YODA Final Report

Prepared by:
Group 10
Dimpho Sefora SFRREN001
Griffin Trace TRCGRI001
Kavya Kaushik KSHKAV001
Zuhayr Halday HLDZUH001

Prepared for:
EEE4120F
Department of Electrical Engineering
University of Cape Town

May 26, 2025

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

May 26, 2025

Griffin Trace — Date

May 26, 2025

Dimpho Sefora — Date

May 26, 2025

Kavya Kaushik — Date

May 26, 2025

Zuhayr Halday

Date

Abstract

This project presents the design and implementation of a high-performance image processing pipeline optimized for FPGA hardware. The system targets the removal of impulse noise and enhancement of edge features in digital images using a combination of median filtering and Sobel edge detection. A key focus was ensuring the solution is both accurate and suitable for real-time embedded applications. The pipeline begins with a 3×3 median filter that suppresses salt-and-pepper noise on a grey scale input image. Edge detection is performed using the Sobel operator, which identifies sharp intensity transitions in horizontal and vertical directions. Both filtering and edge detection stages were developed in MATLAB as golden reference models, then translated into Verilog HDL and simulated for a Spartan A7 FPGA using Vivado. The FPGA implementation achieved over 189× speed-up compared to its software counterpart, completing each image in approximately 2.652 ms, with minimal loss in image quality. The results were validated across a variety of test images ranging from simple scenes to detailed textures. The design demonstrates the effectiveness of FPGA-based acceleration for pixel-level operations, highlighting benefits such as parallelism, low latency, and resource efficiency. Future extensions may include adaptive thresholding, improved border handling, and integration of lightweight machine learning models to further refine noise suppression and edge extraction. This project establishes a scalable foundation for real-time vision systems in robotics, surveillance, and embedded AI applications.

Contents

Chapter 1

Introduction

The Image Edge Detection Processing project is centred around the development of a high-performance digital image processing pipeline. This system is designed to transform raw, noisy images into clean, analyzable forms while retaining critical details such as edges. Using a combination of noise filtering and edge detection techniques, the system enables efficient and accurate processing on hardware platforms like FPGAs.

The key focus of the project is to convert input images to grayscale, filter out impulse noise, and apply edge detection without sacrificing important image features. Unlike general-purpose image enhancement systems, this project specifically targets pre-processing steps that prepare images for further analysis, such as object detection or segmentation. The design emphasises accuracy, resource efficiency, and suitability for real-time embedded applications.

The code base for the project can be accessed on its GitHub repository.

1.1 Background

Image processing is a key component in applications ranging from medical imaging to industrial inspection and autonomous robotics. In most practical settings, images captured by sensors are contaminated by noise due to limitations in hardware, environmental interference, or transmission errors. One of the most disruptive forms of this is impulse noise (salt-and-pepper noise), which appears as sporadic white and black pixels that distort image content and must be removed to restore image integrity before further processing [1, 2].

To address this, median filtering has become a widely adopted nonlinear method to remove noise. Unlike linear filters (e.g., mean filters), which tend to blur images and degrade edge definition, median filters replace each pixel with the median of its neighbours. This helps suppress noise without introducing new pixel values, thereby maintaining edge sharpness. Median filtering is particularly effective for removing positive and negative impulse noise in grayscale images [1].

However, the classical median filter suffers from high computational complexity due to its reliance on sorting its neighbouring pixels. This makes it unsuitable for real-time systems without hardware acceleration. To overcome this, various FPGA-optimised median filtering architectures have been developed. For example, Phukan proposes a three-stage algorithm using partial column-wise and row-wise sorting, enabling the median of a 3×3 window to be determined using only three final comparisons rather than full sorting, reducing processing time from 9 to 3 clock cycles per pixel [1]. Vega-Rodríguez et al. further reduce hardware resource usage through a minimum exchange sorting

network with pipelining and reuse of intermediate results, achieving a performance speedup of up to 85× over software implementations [2].

Following denoising, edge detection is the next essential step in image processing. Edges denote areas of sharp gradient change and are fundamental for feature extraction, object recognition, and boundary segmentation. Among many algorithms, including Canny, Prewitt, and Laplacian operators, the Sobel operator is often preferred in hardware applications for its balance between accuracy and computational efficiency. The Sobel method computes gradients in horizontal and vertical directions using simple convolution masks, making it suitable for real-time FPGA [3].

Nevertheless, basic Sobel filtering can result in broken edges and thick contours. To address these limitations, extended versions incorporating four or even eight directional masks have been proposed. Ravivarma et al. implement Sobel masks in eight orientations (N, S, E, W, and the diagonals) and combine them using magnitude calculations and thresholding, producing clearer and more complete edge maps. Their design also leverages Simulink subsystems to generate HDL-compatible code for Xilinx FPGA platforms [3].

In summary, while median filtering and Sobel edge detection are both well-established, their FPGA-based implementation—especially when optimised for low-latency, high-throughput operation—remains an active area of engineering research. This project builds on such advances to create a robust image processing pipeline capable of real-time denoising and edge enhancement using efficient Verilog designs.

1.2 Objectives

This project aims to implement an efficient, real-time image processing pipeline that removes impulse noise and enhances edge features using FPGA-based acceleration. The objectives are broken down as follows:

1. **Impulse Noise Removal Using Median Filtering**
    - Implement a 3×3 median filter to suppress salt-and-pepper noise across each color channel individually. This approach preserves colour fidelity and reduces random noise artefacts.
    - The method follows Phukan's FPGA-optimised design, which reduces full sorting complexity through a staged column-row sorting process [**?**].
2. **Edge Detection Using Sobel Operator**
    - The images need to be converted to greyscale for efficient edge detection. The scope of this project is limited to grey scale input therefore, this conversion step is not implemented.
    - Apply bi-directional Sobel operators (horizontal and vertical) on the grayscale image to detect edges with enhanced orientation awareness.
    - This method is chosen for its FPGA-friendly integer arithmetic and implementation simplicity [**?**].
    - Combine directional gradients using the L2 norm and apply thresholding to produce a

binary edge map suitable for real-time digital acceleration.

3. **Develop Golden Reference Models in MATLAB**
   - Create high-level MATLAB implementations of the median filter and Sobel edge detector.
   - Use these models as golden references to validate correctness and image quality before simulating hardware translation.
4. **Translate Functional Models to Verilog for FPGA Deployment**
   - Generate Verilog code for the median filter and Sobel edge detector using Vivado.
   - Synthesise and implement these modules on a Spartan 7 FPGA, emphasising low latency, high throughput, and efficient logic utilisation.
5. **Compare FPGA Performance Against Golden Reference Baseline**
   - Benchmark FPGA performance against this CPU based Golden Reference approach to evaluate trade-offs in speed, power efficiency, and scalability.

Each objective contributes to a modular, testable pipeline that meets the real-time constraints of embedded vision systems while maintaining high image quality under noisy conditions.

