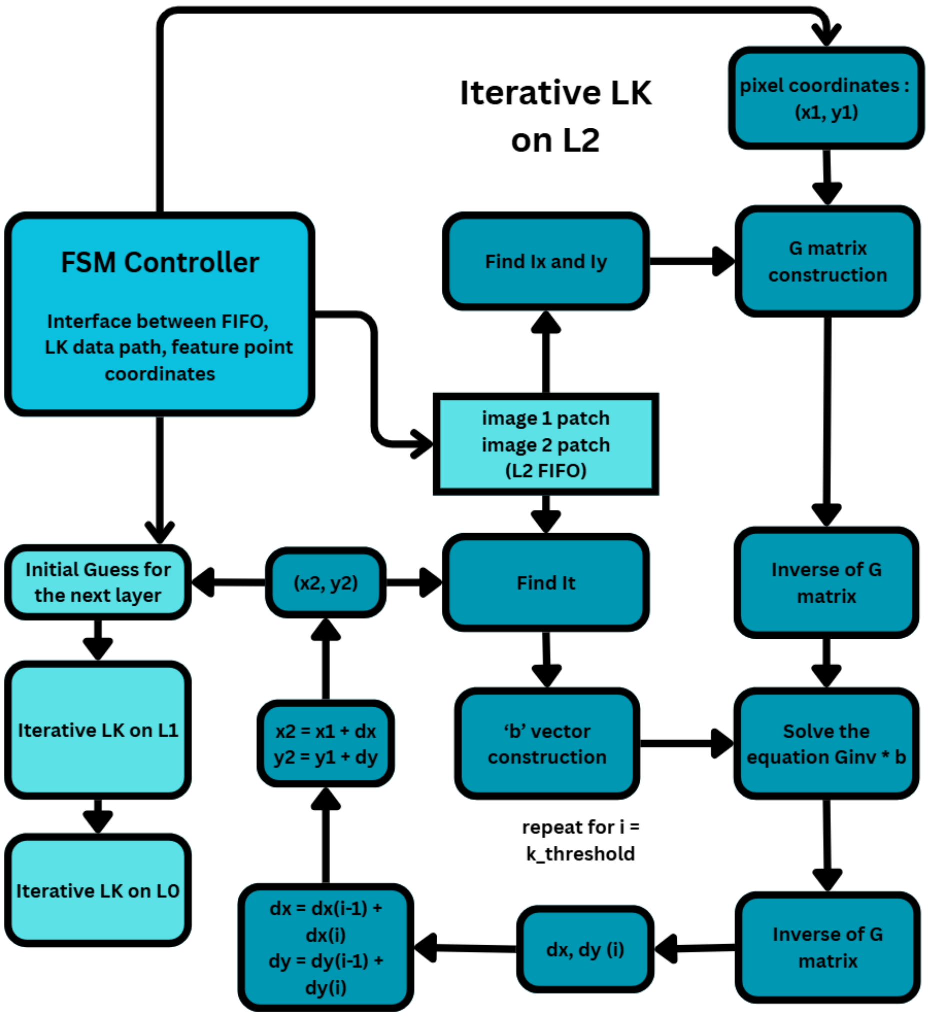


Fig.14. The figure illustrates the block diagram after integration of all the modules and the FSM.

5.6 Testing the Individual Modules

Before the system is integrated, each module must be tested independently. For that, the simulation flow adopted is given in Fig.[]. While testing individual modules, the previous module's output is unavailable, hence they need to be generated using MATLAB, using models that are coded based on the logic implemented in hardware or using built-in functions. It is important to note that the entire system works on row major order. So, the input must be given in that format. Based on the error between the MATLAB output and the Verilog outputs, changes are made to the code until the error is acceptable (or zero for simple operations). The detailed flow for each module's test is given as block diagrams in Fig.16 to Fig.23.

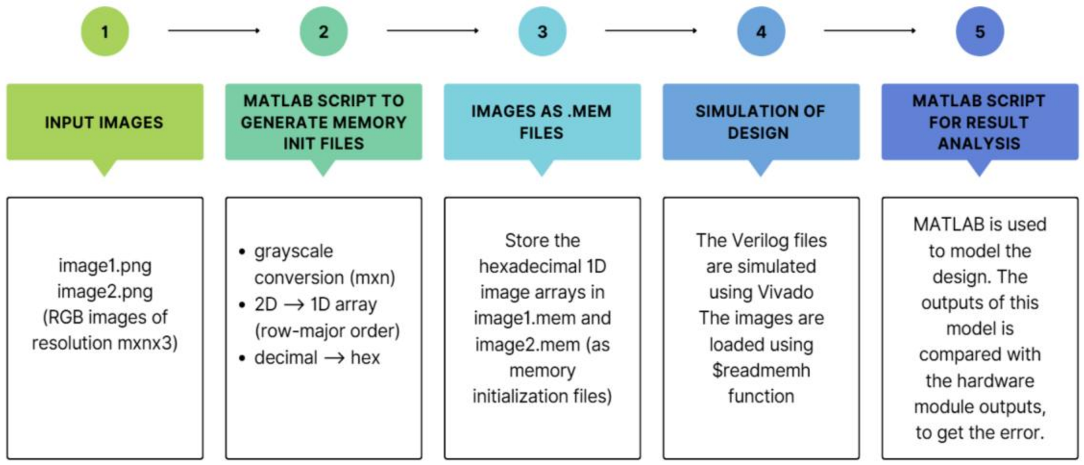

1
INPUT IMAGES
image1.png
image2.png
(RGB images of resolution mxnx3)
2
MATLAB SCRIPT TO GENERATE MEMORY INIT FILES
• grayscale conversion (mxn)
• 2D → 1D array (row-major order)
• decimal → hex
3
IMAGES AS .MEM FILES
Store the hexadecimal 1D image arrays in image1.mem and image2.mem (as memory initialization files)
4
SIMULATION OF DESIGN
The Verilog files are simulated using Vivado The images are loaded using $readmemh function
5
MATLAB SCRIPT FOR RESULT ANALYSIS
MATLAB is used to model the design. The outputs of this model is compared with the hardware module outputs, to get the error.

Fig.15. The simulation flow for testing individual modules. Initially, the input images are test matrices, generated by MATLAB. These are later replaced by images from KITTI 2015 flow dataset or MPI Sintel dataset.

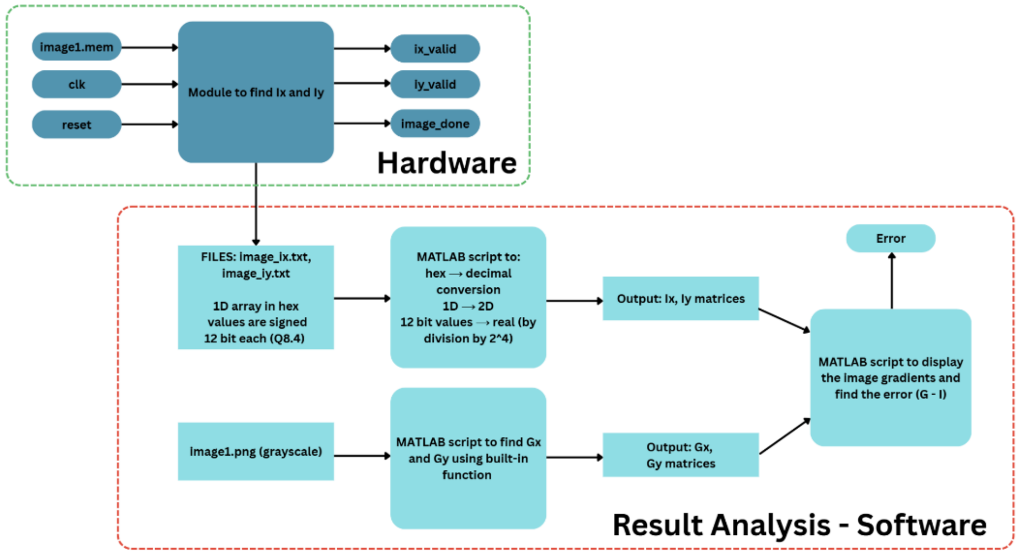


image1.mem
clk
reset
Module to find Ix and Iy
Ix_valid
Iy_valid
image_done
Hardware
FILES: image_Ix.txt, image_Iy.txt
1D array in hex values are signed 12 bit each (Q8.4)
MATLAB script to: hex → decimal conversion 1D → 2D 12 bit values → real (by division by 2^4)
Output: Ix, Iy matrices
Error
MATLAB script to display the image gradients and find the error (G - I)
image1.png (grayscale)
MATLAB script to find Gx and Gy using built-in function
Output: Gx, Gy matrices
Result Analysis - Software

Fig.16. The flow to test the spatial image gradient module.

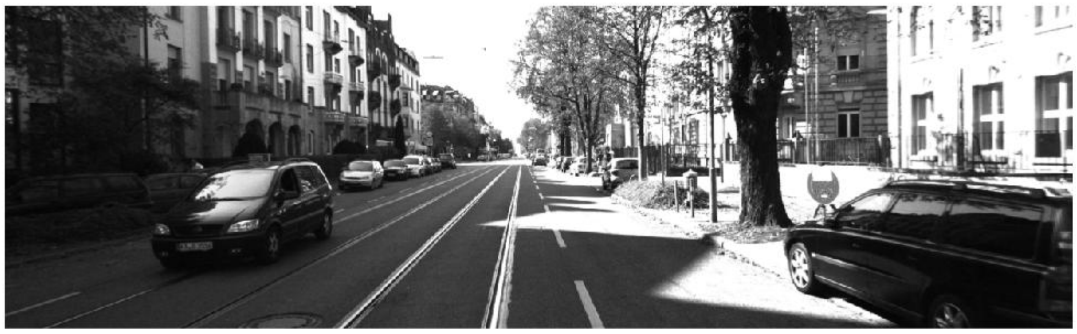


Fig.17(a) An image from KITTI 2015 flow dataset for testing the image gradient modules.

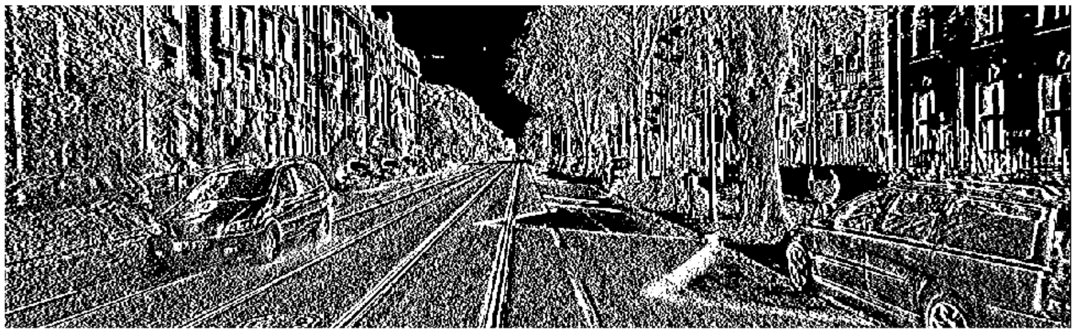


Fig.17(b) The output Ix from the image gradient module. The module output is a 1D array written into a file. A MATLAB script is used to convert it into an image format.

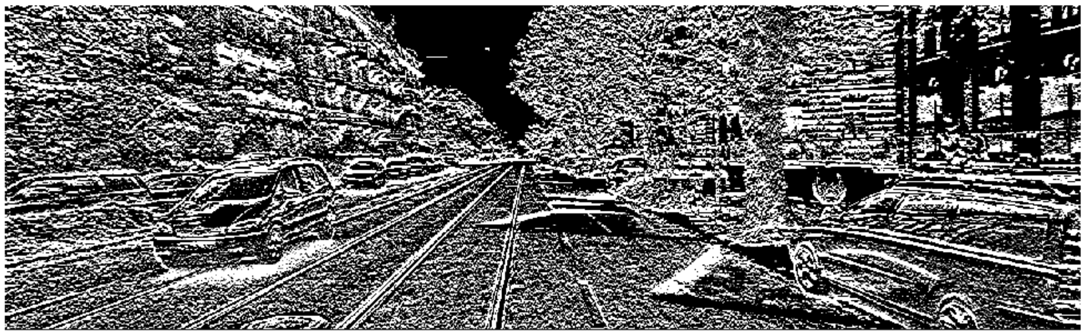


Fig.17(c) The output Iy from the image gradient module as reconstructed by MATLAB.

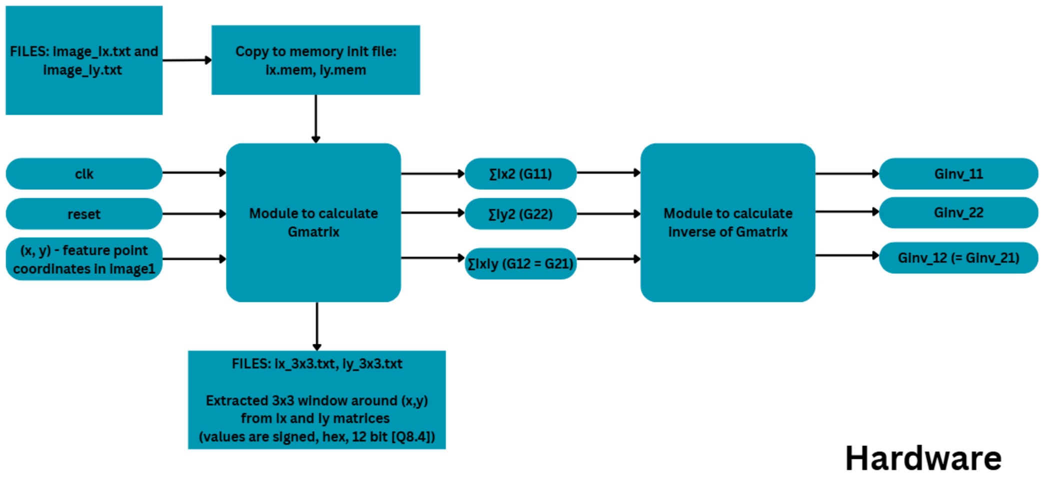

FILES: image_Ix.txt and image_Iy.txt
Copy to memory init file: Ix.mem, Iy.mem
clk
reset
(x, y) - feature point coordinates in image1
Module to calculate Gmatrix
∑Ix2 (G11)
∑Iy2 (G22)
∑IxIy (G12 = G21)
Module to calculate inverse of Gmatrix
Ginv_11
Ginv_22
Ginv_12 (= Ginv_21)
FILES: Ix_3x3.txt, Iy_3x3.txt
Extracted 3x3 window around (x,y) from Ix and Iy matrices (values are signed, hex, 12 bit [Q8.4])
Hardware

Fig.18. The flow of G matrix construction and G inverse calculation module. Since they are only combinational blocks, the outputs were not written into any file but directly compared in MATLAB command window.

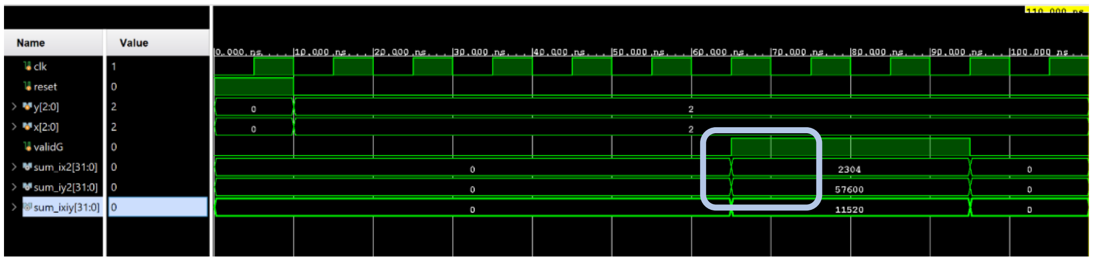
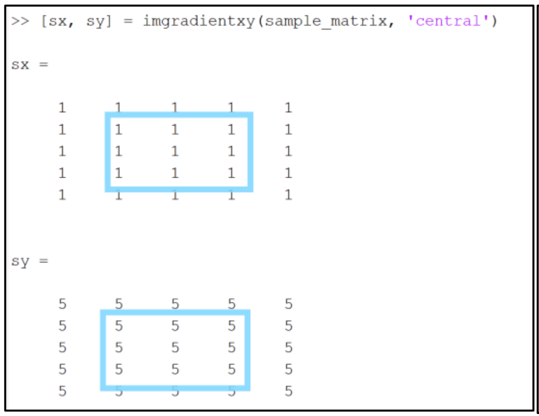


Fig.19(a). This figure shows a snapshot of the simulation window during the testing of G matrix module.