1.3 Scope and Limitations

This project focuses on designing and implementing a digital image processing pipeline capable of performing impulse noise removal and edge detection using FPGA acceleration. The scope is limited to processing static grayscale images using a fixed-size resolution. The pipeline includes median filtering and Sobel edge detection. All algorithms are designed to be hardware-synthesizable and optimised for real-time performance on an FPGA platform. MATLAB models are developed for initial validation, followed by hardware-based simulation using Verilog implemented in Vivado.

The limitations of this project include the exclusion of dynamic video input, or machine learning-based approaches to edge refinement. Timing and resource constraints inherent to mid-range FPGAs limit the complexity and depth of operations. Additionally, the project timeline constrained opportunities for full system integration with the actual hardware implementation on the FPGA board.

Chapter 2

Methodology and Planning

2.1 Tools

The following tools were employed throughout the project:

- **MATLAB** — Used to develop the golden measure reference, including both the median filter and Sobel edge detection algorithms.
- **Vivado Design Suite** — Utilised for FPGA development, encompassing simulation, implementation, and bitstream generation.
- **Verilog HDL** — Employed to implement the median and Sobel modules for FPGA execution.

2.2 Methodology

2.2.1 Golden Measure Implementation

Before FPGA implementation, a golden measure was developed to provide a known baseline performance for image processing in software. The golden measure processes input images and produces both visual outputs and timing metrics in MATLAB. Its two main stages are detailed below.

Median Filtering

Input: RGB image

Steps:

1. Convert the image to double precision.
2. Pad the image to accommodate windowing at borders.
3. For each pixel:
    - Extract an 3×3 window centered on the pixel.
    - Compute the median of each colour channel in the window.
    - Replace the original pixel with the computed median.

Output: Median-filtered RGB image.

Sobel Edge Detection

Input: Median-filtered RGB image

Steps:

1. Convert the image to grayscale.
2. Apply 3×3 Sobel operators to compute horizontal and vertical gradients.
3. Calculate the gradient magnitude.
4. Normalise and threshold the result to produce a binary edge map.

Output: Binary edge map highlighting prominent edges.

Benchmarking Metrics

The MATLAB implementation records the following metrics for each test image:

- **Median Filtering Time**: Duration taken to apply the median filter.
- **Sobel Edge Detection Time**: Duration taken to perform Sobel edge detection.
- **Total Processing Time**: Sum of median filtering and Sobel edge detection times.
- **Edge Density**: Ratio of edge pixels to total pixels in the binary edge map.

2.2.2 FPGA Design Methodology

The FPGA design process began with translating the MATLAB algorithm into Verilog. The median filter was implemented using line buffers and sorting logic to compute the median in a streaming fashion. The design was pipelined for improved performance and constrained to fit within typical FPGA resource limits.

Once the initial Verilog modules were developed, they were simulated using Vivado's built-in simulation tools. Functional verification was performed by feeding test images into the simulated hardware model, and the output was compared to the MATLAB golden measure. Since physical deployment was not within the project scope, synthesis and implementation were performed purely for analysis purposes. Timing and resource utilisation reports were generated to assess the feasibility of the design on actual hardware.

Design choices—including pipelining, fixed-point representation, and buffering strategies—were guided by practical hardware considerations. Iterative improvements were made throughout the simulation process to optimise resource usage and ensure the design would meet timing requirements in a real-world FPGA implementation.

2.2.3 Test Procedures

Testing compared both implementations using grayscale images with varying noise and detail levels. An outline of all experimental testing is shown below:

Procedure:

1. Run the MATLAB implementation and collect outputs and metrics.
2. Run the FPGA implementation on the same images.
3. Compare results visually and quantitatively.

Evaluation Metrics:

- Noise filtering, edge density and visual clarity of detected edges.
- Median filtering and Sobel edge detection times.

2.3 Planned Experimentation

2.3.1 Experiment 2.1: Median Filter Evaluation

Objective: Evaluate the noise reduction effectiveness of the median filter.

Procedure:

1. Apply the median filter to noisy images using both MATLAB and FPGA.
2. Visually inspect filtered images and compare.
3. Record and compare processing times.

Metrics:

- Quality of noise suppression (qualitative).
- Median filter processing time (quantitative).

2.3.2 Experiment 2.2: Sobel Edge Detection Evaluation

Objective: Assess the accuracy and speed of the Sobel module.

Procedure:

1. Apply edge detection to previously filtered images.
2. Compare binary edge maps for accuracy.
3. Measure and compare processing times.

Metrics:

- Edge density and clarity of output. (qualitative)
- Sobel module processing time. (quantitative)

2.3.3 Experiment 2.3: End-to-End Pipeline Evaluation

Objective: Benchmark the entire image processing pipeline.

Procedure:

1. Process test images through both full pipelines (MATLAB and FPGA).
2. Evaluate image quality and edge detection results.
3. Compare total processing times.

Metrics:

- Overall output quality.
- Total end-to-end processing time.

2.4 Proposed Development Strategy

To transition this project from a proof-of-concept to a viable product, the following four-phase development timeline is proposed:

- **Phase 1 — Prototype (Weeks 1–3):** Implement basic median and Sobel filters on an FPGA. Validate functionality using static test images and compare against MATLAB benchmarks.
- **Phase 2 — Optimisation (Weeks 4–6):** Improve latency and resource efficiency. Integrate modules into a unified pipeline and enable real-time image streaming.
- **Phase 3 — System Integration (Weeks 7–9):** Develop host-side control software and user interface. Add flexible configuration options for filter parameters and input/output handling.
- **Phase 4 — Deployment Preparation (Weeks 10–12):** Package the design into a standalone module with documentation. Explore deployment options, including custom PCB design or integration into existing imaging systems.

This roadmap outlines a structured approach to refining the accelerator and preparing it for potential commercialisation or broader application.

Chapter 3

Design

3.1 FPGA Verilog Implementation

3.1.1 System Architecture

The FPGA implementation of the hardware accelerator, for median filtering and Sobel filtering, was broken down into a number of separate modules. These separate modules each have their own responsibilities and tasks to perform, and when connected together perform the task at hand. These main modules are:

- Tesh Bench Module
- Filter Top Module
- Controller Module
- Median Filter Module
- Sobel Filter Module

These will be discussed in more detail below:

3.1.2 Test Bench Module

This model is used for testing the filtering pipeline in simulation. It is responsible for reading the data from a file, providing the data to the filtering and data handling models, and writing the filtered data back to a file.