To test, a sample 5x5 matrix is considered with values 1 to 25 (row major).



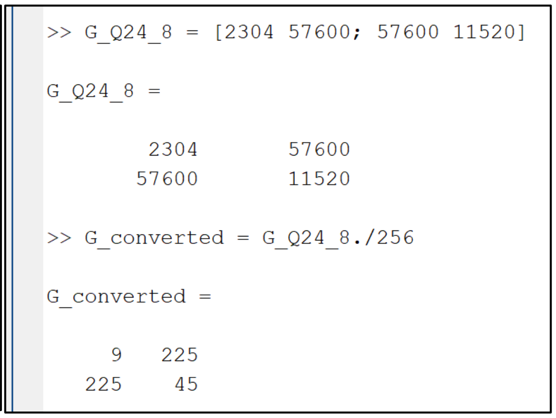
```
>> [sx, sy] = imgradientxy(sample_matrix, 'central')

sx =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1


sy =

     5     5     5     5     5
     5     5     5     5     5
     5     5     5     5     5
     5     5     5     5     5
     5     5     5     5     5
```



```
>> G_Q24_8 = [2304 57600; 57600 11520]

G_Q24_8 =

        2304       57600
       57600       11520

>> G_converted = G_Q24_8./256

G_converted =

     9   225
   225    45
```

Fig.19(b). The snapshot on the left shows the spatial gradients of the sample matrix using built in function in MATLAB. In the initial phases, a 3x3 window around the centre pixel was considered (which was then modified to 5x5 window). The snapshot on the right shows the module output and it’s conversion into fixed point notation (Q24.8) using division by 256. Using simple math, the output can be verified against the given image gradient matrices.

With operations like division, inverse and such, error is usually non-zero because 4-bit (Q8.4) precision is used while MATLAB uses double precision. No matter the number of fractional bits available, it is observed that there is always a non-zero error, despite it being small. That is, perhaps, inherent to fixed point representation.

Referring to Fig.23., it is important to note that it is necessary to shift the G matrix by 4 bits to the left before continuing with the calculation of it’s inverse as given by Eq. 5.10 to Eq. 5.15.

g11 → Q8.4 Eq. 5.10

g11*g22 → (Q.8.4)*(Q8.4) → Q16.8 Eq. 5.11

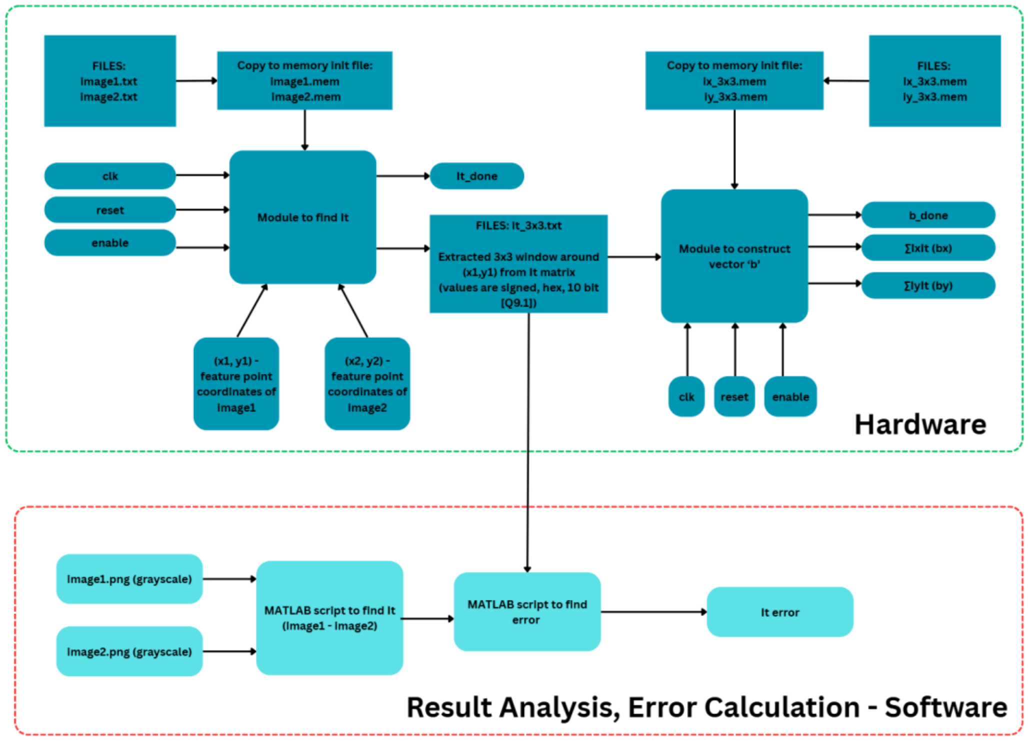

FILES:
Image1.txt
Image2.txt
Copy to memory init file:
Image1.mem
Image2.mem
Copy to memory init file:
Ix_3x3.mem
Iy_3x3.mem
FILES:
Ix_3x3.mem
Iy_3x3.mem
clk
reset
enable
Module to find It
It_done
FILES: It_3x3.txt
Extracted 3x3 window around (x1,y1) from It matrix (values are signed, hex, 10 bit [Q9.1])
Module to construct vector 'b'
b_done
ΣIxIt (bx)
ΣIyIt (by)
(x1, y1) - feature point coordinates of Image1
(x2, y2) - feature point coordinates of Image2
clk
reset
enable
Hardware
Image1.png (grayscale)
Image2.png (grayscale)
MATLAB script to find It (Image1 - Image2)
MATLAB script to find error
It error
Result Analysis, Error Calculation - Software

Fig.20. This figure illustrates the flow of testing the module responsible for temporal gradient calculation and b vector construction. To test the module that calculates the b vector, spatial gradients must be supplied through a file and temporal gradients are given by the module responsible for it. Typically, designing and testing these two modules makes things easier. Similarly, the modules responsible for spatial gradient calculation and G matrix construction are designed and tested together.

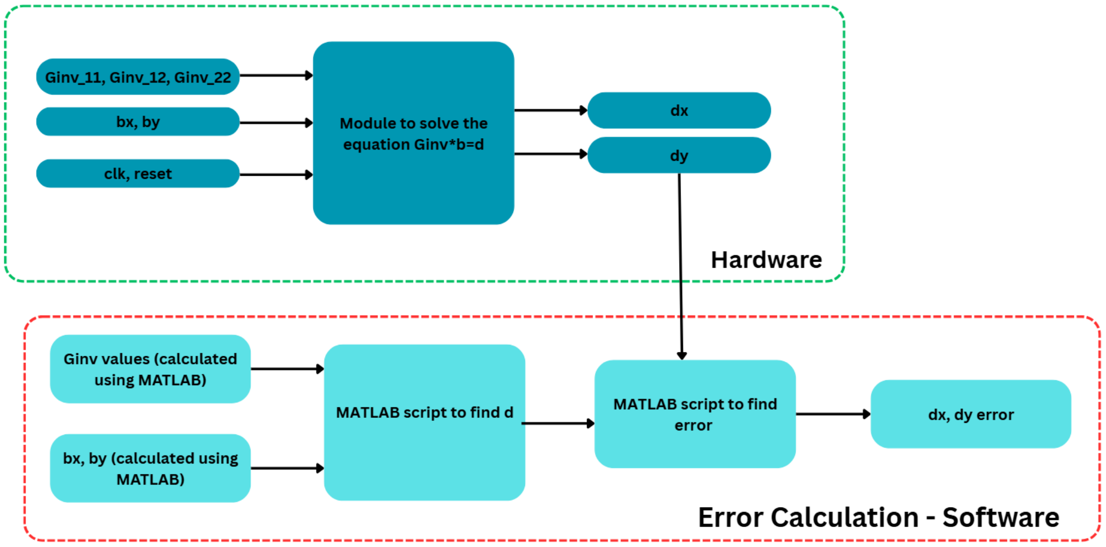

Ginv_11, Ginv_12, Ginv_22
bx, by
clk, reset
Module to solve the equation Ginv*b=d
dx
dy
Hardware
Ginv values (calculated using MATLAB)
bx, by (calculated using MATLAB)
MATLAB script to find d
MATLAB script to find error
dx, dy error
Error Calculation - Software

Fig.21. The flow for testing the module that calculates the G inverse values.