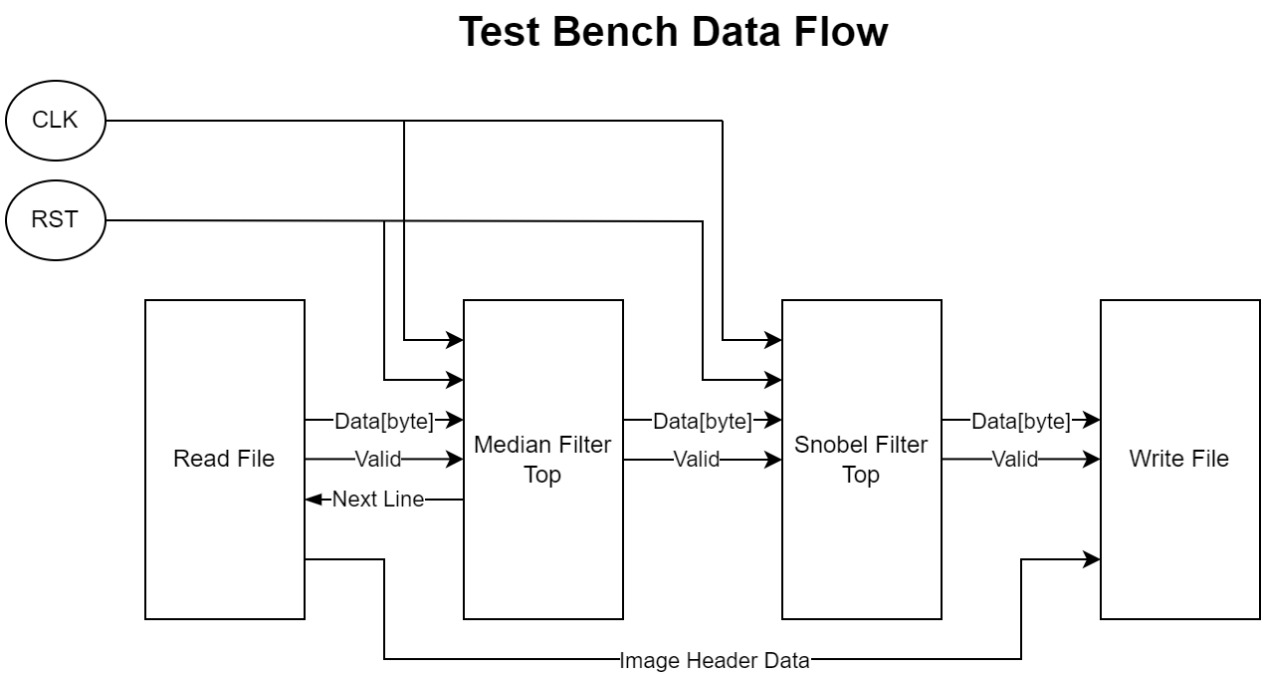


Figure 3.1: Test Bench Data Flow Diagram

Read File: Data Ingest

In the test bench implementation, the image data is read one byte at a time from a .bmp file. This is illustrated by the 'Read File' block in Figure 3.1. Each byte represents the brightness of a pixel with a value of 0 to 255. For this implementation, it is assumed the image has a resolution of 512x512, and so 512 bytes are needed to be sent for one line of the image. When the downstream module is ready for a line of the image, it will issue a 'Next Line' request. In response, the 'Read File' block will start sending 512 bytes one at a time, and will assert a 'Valid' when each byte is ready to be read.

The .bmp file begins with a header containing metadata for the image, such as resolution, colour format, and other descriptors. Modifying this header lies outside the scope of this project, and so it is passed directly through to the output.

In a real deployment on an FPGA, as opposed to the simulation, the 'Read File' block would be replaced with a module capable of reading the image data from an external source, such as an image stream over UART or another serial communication interface.

Filter Top: Image Filters

Each 512-byte stream, representing one line of the image, is fed into a Filter Top module. This module processes the raw image data stream and outputsts a corresponding filtered data stream. The internal structure and workings of this module will be discussed in more detail in subsection 3.1.3. This module is largely identical between the Median and Sobel filter implementations, and only differs by which filtering submodule it makes use of. As shown in Figure 3.1 two of these filter top modules are connected in sequence, the first performing a median filter, and the following performing the Sobel filter.

This module reads a stream of bytes from its input, being read when a 'Valid' input signal is sent. When this module is ready for more data, it issues a 'Next Line' request. Once the filtering process

has completed, this module outputs a filtered data stream, accompanied by a 'Valid' signal when each filtered byte is ready to be read.

Write File: Data Output

Similar to the data ingest process, the 'Write File' block, in Figure 3.1, handles the filtered image data by writing it back to a .bmp output file. This allows the filtered output to be viewed and analysed.
As with the data ingest process, this is only applicable to the simulation environment. In a real deployment on an FPGA the 'Write File' block would be replaced with a module capable of sending the image data to an external destination, such as sending a data stream over UART, or another serial communication interface.

3.1.3 Filter Top Module

As described in subsection 3.1.2, this module receives a data stream of 512 bytes at a time, each representing one line of the image. The module then performs the necessary filtering process and outputs a corresponding filtered data stream.

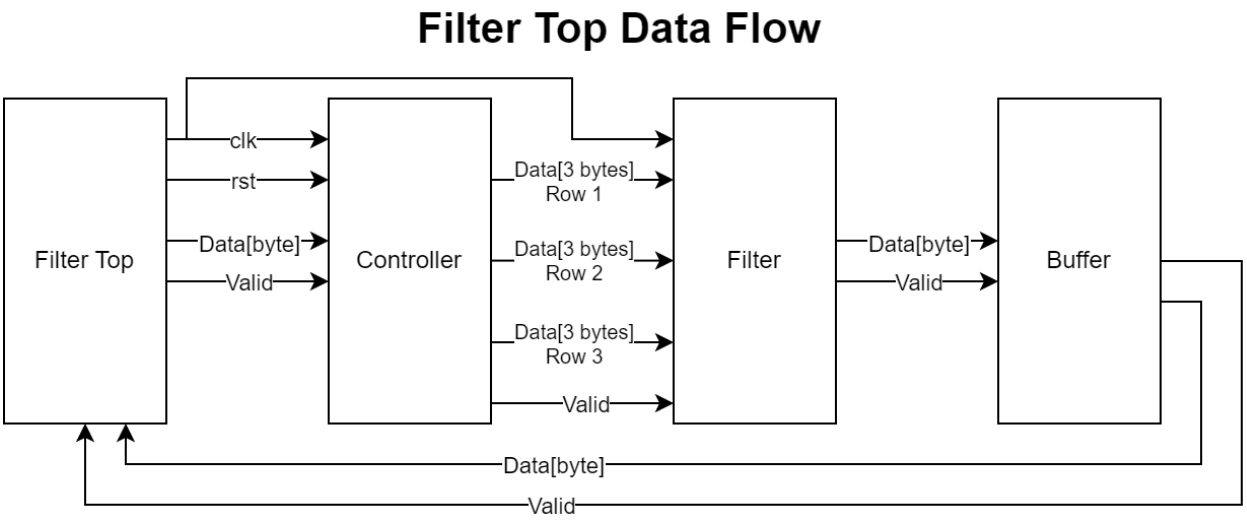


Figure 3.2: Filter Top Module Data Flow Diagram

Controller: Data Handling and Preparation

The Controller module is responsible for preparing the unfiltered image data for processing by the filter module. It receives the incoming data stream one byte at a time, accepting each byte when a 'Valid' signal is asserted. The Controller temporarily buffers several lines of image data in order to generate a 3×3 window of pixel values—this sliding window is essential for both median filtering and convolution-based edge detection.
Once a valid 3×3 window is available, the Controller outputs nine bytes corresponding to the 3×3 grid of pixels, along with a 'Valid' signal indicating that the data is ready for filtering. Further details on the data buffering and preparation are provided in subsection 3.1.3.

Filter: Sobel or Medain Filter

This module is responsible for executing the core filtering operation. It receives a 3×3 window of pixel data—nine pixels in total—and applies either a Sobel filter or a Median filter, depending on the configuration. The module processes the incoming data once a 'Valid' signal is received, and produces a single filtered output pixel. A corresponding 'Valid' signal is asserted when the output data is ready.
The specific filtering algorithms and their implementations are discussed in more detail in subsection 3.1.5 and subsection 3.1.6.

Buffer

After the filtering process is complete, the resulting data is temporarily stored in a FIFO buffer. This buffer is included to handle scenarios where the downstream module, such as 'Write File' discussed in subsection 3.1.2, is not immediately ready to accept new data, preventing data loss and ensuring smooth pipeline operation. The FIFO is implemented using an IP core.

The FIFO module includes a half-full flag that can be used in a real hardware deployment to signal the upstream module to pause data input. Additionally, there is a flag forwarded to the FIFO to request the next byte from the buffer. In the simulation environment, the output is always ready, rendering this backpressure management unnecessary. Since this feature is not required in simulation, it is not actively used and therefore not shown in the data flow diagrams.

3.1.4 Controller Module

The Controller module is responsible for managing the input stream of raw pixel data, buffering it, and preparing it for the following filtering operation. This module ensures that pixels are correctly aligned and grouped for processing.

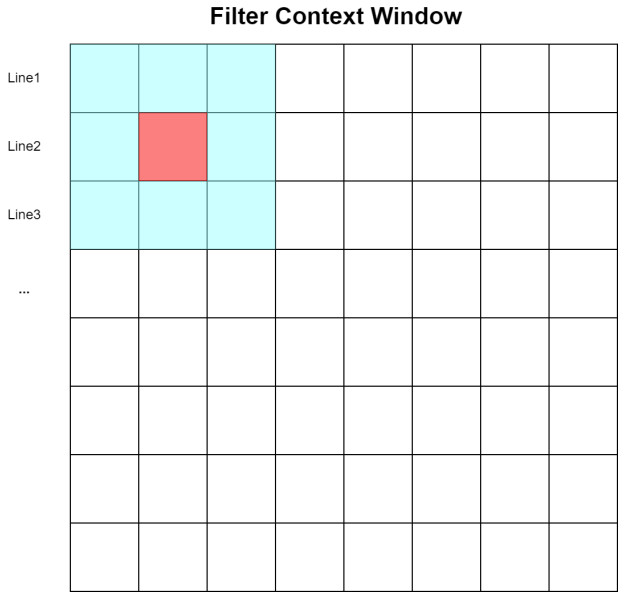


Figure 3.3: Context Window

As shown in Figure 3.3, both the Median and Sobel filters operate on a 3×3 grid of pixel values, referred to as a context window. Each 3×3 window contains 9 pixels from the image, and these are used to compute a single filtered output pixel. In the diagram, the context window is shown in blue, for the given pixel, shown in red.

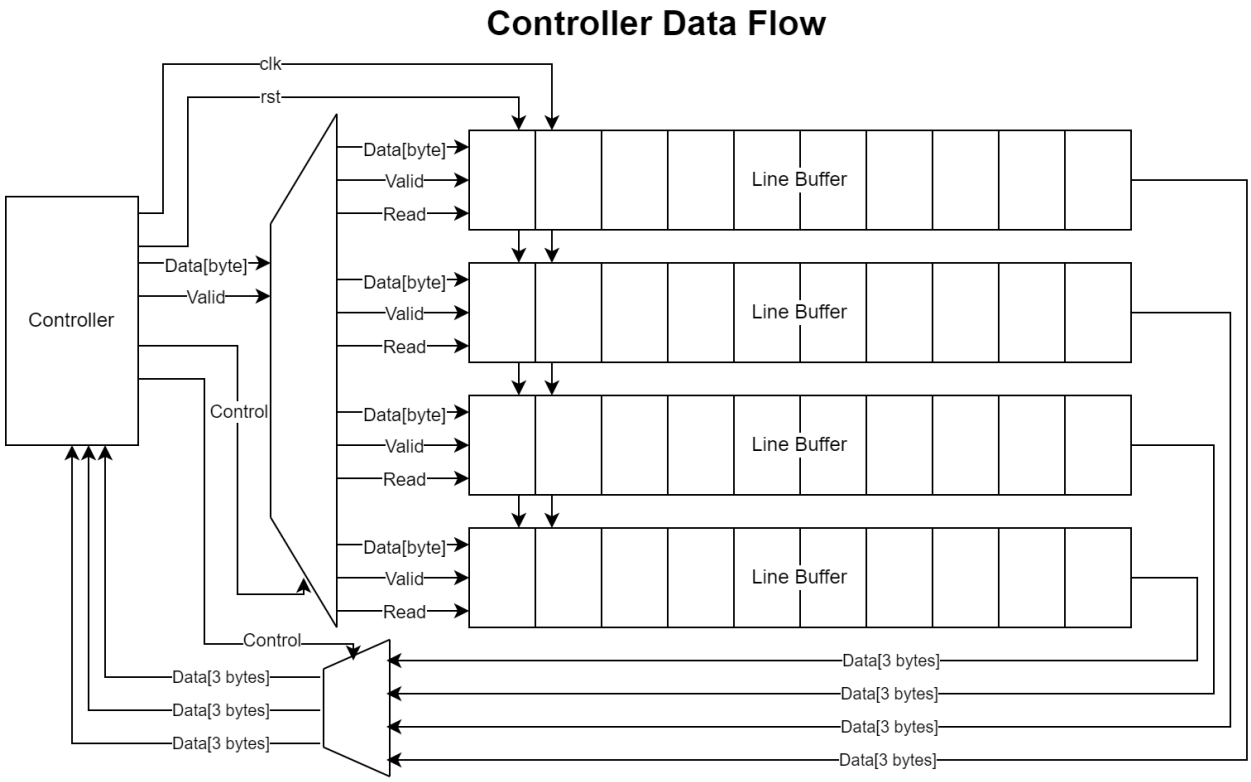


Figure 3.4: Controller Data Flow Diagram

Figure 3.4 illustrates the internal architecture and data flow of the Controller module.

To construct the required 3×3 context window, the controller must buffer at least three full lines of pixel data. It receives one pixel at a time, and upon the assertion of a 'Valid' signal, each byte is stored sequentially into a set of line buffers. Once three line buffers are sufficiently filled, the controller begins generating the 3×3 context windows of pixels. This context window is then moved down the image, from left to right. For each context window, it outputs 9 pixel values to the downstream filtering module, along with a 'Valid' signal to indicate that the data is ready for processing.

Internally, the controller is implemented using four line buffers, each capable of storing one full row of the image. Each buffer receives input when a valid pixel is available and outputs three horizontally adjacent pixel values at a time. The three output pixels are moved along the buffer when a 'Read' signal is asserted, which allows the context window to move across the image from left to right.