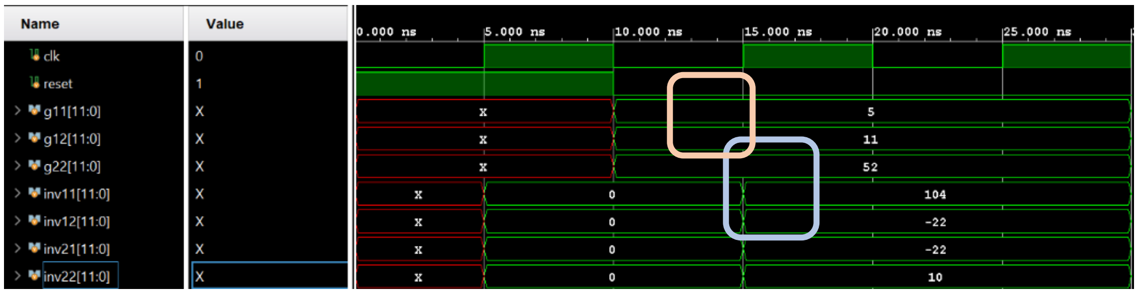
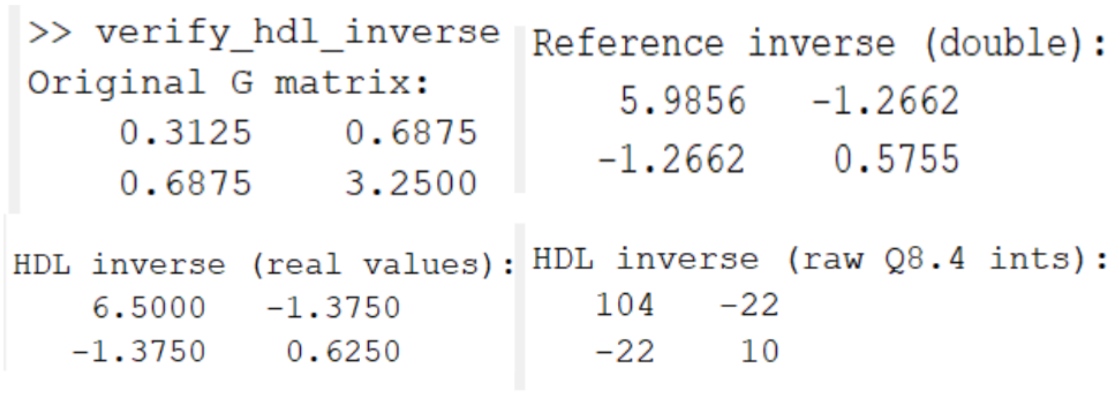


Fig.22. shows a simulation snapshot of the module that calculates the inverse of the G matrix. The values bounded in blue and orange box are the values of interest here. The G matrix values (bounded by the orange box) are calculated by multiplying the original G matrix by 16. The inverse G values (bounded by blue box) need to be divided by 16 to get the fixed-point representation Q8.4.



```
>> verify_hdl_inverse
Original G matrix:
    0.3125    0.6875
    0.6875    3.2500

Reference inverse (double):
    5.9856   -1.2662
   -1.2662    0.5755

HDL inverse (real values):
    6.5000   -1.3750
   -1.3750    0.6250

HDL inverse (raw Q8.4 ints):
   104   -22
   -22    10
```

Fig.23. On the top left, the original G matrix is given. On the top right, the inverse of G using built-in funtion is given. On the bottom right, the Verilog code outputs are given and on the bottom left, the converted values (Verilog output divided by 16) are given.

Det(G) = g11*g22 – g12*g21 $\rightarrow$ (Q.8.4)*(Q8.4) - (Q.8.4)*(Q8.4) = Q16.8

Eq. 5.12

ginv11 = g22/det(G) $\rightarrow$ (Q8.4)/(Q16.8) Eq. 5.13

ginv12 = ginv21 = -g12/det(G) $\rightarrow$ (Q8.4)/(Q16.8) Eq. 5.14

ginv22 = g11/det(G) $\rightarrow$ (Q8.4)/(Q16.8) Eq. 5.15

From Eq. 5.13, the fractional part = 4 – 8 = -4 (invalid). The number of bits in numerator is less than in denominator. The division operator gives 0 as the output for such cases. Hence the number must have atleast the same number of bits as the denominator, which is why the original G matrix must be shifted left by 4 (or multiplied by 16) and signed extended before using to solve Eq. 5.13, Eq. 5.14 and Eq. 5.15. Now, the equations change into Eq. 5.16, Eq. 5.17 and Eq. 5.18.

ginv11 = g22/det(G) → (Q16.8)/(Q16.8) Eq. 5.16

ginv12 = ginv21 = -g12/det(G) → (Q16.8)/(Q16.8) Eq. 5.17

ginv22 = g11/det(G) → (Q16.8)/(Q16.8) Eq. 5.18

Now, to preserve the Q8.4 representation, the module internally divides the G inverse values by 16 (arithmetic right shift by 4, arithmetic shift used to preserve sign). The module output is then divided by 16 again (arithmetic right shift by 4), so the G inverse values are shifted by 8 bits in total (or divided by 256) to get the fixed-point representation in MATLAB. This is an extremely important point to note; else all the inverse values are misinterpreted. If the outputs are not matching, it is recommended to keep a reference G inverse matrix (using MATLAB) and perform division using different powers of 2. The correct power of 2 (or the correct number of fractional bits) give the closed result to the reference.
The next section moves to testing the integrated system.

5.7 Testing the Integrated System

The integrated testing was done in 3 phases. When moving from one phase to the next, the complexity of the input images was increased. This makes debugging much easier and the transition from one phase to the next easier. Once the lower phases give satisfactory results, it increases the confidence in the design, and helps identify parts of the system that are working perfectly, and the parts that aren't. This helps narrow down the possibilities during the debugging process, when in the code does not work in the later phases. For example, if the output of a FIFO is x's, then the most likely cause would be out of bound read pointer, or poorly handled boundary conditions (for pixels close to the image boundary). If a simulation stops in the mid of a process and only outputs x's, then that's because the zero determinant condition is not handled properly. If 'd' values keep increasing from iteration to iteration, it's likely because the wrong pixels are being streamed. The best way to rectify wrong pixel streaming logic is to consider a small image, trace the pixel flow manually and change the streaming logic appropriately.

The most important requirement when debugging is to have and logical idea or a lead of where things could be going wrong. Blindly changing the code leads to more errors, that sometime accumulate or sometimes cancel each other but are still present but cannot be known.

The simulation flow used for testing the individual modules was used in Phase 1, refer to Fig.15, but the simulation flow in Fig.24 was used in Phase 2 and Phase 3.