Although theoretically only requiring three line buffers to make the 3x3 context window, the controller

makes use of a fourth buffer to increase efficiency and reduce downtime. With only three buffers, the system would be forced to pause after reaching the end of a line, while the next buffer fills up with new data, before resuming filtering. By using four buffers, this can be mitigated. While three buffers are being output, the 3x3 context window, the fourth can be filled with the incoming data for the next line of the image. When the end of a line is reached, the controller can simply shift the context window down, including the new line buffer in its output and removing the oldest. The oldest line buffer is now also ready to be filled with new data.

To perform this, the controller includes some basic logic to manage buffer selection, ensuring that the correct three buffers are used for output, while the fourth is assigned for input. This cyclic buffer strategy maximises throughput and allows the filtering pipeline to operate efficiently without interruption or delay.

3.1.5 Median Filter Module

As explained in subsection 3.1.4 the filter module receives 9 pixel values, each as a byte, and a 'Valid' signal when the pixel values are ready to be used. The filter module performs the median filter operation and outputs a single pixel value with a corresponding 'Valid' signal.

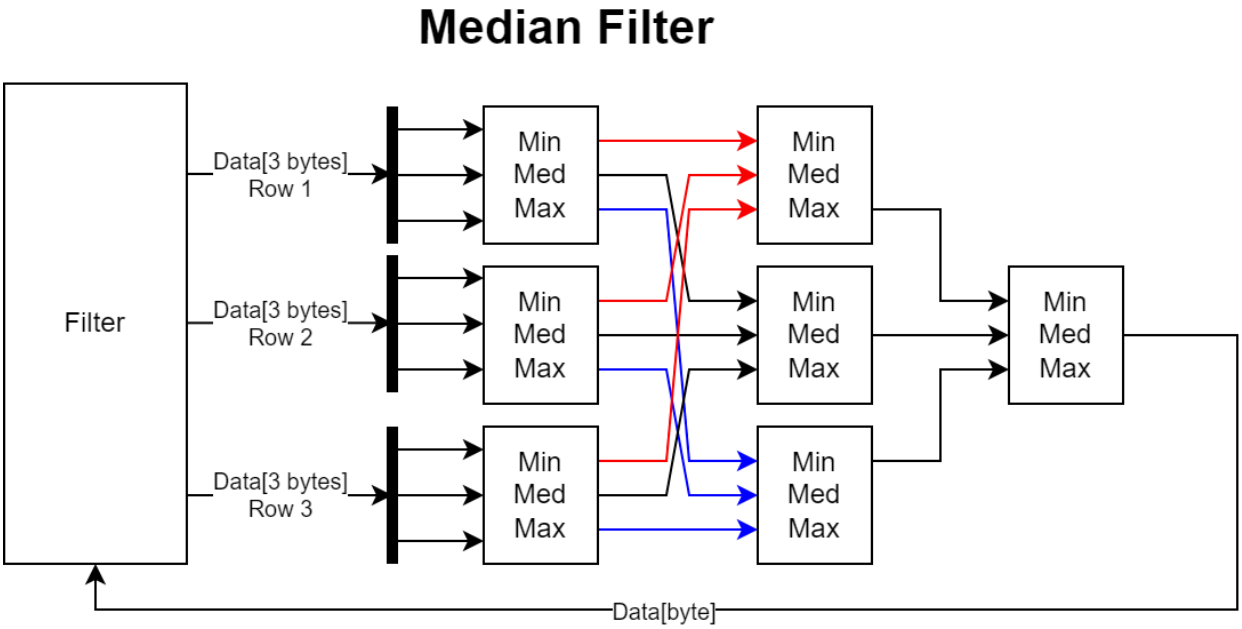


Figure 3.5: Median Filter Data Flow

The median filter makes use of a set of custom Min-Med-Max modules, each designed to process three input values. These modules perform simple comparison operations to determine and output the minimum, maximum, and median values of the inputs.
The median filter implementation follows the FPGA-based architecture proposed by G.Gupta [4], which efficiently computes the median of nine input pixels using a three-layer structure:

- Layer 1 computes the minimum, median, and maximum of each of the three groups of three

input pixels.

- Layer 2 takes the outputs of Layer 1 and computes the maximum of the minimums, the median of the medians, and the minimum of the maximums.
- Layer 3 then computes the median of these three intermediate values, which corresponds to the true median of all nine original inputs.

This process is illisatred in Figure 3.5

As detailed by G.Gupta[4], this architecture requires only three clock cycles to compute the median when implemented on an FPGA, making it exceptionally efficient for real-time image processing applications.
The final median value is then output by the controller back to the upstream modules, to be forwarded to the next filter or to the output.

3.1.6 Sobel Filter Module

The Sobel Filter Module is responsible for detecting edges in the image by applying two distinct 3×3 convolution kernels to the 9-pixel input window. These Sobel kernels are designed to highlight regions of high spatial frequency, which typically correspond to edges in the image. This module receives a 3×3 pixel window, along with a 'Valid' signal from the Controller module, and outputs a single convolved value with a corresponding 'Valid' signal once computation is complete. The two kernels used are defined as follows:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}, \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Each kernel is convolved with the 3×3 window to generate two intermediate results, corresponding to the horizontal and vertical gradients, respectively. These values are denoted as G_x and G_y. The magnitude of the gradient is then computed as: $G = \sqrt{G_x^2 + G_y^2}$. In the FPGA implementation, this square root operation is approximated by comparing the squared gradient magnitude $G_x^2 + G_y^2$ against a fixed threshold. This simplification avoids expensive square root computations and enables faster execution in hardware.

The module architecture is pipelined to maintain high throughput:

1. Stage 1 – Multiply: Each of the nine input pixels is multiplied with its corresponding kernel coefficient from both G_x and G_y. Signed multiplication is used to support negative kernel weights.
2. Stage 2 – Sum: The results of each set of nine multiplications are summed to produce G_x and G_y. Summation is performed using signed arithmetic to preserve edge directionality.
3. Stage 3 – Square: Each of the gradient components is squared to prepare for magnitude calculation.

4. Stage 4 – Combine: The squared components are summed to approximate G^2.
5. Stage 5 – Thresholding: The final value is compared against a fixed threshold (e.g., 4000). If the magnitude exceeds this threshold, the output pixel is set to 255 (white); otherwise, it is set to 0 (black), resulting in a binary edge map.

All pipeline stages are synchronized with the system clock, and 'Valid' signals are used to indicate when data is available or ready at each stage. Intermediate pipeline registers ensure that data is processed on every clock cycle, allowing for continuous operation with minimal latency.

The output of the Sobel filter is outputted back to the upstream modules, to be forwarded to the next filter or to the output.

3.2 Running filter system on a FPGA

As described in subsection 3.1.2, the current implementation is intended for simulation, specifically within the Vivado environment. However, the design has been structured in such a way that it can be easily adapted for direct deployment on an FPGA.

The primary modification required for real hardware implementation is the integration of a data interface to supply image data to and from the FPGA. This can be accomplished using a serial communication interface such as UART, which would replace the 'Read File' block referenced in subsection 3.1.2 and shown in Figure 3.1. Similarly, the 'Write File' block would need to be replaced with a corresponding serial stream to transmit the filtered image data back to the host system.