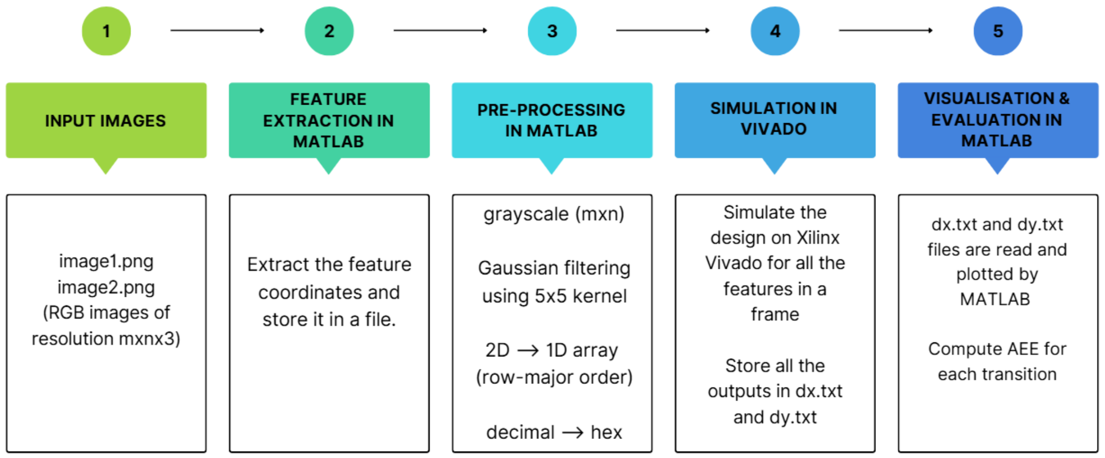

1
2
3
4
5
INPUT IMAGES
FEATURE EXTRACTION IN MATLAB
PRE-PROCESSING IN MATLAB
SIMULATION IN VIVADO
VISUALISATION & EVALUATION IN MATLAB
image1.png image2.png (RGB images of resolution mxnx3)
Extract the feature coordinates and store it in a file.
grayscale (mxn)
Gaussian filtering using 5x5 kernel
2D → 1D array (row-major order)
decimal → hex
Simulate the design on Xilinx Vivado for all the features in a frame
Store all the outputs in dx.txt and dy.txt
dx.txt and dy.txt files are read and plotted by MATLAB
Compute AEE for each transition

Fig.24. The simulation flow used to perform Phase 2 and Phase 3 of testing on the integrated system. The experiments, too, were performed using this flow.

5.7.1 Testing the Integrated System – Phase 1

In this phase, a MATLAB generated 33x33 matrix was given as input images. The matrix has a single grey point (the feature point), surrounded by 24 white pixels. The rest of the image is black. Using this, the system is given the simplest feature point to track: it is unique (there is no other pixels with similar intensities anywhere in the image, so theoretically, the system must track this point perfectly), the white window around the feature point must make finding the feature pixel easier and the rest of the image is black (intensity = 0) so this simplifies all calculations.

Typically, without the pyramid, the LK algorithm tracks a point to the nearest pixel with the most similar intensity value. This becomes a problem where there are multiple pixels with similar intensity values near the feature pixel. The image pyramid helps point the LK algorithm in the right direction to reduce this problem. The uniform back ground in the test image is considered to eliminate this possibility during initial testing.

Further, because the calculations are so simple, it makes it easier to track the issue back to its source. When the first few tests are preformed on a newly integrated system, it is always recommended to trace the data flow in the simulation results and compare it with MATLAB outputs. In fact, much of the problems with address calculations can be solved simply by comparing the FIFO values (when it is streaming) to the matrices stored in MATLAB workspace.

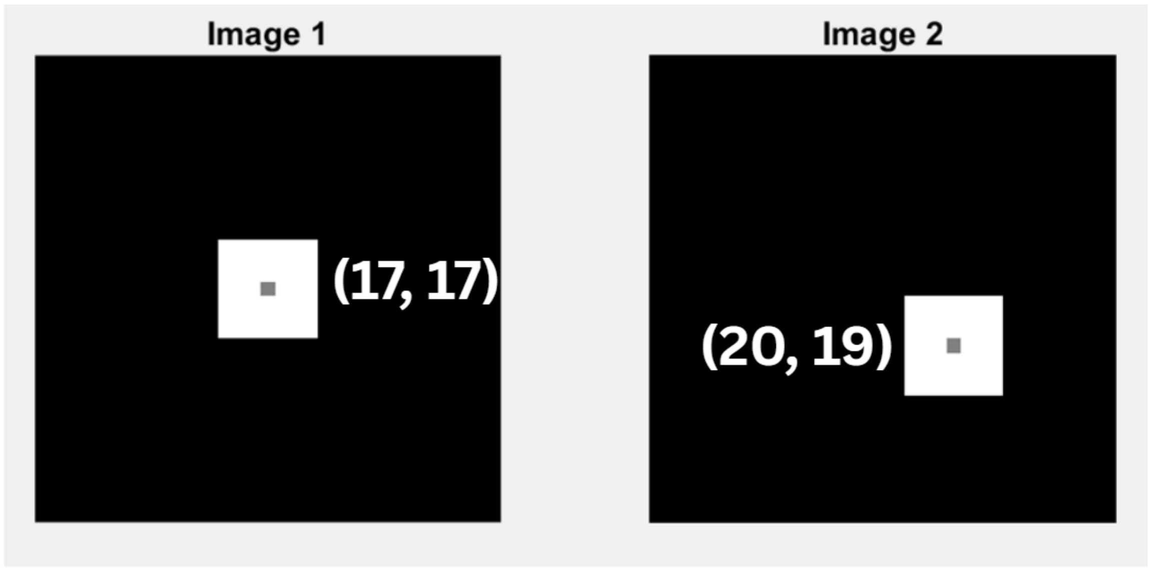
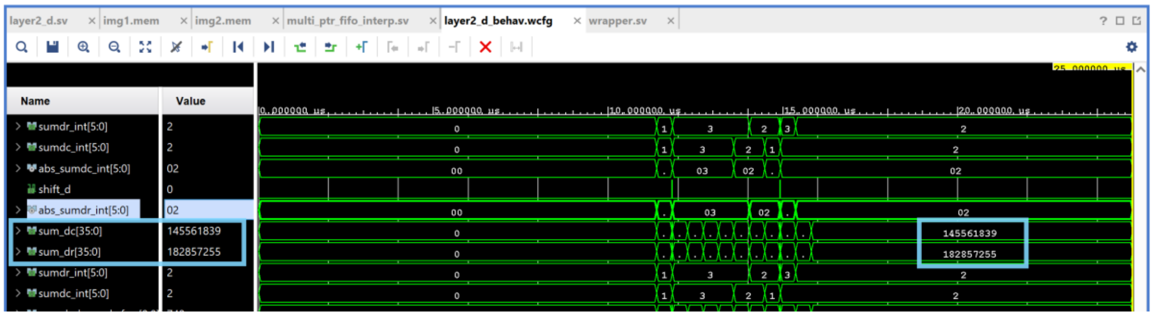
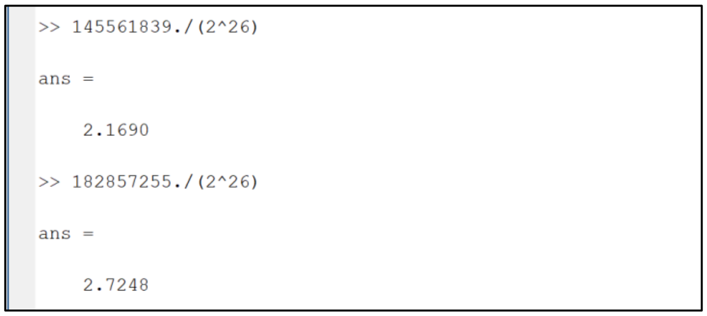


Fig.25. The images used in Phase 1 of testing the integrated system. Image 1 (on the left) is the first image, with feature point as the centre pixel. Image 2 (on the right) is the second image with shifted/moved feature point. The coordinates are labelled as (row, column). Each image is 33x33 in resolution. The ground truth is (3, 2) where 3 is the shift in rows and 2 is the shift in rows.



Name	Value
sumdr_int[5:0]	2
sumdc_int[5:0]	2
abs_sumdc_int[5:0]	02
shift_d	0
abs_sumdr_int[5:0]	02
sum_dc[35:0]	145561839
sum_dr[35:0]	182857255
sumdr_int[5:0]	2
sumdc_int[5:0]	2

Fig.26. This figure shows a snapshot of the simulation during Phase 1 of testing the integrated system. The blue box bounds the total flow as output of the Verilog module.



```
>> 145561839./(2^26)

ans =

    2.1690

>> 182857255./(2^26)

ans =

    2.7248
```

Fig.27. This figure shows the fixed point conversion of the Verilog output. The value is divided by 2^26 to preserve as much precision as possible out of a 32 bit optical flow vector. This limits the integer part of the vector to 6 bits. The error is still non-zero perhaps due to fixed point representation.

The error is considered to be acceptable, since it is less than 1 pixel. The testing now moves on to Phase 2, which is explained in detail in the next section.

5.7.2 Testing the Integrated System – Phase 2

In this phase, the complexity is increased by one level relative to Phase 1. The input images are selected from bamboo_1 sequence in MPI Sintel dataset, which is much more complex compared to the simple image input in the previous phase. One feature point with coordinates (117, 195) is considered as the point to track. The patch size remains 33x33, with k=7 and eta = 0.0001. When giving the coordinates as inputs in the testbench, it is decreased by 1 each to convert the values in one-based indexing from MATLAB to the required zero-based indexing. The results are comparable to the software values, as given in Fig.27, so the testing moves to the next phase.

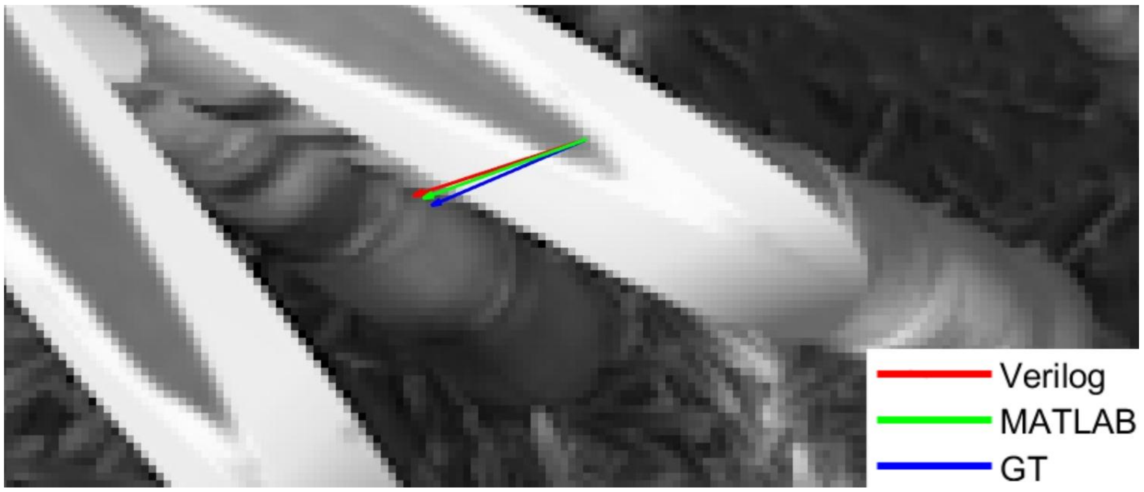


Fig.28. This figure plots the flow vector from MATLAB code, Verilog code and the ground truth. It can be seen that the Verilog output is very close to the MATLAB value which is considered to be comparable values for Phase 2 of testing. The image is a zoomed in version of the image in the dataset.

5.7.3 Testing the Integrated System – Phase 3

This phase tries to increase the complexity level by trying to track multiple features across one transition. The input images are selected from alley_1 sequence of MPI Sintel dataset. For this, the feature coordinates are read from a file (separate for the row and column coordinate) and the optical flow vector for each feature is written into another file (separate for the two values within the flow vector). This file is used to visualise the flow vectors, as can be seen in Fig.[]. The patch size remained as 33x33, k=7 and eta = 0.0001. The number of features considered is 75, extracted using Shi-Tomasi algorithm in MATLAB. It is important to note that the entire system is designed to track one pixel across one transition. Multiple features are given in sequence using a testbench. Reading and writing the relevant files is also done by the testbench.

Now that the system is tested to a certain extent, the next section explains the experiments performed on this system. These experiments intend to try using the system for some real-world applications.

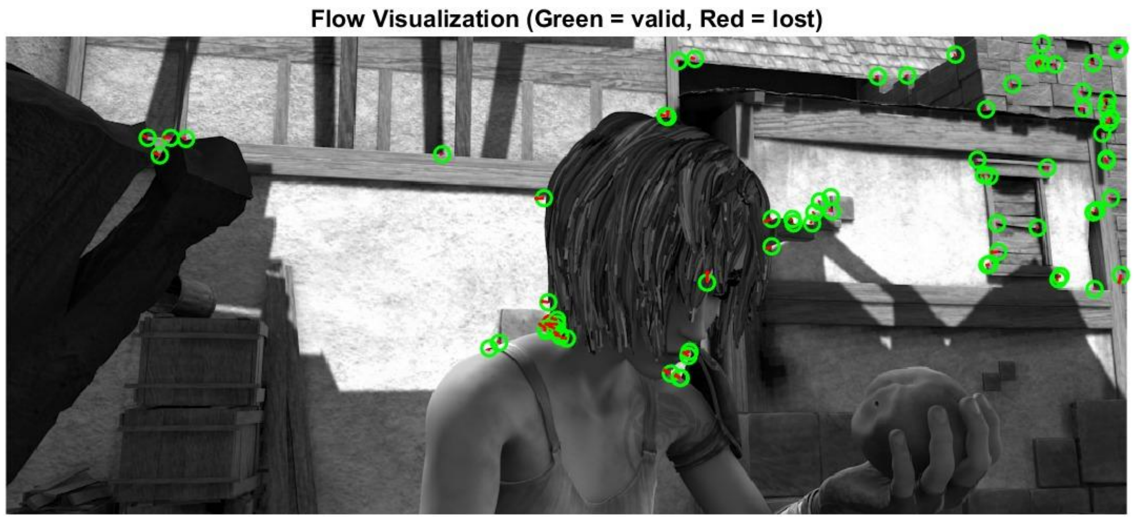


Fig.29. This figure plots all the optical flow vectors calculated by the Verilog module. One of the features is lost because of zero det(G) condition. The red arrows are the optical flow vectors.

5.8 Experiments

5.8.1 Experiment 1

Aim: Using YOLOv4 for detecting the person and tracking using Pyramidal LK

The system was used to track a person moving through the alley_2 sequence of MPI Sintel dataset. The simulation flow in Fig.[] can be used to track a point across one transition only and has not been automated for multiple frames. Hence, only 15 frames out of 50 available were used to calculate the average EPE or AEE. The patch size considered is 141x141, as decided by the MATLAB trial runs, k=7 and eta = 0.0001. The frame resolution is 436x1024, same as the images used in the testing phase (since the same dataset is used). The number of features per frame is 100.

The next section is about the results and analysis of the experiments performed on this system. This project stops at simulation and synthesis only, and records the resource utilization and the latency.

6. RESULTS AND ANALYSIS

This section contains the details on the evaluation process that our project has undergone. There were two experiments performed on the system implemented. Each of them is explained in section 5.8. The entire simulation flow used to perform the experiment for a single transition is given in Fig.24.

TABLE II

Comparison of Latency/FPS Between Various Works Implementing Optical Flow Algorithm/Pyramidal Lk

Works	Latency/FPS
[1] Real-time and efficient optical flow tracking accelerator	93 FPS
[17] Reduced memory access	196 FPS
[4] Ultra-high performance and scalable hardware accelerator	405 FPS
[8] RAFT hardware accelerator	95 ms for 1 frame
[7] Optical Flow for video under Dynamic illumination (fast version)	8 s (CPU)
[18] VLSI Architecture for Lucas Kanade	30 FPS
Shi-Tomasi and Lucas Kanade (our work)	1.5ms for 1 transition
YOLO person tracking and Lucas Kanade (our work)	22ms for 1 transition

In terms of latency, using Shi-Tomasi our work runs at 1.5ms for 1 frame with 75 features. This is better than RAFT, a deep network based optical flow algorithm. Even with YOLO feature extraction, our latency is lesser. But decreasing latency has increased the AEE. Since the latency of the system varies with patch size and the appropriate patch size needs to be experimentally found for a dataset, the comparison can only happen with another implementation that tests on the same dataset. Same for AEE as well. The only relevant papers for comparison are [8], which implements RAFT hardware accelerator, and [7], which is a completely software implementation and runs on a CPU. For the same dataset, compared all the above-mentioned papers, our work gives the least latency.