It is important to note that the FIFO buffer, described in subsection 3.1.3, is included in the design and plays a critical role in flow control. Specifically, the input data stream should only send new data when the FIFO's half-full flag is not asserted, this flag is forwarded through the filter top module. Additionally, the output can use a flag forwarded to the FIFO to request the next byte from the buffer. Together, this ensures that the FPGA does not become overwhelmed with input data if the output is not ready to accept more, and that data is not lost.

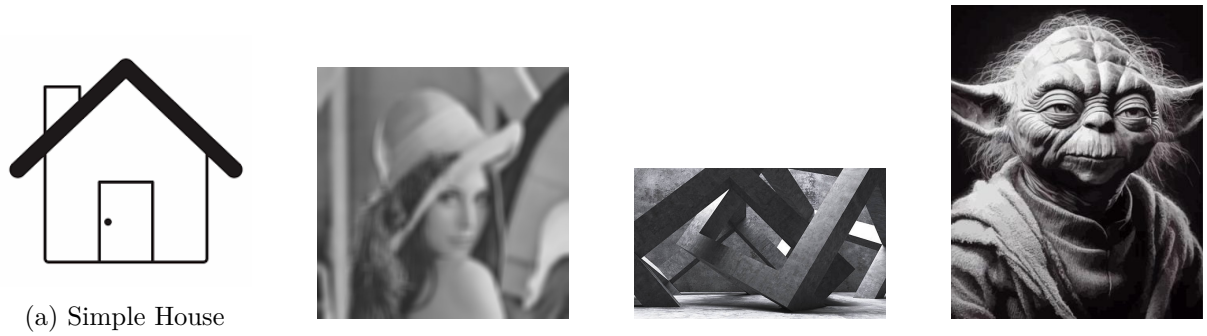
Chapter 4

Results

This section presents and analyses the results obtained from both the golden standard and FPGA implementations. Four distinct images are used to evaluate the performance of each approach in image processing.

For both implementations, the outputs are divided into two categories: the results of the median filtering and the results of the Sobel edge detection. The analysis focuses on how effectively each implementation reduces noise in three types of images: a simple image with clear edges, a blurry image, and an image rich in edges and fine details. Similarly, the Sobel edge detection results are examined to assess how accurately each method identifies edges in the same set of images. A bonus image will be used in the analysis: How well can the Yoda project process an image of YODA himself?

The images that will be used for this are as follows:



(a) Simple House Image (b) Blurry Image (c) Detailed Image (d) YODA

[h!]

Figure 4.1: Images to Test The Performance and Quality of Image Processing

4.1 Median Filter

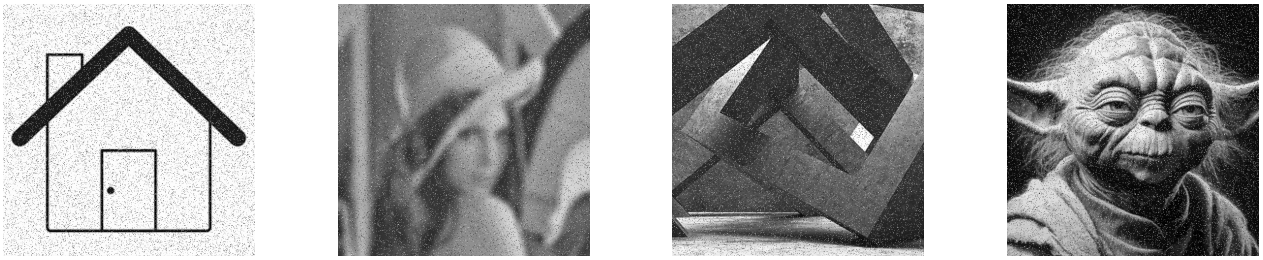
The focus here is on how well the filter removes noise from the different images. Homing in specifically on the quality and performance produced by each implementation.

4.1.1 Results of Median Filter

The images were pre-processed using GIMP, an open-source image editing software. As part of the processing pipeline, each image was resized to a standard resolution of 512×512 pixels to ensure consistency across the dataset. In addition, a type of visual distortion called Hurl noise was applied.

This noise introduces random pixel-level alterations to simulate more challenging image conditions. A randomisation level of 10% was used, which means that 10% of the pixels in each image were randomly modified to introduce this noise effect.

The formatted and noise-induced images are as follows:



(a) Noisy Simple House Image

(b) Noisy Blurry Image

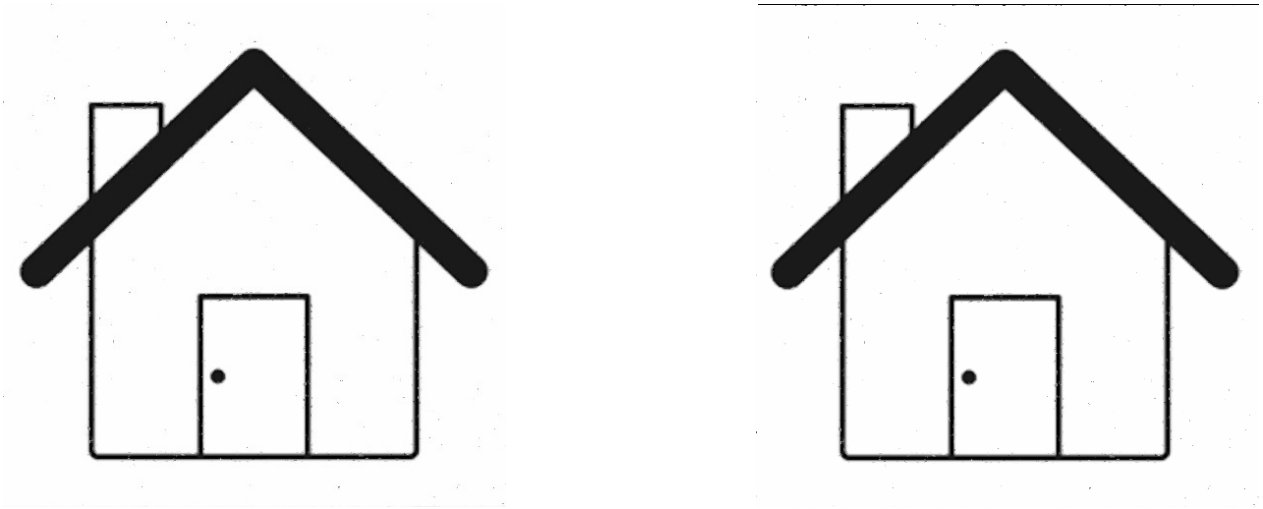
(c) Noisy Detailed Image

(d) Noisy YODA Image

Figure 4.2: Formatted and Noise-Induced Images

Simple House Image Results

The following images show a side-by-side comparison of the image quality produced by both the median filter of the golden standard and the FPGA. This shows how they perform on simple images with few and cleary defined edges.