Resource utilisation for optical flow is divided into LUT count, DSPs, FFs, with respect to the data path and control path implementation, and BRAM count with respect to the memory required (for storing images or pseudo images). The memory elements utilised in this design could not be synthesized hence, only the LUT, DSP and FF counts are compared. This comparison can happen between works using different datasets for testing, as given in Table II. The direct comparison must happen with [1] since this design is mostly following the hardware architecture used there. Comparing only the LUT counts, our design has much lesser value. This is because [1] has multiple LK cores in their architecture while this design implements only one core, to satisfy the objective of reducing resources.

TABLE III

Comparison of the LUT Count Between Various Works Implementing Pyramidal LK

Works	LUT Count
[1] Real-time and efficient optical flow tracking accelerator	66440
[17] Reduced memory access v-6	31305
[17] Reduced memory access v-4	43228
[4] Ultra-high performance and scalable hardware accelerator	57960
[8] RAFT hardware accelerator	129791
[18] VLSI Architecture for Lucas Kanade	11086 (Uses External memory)
Shi-Tomasi and Lucas Kanade (our work)	13838
YOLO person tracking and Lucas Kanade (our work)	13838

TABLE IV

Comparison of AEE Between Various Works Implementing Optical Flow Algorithm

Works	Average End Point Error (AEE) in pixels
[8] RAFT hardware accelerator	3.75
[7] Optical Flow for video under Dynamic illumination	0.371

Shi-Tomasi and Lucas Kanade (our work)	2.5852
YOLO person tracking and Lucas Kanade (our work)	13.404924

7. CONCLUSIONS AND FUTURE DIRECTION

Our work has proven better in terms of resource utilisation and latency compared to most of the existing works. The project's focus has been to achieve low latency for usage in resource constrained environments, which has been achieved, but with a trade-off with accuracy. The AEE is comparable when using Shi-Tomasi but using YOLOv4 for feature extraction has given worse results.

This work can be used for object tracking application which do not require high accuracy but have resource constraints. The design can be optimized to maintain the resources utilised but decrease the AEE. This work does not include multi-feature pipelining, which can be introduced for decreased latency. Gaussian filtering can be done for every layer of the image pyramid to increase accuracy.

REFERENCES

[1] Y. Gong et al., "A Real-Time and Efficient Optical Flow Tracking Accelerator on FPGA Platform," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 70, no. 12, pp. 4914-4927, Dec. 2023

[2] Y. Feng, R. Zhang, J. Du, Q. Chen and R. Fan, "Freespace Optical Flow Modeling for Automated Driving," in IEEE/ASME Transactions on Mechatronics, vol. 29, no. 2, pp. 1511-152

[3] D. C. Stumpp, H. Akolkar, A. D. George and R. B. Benosman, "hARMS: A Hardware Acceleration Architecture for Real-Time Event-Based Optical Flow," in IEEE Access, vol. 10, pp. 58181-58198, 2022

[4] Y. Liu et al., "An FPGA-based Ultra-High Performance and Scalable Optical Flow Hardware Accelerator for Autonomous Driving," IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1-5, 2024

[5] A. Elasri, L. Cherroun, M. Nadour and A. Elaoua, "Mobile Robot Visual Navigation Systems using Optical Flow," 2nd International Conference on Electronics, Energy and Measurement (IC2EM), 2023, pp. 1-6, 2023

[6] A. S. Editya, T. Ahmad and H. Studiawan, "Direction Estimation of Drone Collision Using Optical Flow for Forensic Investigation," 10th International Symposium on Digital Forensics and Security (ISDFS), pp. 1-6, 2022.

[7] Yun Chen, HuiDuan, Yuanxin Song, Zemin Cai and Guangguang Yang, “Optical Flow Computation for Video Under the Dynamic Illumination”, IEEE Transactions On Multimedia, VOL. 25, pp. 6285-6300, 2023

[8] Y. Li, Y. Gao, Z. Su, S. Chen and L. Liu, "FPGA Accelerated Real-time Recurrent All-Pairs Field Transforms for Optical Flow," China Automation Congress (CAC), pp. 4799-4804, 2022

[9] S. Vikruthi, P. Ramalingamma, G. Subbarao, K. Hari, S. Naveena and M. Gnana Vardhan, "Tracking and Detection of Moving Vehicles using Hybrid Optical Flow Technique in Complex Transport Atmosphere," 2024 International Conference on Expert Clouds and Applications (ICOECA), pp. 500-504, 2024

[10] J. Mei, T. Zuo and D. Song, "Highly Dynamic Visual SLAM Dense Map Construction Based on Indoor Environments," in IEEE Access, vol. 12, pp. 38717-38731, 2024

[11] Al-Qudah, S.; Yang, M. Large Displacement Detection Using Improved Lucas–Kanade Optical Flow. Sensors 2023, 23, 3152

[12] L. Zhong, L. Meng, W. Hou and L. Huang, "An Improved Visual Odometer Based on Lucas-Kanade Optical Flow and ORB Feature," in IEEE Access, vol. 11, pp. 4717947186, 2023

[13] Alfarano, Andrea, et al. "Estimating optical flow: A comprehensive review of the state of the art." Computer Vision and Image Understanding (2024): 104160.

[14] S. J, S. S. M. Vadhoolas, V. K. K V, V. R. H K and K. N.Rajanikanth, "Visual Kinetic SLAM Hardware Acceleration for Navigation," 2023 International Conference on Network, Multimedia and Information Technology (NMITCON), Bengaluru, India, 2023, pp. 1-8

[15] Bouguet, Jean-Yves. "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm." Intel corporation 5.1-10 (2001): 4.

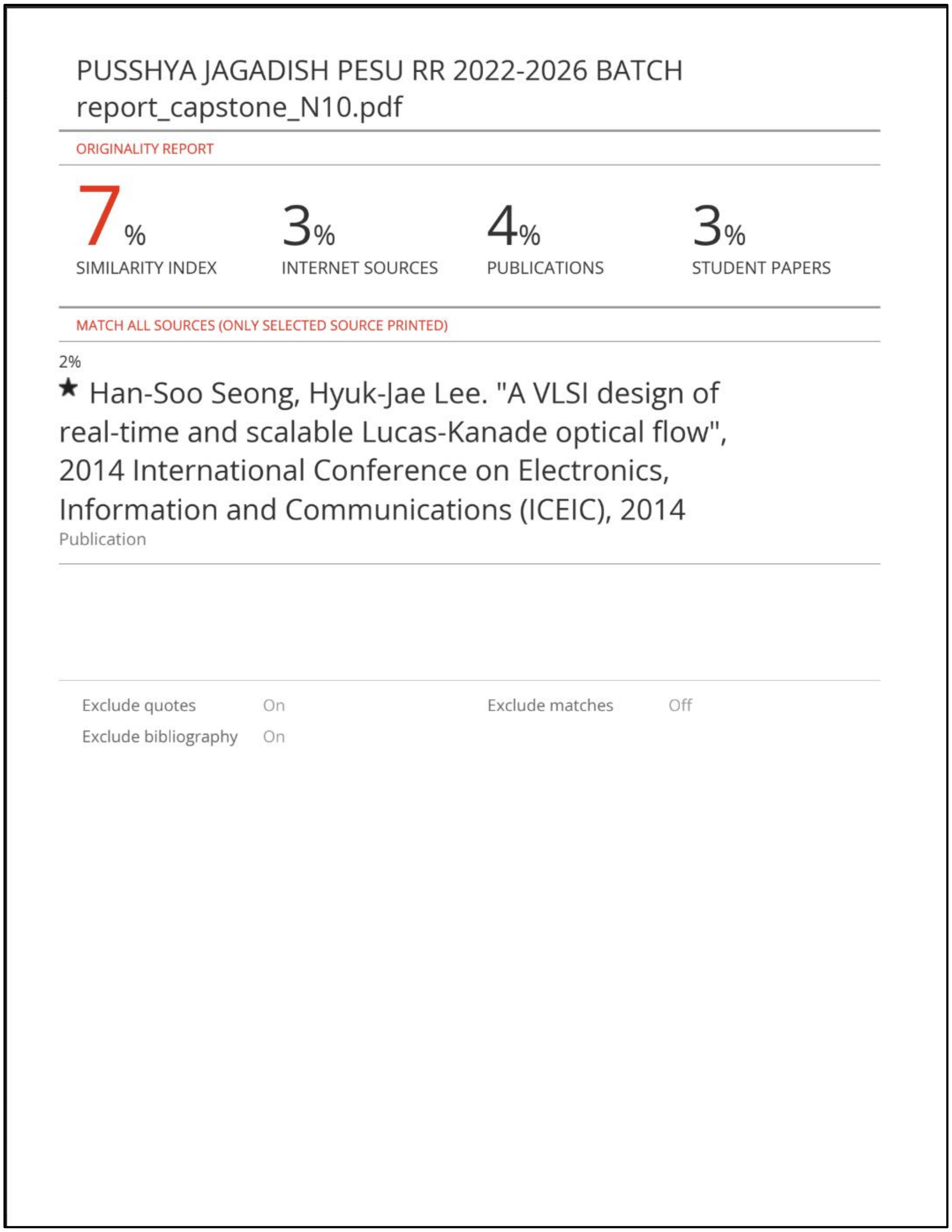
[16] Zhengguang Chen et al 2025 Meas. Sci. Technol. 36 035201 DOI 10.1088/13616501/adad91