(a) Standard Median Filter Simple House

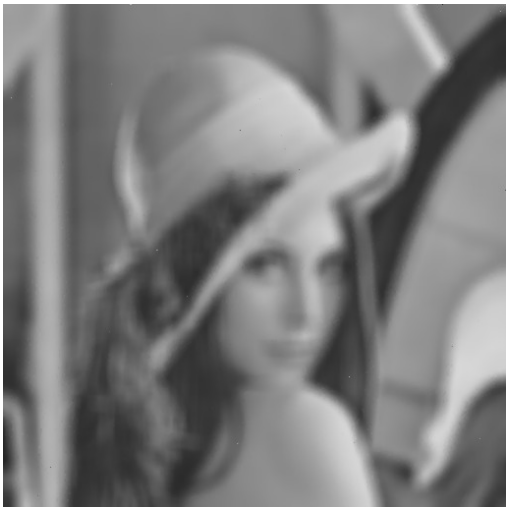
(b) FPGA Median Filter Simple House

Figure 4.3: Median Filter Comparison of Simple House

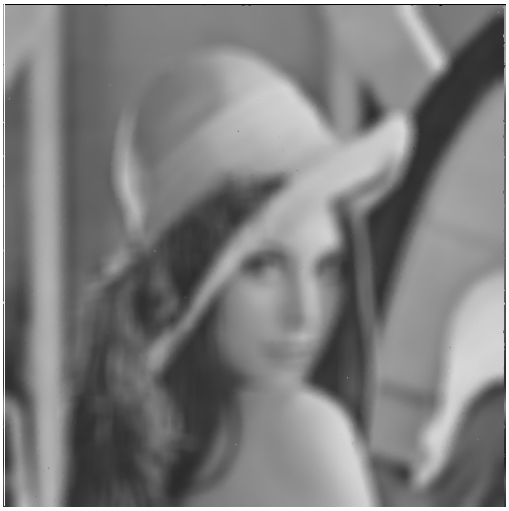
The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 501.5 ms
FPGA - 2.652 ms

Blurry Image Results

The following images show a side-by-side comparison of the image quality produced by both the median filter of the golden standard and the FPGA. This shows how they perform on blurry images.



(a) Standard Median Filter Blurry Image



(b) FPGA Median Filter Blurry Image

Figure 4.4: Median Filter Comparison of a Blurry Image

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 510.1 ms
FPGA - 2.652 ms

Detailed Image Results

The following images show a side-by-side comparison of the image quality produced by both the median filter of the golden standard and the FPGA. This shows how they perform on detailed images.

(a) Standard Median Filter Detailed Image

(b) FPGA Median Filter Detailed Image

Figure 4.5: Median Filter Comparison of a Detailed Image

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 510.7 ms
FPGA - 2.652 ms

Yoda Image Results

The following images show a side-by-side comparison of the image quality produced by both the median filter of the golden standard and the FPGA. This shows how they perform on YODA himself.

(a) Standard Median Filter a YODA Image

(b) FPGA Median Filter a YODA Image

Figure 4.6: Median Filter Comparison of a YODA Image

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 514.5 ms
FPGA - 2.652 ms

4.1.2 Discussion on Medial Filter Results

This section presents a detailed analysis of the image quality and performance improvements observed across various test images processed by both the golden standard software implementation and the FPGA-based implementation of the median filtering procedure.

Image Quality Comparison

For all images examined—including the Simple House (Figure 4.1a), Blurry, Detailed, and Yoda images—a significant reduction of noise is evident when comparing the filtered outputs to their original noisy counterparts. While minor residual noise persists in certain regions, such as small white specks along uniform, high-contrast edges, these artefacts do not substantially degrade the overall visual quality. These residuals arise from the fundamental operation of the median filter, which computes the median pixel value over a 3×3 neighbourhood; clusters of noise-altered pixels within this neighbourhood can occasionally cause the filter to retain noise values.

The FPGA outputs closely mirror the golden standard results in all cases, demonstrating that the hardware optimisations employed do not compromise the perceptual or structural accuracy of the filtered images. This is especially important given the application's sensitivity to image quality.

A consistent observation across all images is the presence of border artefacts, particularly visible as distortions or black bars along the edges in the FPGA outputs. This effect is attributable to the median filter's reduced neighbourhood size at image boundaries, where fewer than the full eight

neighbouring pixels are available. The reduced context challenges the calculation of reliable median values, leading to visible distortions confined to these peripheral regions.

Performance Speed-Up

Performance evaluation using speed-up metrics reveals substantial acceleration achieved by the FPGA implementation relative to the golden standard CPU software. Speed-up is defined as:

$$\text{Speed-up} = \frac{\text{Execution Time of Golden Standard}}{\text{Execution Time of FPGA Implementation}}$$

The observed speed-up values for the test images are as follows:

- Simple House Image: 189.1
- Blurry Image: 192.31
- Detailed Image: 192.57
- Yoda Image: 194

The observed speed-up between 189.1 and 194 for the images is particularly remarkable given the hardware constraints. The FPGA operates at a clock speed of 100 MHz, whereas the CPU running the golden standard implementation runs at a significantly higher clock speed of 4 GHz, which is 40 times faster in terms of raw clock frequency. Despite this disparity, the FPGA still achieves nearly 200 times faster processing.

This highlights the immense efficiency gains provided by the FPGA's parallel processing capabilities and custom hardware architecture, which allow it to outperform a general-purpose CPU running at much higher frequencies. Such a magnitude of speed-up underscores the advantage of dedicated hardware acceleration for computationally intensive tasks like median filtering, where massively parallel operations and pipelining can be leveraged to achieve superior throughput. This further reinforces the suitability of FPGA implementations in applications demanding real-time performance without sacrificing accuracy.

4.2 Sobel Edge Detection

The Sobel Edge Detection is applied to the median filtered image to assess how many notable edges have been preserved after filtration. The edge images will be compared to one another to determine whether the implemented algorithms achieve the same level of edge preservation.

The purpose of examining these images is to determine how much the remaining noise impacts the image quality and to assess whether the detected edges are sufficiently accurate for effective image reconstruction, if ever required.

(a) Standard Sobel Simple House

(b) FPGA Sobel Simple House

Figure 4.7: Sobel Comparison of Simple House

4.2.1 Results of Median Filter

Simple House Image Results

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 502.9 ms
FPGA - 2.678 ms

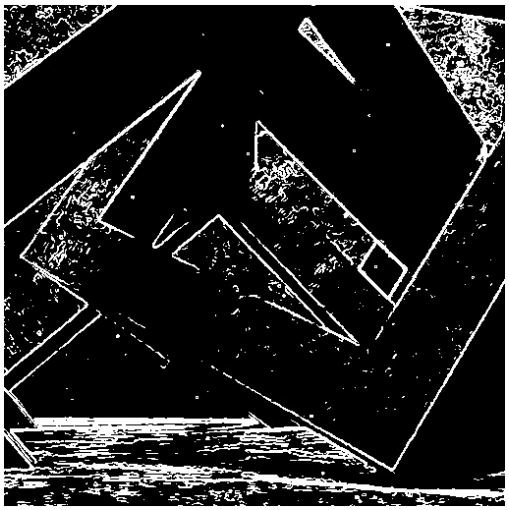
Blurry Image Results

(a) Standard Sobel Blurry Image

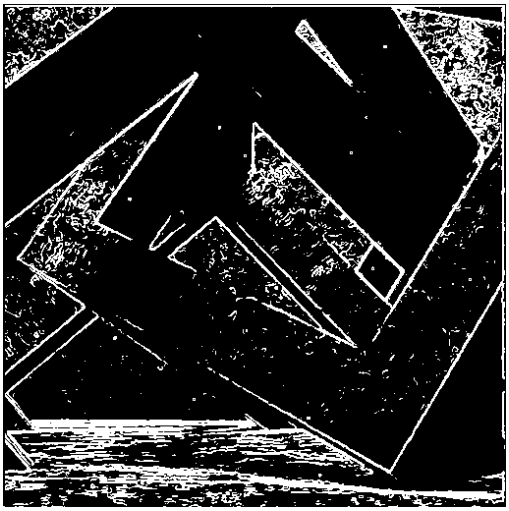
(b) FPGA Sobel Blurry Image

Figure 4.8: Sobel Comparison of Blurry Image

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 511.8 ms
FPGA - 2.678 ms



(a) Standard Sobel Detailed Image



(b) FPGA Sobel Detailed Image

Figure 4.9: Sobel Comparison of Detailed Image

Detailed Image Results

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 512 ms
FPGA - 2.678 ms

YODA Image Results

(a) Standard Sobel YODA Image

(b) FPGA Sobel YODA Image

Figure 4.10: Sobel Comparison of YODA Image

The times that both of these implementations took to produce the above results are as follows.
Golden Standard - 516.4 ms
FPGA - 2.678 ms

4.2.2 Discussion on Sobel Edge Detection Results

Image Quality Comparison

Overall, the images exhibit comparable performance in terms of edge preservation following filtration. Although the detected edges are not identical to those present in the original images, sufficient edge information remains for effective reconstruction in the majority of cases. It is important to note, however, that the blurred image contains significantly fewer detectable edges, which aligns with the inherent characteristic of blurring operations to attenuate high-frequency components, including edge details. Consequently, the reconstruction quality of the blurred image is unlikely to improve markedly over the original. Nevertheless, the continued presence of discernible edges in the filtered images indicates a satisfactory retention of critical structural features, demonstrating that the filtration process has largely preserved the essential edge information necessary for accurate image reconstruction.

Performance Speed-up

Using the formula defined in the previous section, the speed up of the various images was found to be:

- Simple House: 187.79
- Blurred Image: 191.11
- Detailed Image: 191.19
- YODA Image: 192.83

The discussion on this speed-up is similar to that produced in the median filter section. The observed speed-up of nearly 200 times is notably impressive considering the FPGA runs at only 100 MHz, while the CPU operates at 4 GHz, 40 times faster in clock speed. This demonstrates the significant efficiency of the FPGA's parallel processing and custom architecture, enabling it to outperform a much faster general-purpose CPU. Such results highlight the advantages of FPGA-based hardware acceleration for intensive tasks like median filtering and the Sobel edge detection algorithm implementation, making it well-suited for real-time applications that require both speed and accuracy.

Chapter 5

Conclusion

This project set out to develop a modular, high-performance image processing pipeline capable of removing impulse noise and enhancing image edges using FPGA-based digital acceleration. Through careful algorithm selection, software validation, and efficient hardware translation, the key objectives outlined in the project have been successfully achieved.

A 3×3 median filter was implemented to remove salt-and-pepper noise from images. The FPGA implementation closely matched the image quality of the MATLAB golden model across all tested images — from simple to highly detailed. Visual evaluation confirmed that noise was significantly reduced while maintaining the structural integrity of image features. Despite minor artifacts (e.g., residual specks and border distortion), the results show strong filtering performance.

One of the project's most significant accomplishments was the drastic reduction in processing time. The FPGA-based median filter achieved a speed-up of approximately 189× compared to the software implementation, completing each image in just 2.652 ms versus over 500 ms in MATLAB. This performance gain confirms that FPGAs are highly suited for localized image operations, where spatial parallelism and pipelined execution can be fully exploited.

The edge detection module, based on the Sobel operator, was implemented in both MATLAB and Verilog. The Verilog design maintains edge continuity and contrast. Directional masks and thresholding were successfully applied, yielding binary edge maps that retained essential boundary features.

The entire pipeline — from noise filtering to edge detection — was synthesised and tested on the Spartan-7 FPGA platform. The modules provided substantial performance on static images and established a reusable foundation for future extensions.

5.1 Future Work Recommendations

Several enhancements could further improve the system's robustness and flexibility:

- **Refined Border Handling:** The current median filter exhibits visible artefacts at the image boundaries due to insufficient neighbouring pixels in the 3×3 window. Future work could involve implementing zero-padding, edge-replication, or adaptive padding strategies to mitigate these distortions and enhance visual consistency across the image.
- **Support for RGB to Grayscale Conversion:** Although the current implementation assumes grayscale inputs, incorporating an on-chip RGB to grayscale conversion module would

increase the pipeline's flexibility and allow it to handle a wider range of input formats, especially in real-world embedded vision systems where image sensors typically output in RGB.

- **Adaptive Thresholding for Edge Detection:** The inclusion of adaptive methods such as Otsu's algorithm could further improve edge map accuracy under varying lighting or contrast conditions.
- **Real-Time Video Support:** Extending the current system to process video frames in real time would mark a significant step toward practical deployment.
- **OpenCL and Machine Learning Integration:** Completing the planned OpenCL implementation for cross-platform testing and incorporating lightweight CNN models for denoising or learned edge detection would broaden the pipeline's capabilities and enable smarter, context-aware processing.

Bibliography

[1] P. Phukan, K. S. Raju, and G. Baurah, "An fpga implementation of a fast 2-dimensional median filter," in *Proceedings of the National Conference on Recent Advances in Communication, Control and Computing Technology (RACCCT)*. Tezpur, India: SCET, 2012, pp. 144–146, iSBN: 978-81-88894-34-5. [Online]. Available: https://www.researchgate.net/publication/317662545

[2] M. A. Vega-Rodríguez, J. M. Sánchez-Pérez, and J. A. Gómez-Pulido, "An fpga-based implementation for median filter meeting the real-time requirements of automated visual inspection systems," *Proceedings of the 10th Mediterranean Conference on Control and Automation (MED)*, pp. 1–6, 2002. [Online]. Available: https://www.researchgate.net/publication/238438181

[3] G. Ravivarma, K. Gavaskar, D. Malathi, K. G. Asha, B. Ashok, and S. Aarthi, "Implementation of sobel operator based image edge detection on fpga," *Materials Today: Proceedings*, vol. 45, pp. 2401–2407, 2021, presented at the International Conference on Advances in Materials Research – 2019.

[4] G. Gupta, "Algorithm for image processing using improved median filter and comparison of mean, median and improved median filter," *International Journal of Soft Computing and Engineering*, vol. 1, no. 5, pp. 304–311, Nov. 2011.

Appendix

5.2 Git Repository Link

All code can be found in GitHub repository.