[17] H. -S. Seong, C. E. Rhee and H. -J. Lee, "A Novel Hardware Architecture of the Lucas–Kanade Optical Flow for Reduced Frame Memory Access," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 26, no. 6, pp. 1187-1199, June 2016, doi: 10.1109/TCSVT.2015.2437077

[18] V. Mahalingam, K. Bhattacharya, N. Ranganathan, H. Chakravarthula, R. R. Murphy and K. S. Pratt, "A VLSI Architecture and Algorithm for Lucas–Kanade-Based Optical

Flow Computation," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 1, pp. 29-38, Jan. 2010, doi: 10.1109/TVLSI.2008.2006900

PLAGIARISM REPORT



PUSSHYA JAGADISH PESU RR 2022-2026 BATCH
report_capstone_N10.pdf

ORIGINALITY REPORT

7%	3%	4%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

MATCH ALL SOURCES (ONLY SELECTED SOURCE PRINTED)

2%

★ Han-Soo Seong, Hyuk-Jae Lee. "A VLSI design of real-time and scalable Lucas-Kanade optical flow", 2014 International Conference on Electronics, Information and Communications (ICEIC), 2014
Publication

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		

POSTER

PES UNIVERSITY

FPGA- IMPLEMENTATION OF PYRAMIDAL LUCAS-KANADE OPTICAL FLOW FOR REAL-TIME SYSTEMS

Pusshya Jagadish, Pragati Ramesh, Pradyun Kamath
Project Guide: Prof. B Rajeshwari

MOTIVATION

- Real-time systems demand low latency and high throughput.
- Rapidly increasing data volume for complex operations require hardware acceleration.
- Pyramidal Lucas-Kanade algorithm on FPGA balances latency, accuracy and resoruces effeciently.

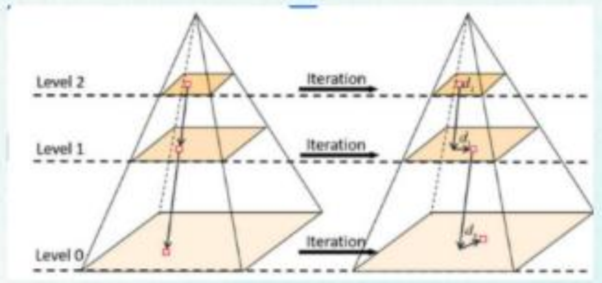


Figure 1:Optical flow calculation scheme in the multi-scale method

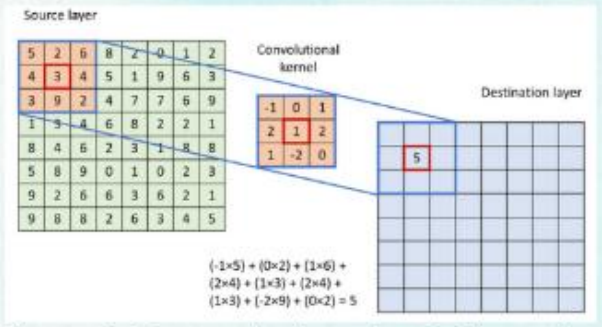


Figure 2: 2D convolution using 3x3 kernel

IMPLEMENTATION

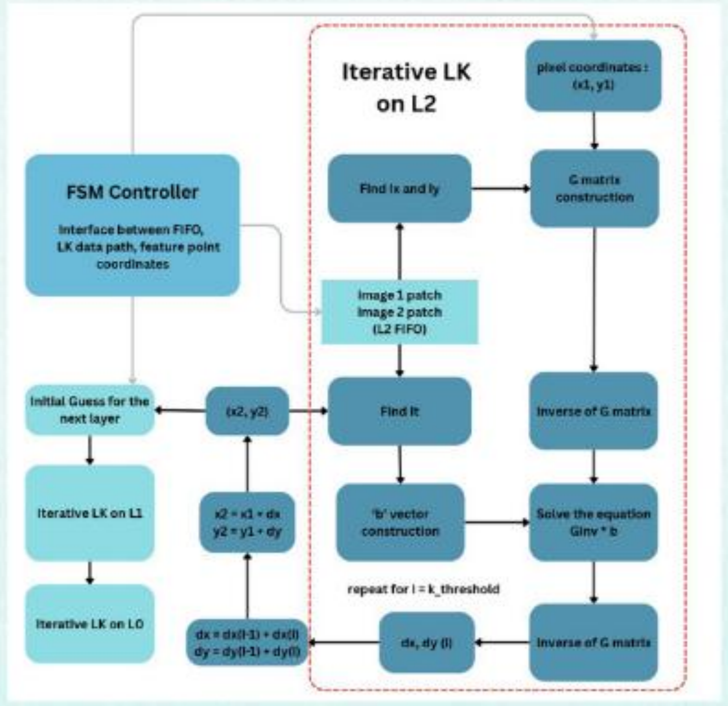


Figure 3: System Integration

- Pre-processing and visualization tasks performed in MATLAB
- Fixed-point arithmetic used
- Systolic array architectures used for 2D convolution for high throughput
- Pseudo image pyramid constructed to reduce memory and latency
- FIFO used to store each pyramid layer

RESULTS AND CONCLUSION

Shi-Tomasi Feature Extraction with Pyramidal LK	
Latency (ms/transition)	1.5
LUT	13838
AEE (in pixels)	2.5852

- The results are for 1 frame of 436x1024 resolution
- 33x33 patch of pixels considered around the given pixel coordinate
- The dataset used for evaluation is MPI Sintel
- The latency is comparable to the state-of-the-art works in the domain
- The LUT count is measured for the design without considering the layer FIFOs
- The low latency and resource utilisation is a trade-off with the accuracy (AEE)

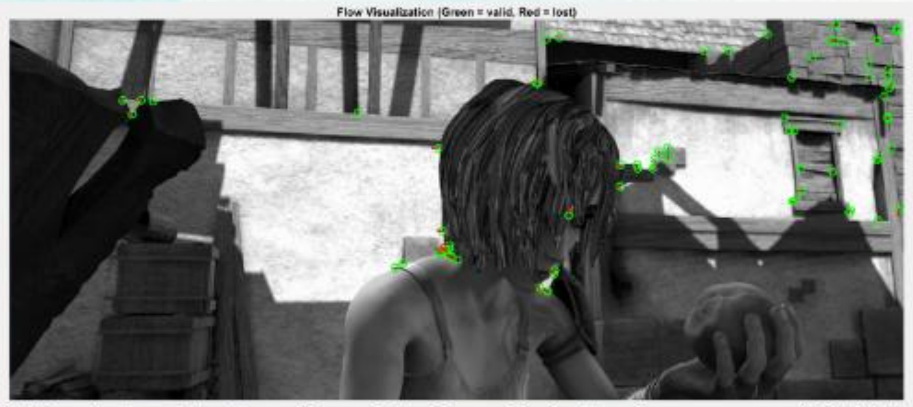


Fig.3. Tracked points from frame1 to frame2 of alley_1 sequence, MPI Sintel