YODA Project (Group 8)
IEDP - Image Edge Detection Processing

Kwanele Mabanga[†] Atrisha Maribe[‡], Zolile Zoko, [§]Khanyisa Hlungwani[¶]
EEE4120F
University of Cape Town
Group Name: R2-D2.CFG
[†]MBNKWA005 [‡]MRBATR001 [§]ZKXZOL001 [¶]HLNKHA010

***Abstract*—This paper focuses on the design and implementation of a digital accelerator for a multi-stage image processing pipeline. This pipeline begins with a median filtering algorithm aimed at reducing noise in RGB images while preserving the edges, followed by greyscale conversion using a standard luminance-based weighted average, and then applying the Sobel operator for edge detection. The hardware architecture, described in Verilog HDL (VHDL), is optimized for efficient full-frame processing. The design leverages line buffers, a sliding 3×3 window, and a finite state machine (FSM) for full-frame processing on a 320×240 image. To validate functional correctness, a golden measure was developed in C, implementing the same algorithmic pipeline.**

I. Introduction

In image processing, median filtering is a widely used noise reduction technique that preserves edges while removing outliers, for example, salt-and-pepper noise, characterized by random occurrences of black and white pixels, as shown in Figure 1.

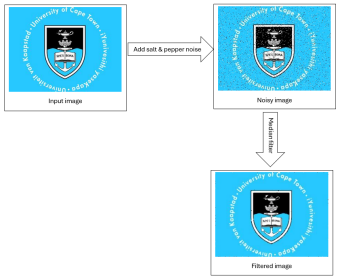

Fig. 1: Effect of median filtering on an image corrupted by salt-and-pepper noise

This project is centred around implementing a basic median filter where each output pixel is derived from the median of brightness values in a 3×3 window. The brightness is calculated by summing the R, G, and B components of each pixel. The pipeline is extended by converting the filtered image to grayscale and then applying a Sobel operator for edge detection.

This report starts by outlining a brief overview of image processing principles. It then moves on to discuss in detail the design of an image edge detection processing system, giving a high-level systems view through block diagrams and schematics, and later diving into code snippets that reveal the key functionality of each stage in the pipeline. Simulation results are presented to validate the system, with a comparative analysis between the Verilog design and a C-based golden measure. The report concludes with a summary of key findings and recommendations for future work.

The following are key objectives of this project, aimed at developing and validating an efficient image processing pipeline:

- To design and implement a software-based median filter for real-time image noise reduction.
- To develop a full image processing pipeline using the Sobel operator, including grayscale conversion and edge detection.
- Develop and simulate the design using Verilog and compare functional accuracy and performance against a C-based golden measure code.

II. Background

A. Median Filter

The median filter is a nonlinear technique widely used to reduce impulsive noise in digital images, such as salt-and-pepper noise. Unlike linear filters that average pixel values and tend to blur edges, it replaces the center pixel in a neighborhood with the median of its surrounding pixel values, preserving edges while removing outliers. In this project, we calculate the brightness of each RGB pixel in a 3×3 window by summing its red, green, and blue components. These brightness values are then sorted to determine the median, which replaces the central pixel. [1]

Median Filter Algorithm:

1) For each pixel (excluding borders), extract the 3×3 neighborhood.
2) Compute brightness: brightness $= R + G + B$ for each pixel in the window.
3) Sort the 9 brightness values.
4) Assign the median value to the center pixel in the output image.

B. Greyscale Conversion

After noise reduction, the image is converted to grayscale to prepare for edge detection. Grayscale conversion reduces the three-channel RGB image to a single-channel intensity image using a standard luminance-based weighted average formula based on human visual perception:

$$\text{Gray} = 0.299R + 0.587G + 0.114B \tag{1}$$

C. Sobel Edge Detecion

To identify edges in the image, the Sobel operator is applied to the grayscale image. The Sobel operator is a gradient-based edge detection technique that uses two 3×3 convolution kernels to estimate the image gradient in horizontal and vertical directions. The resulting gradient magnitude highlights areas of rapid intensity change, effectively identifying edges in the image. This combination of median filtering and Sobel edge detection creates a robust pipeline for noise-resistant edge detection on hardware. [2]

Sobel Edge Detection Algorithm:

1) Apply the G_x and G_y kernels to the grayscale image:

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

2) For each pixel, compute:
   - Gradient X = sum of element-wise product with G_x
   - Gradient Y = sum of element-wise product with G_y
3) Compute gradient magnitude:

$$\text{Edge} = |G_x| + |G_y| \text{ (approximation of } \sqrt{G_x^2 + G_y^2}) \tag{3}$$

III. Methodology

A. Proposed Development Strategy

This project aims to develop a digital image processing accelerator for image edge detection processing implemented on an FPGA platform. The development is structured into two progressively complex design versions: the first focusing on noise reduction, and the second integrating a full edge detection pipeline.

Both versions target grayscale images of 320×240 resolution, and are implemented using synthesized HDL modules. The final implementation is simulated and evaluated using the Vivado Design Suite.

Version 1: Median Filtering Accelerator: The initial version establishes our baseline architecture, focusing on noise reduction through median filtering. This accelerator applies a 3×3 median filter to reduce noise from a 320×240-pixel input image. The design consists of three primary modules: an **Image Loader** that fetches image data from a .mem file into on-chip memory, a **Median Filter Core** that processes each pixel using a sliding 3×3 window, and an **Output Writer** that stores filtered results to memory or an output buffer.

Version 2: Edge Detection Accelerator: Building upon Version 1, the second version expands the architecture to implement a complete edge detection pipeline. This enhanced accelerator detects edges by first converting an RGB image to grayscale and then applying the Sobel edge detection operator. The design incorporates an RGB-to-Grayscale Converter that transforms color input to intensity values, an Edge Detection Core that implements Sobel filtering for gradient computation, and an Output Writer that stores edge-detected results to memory.

Due to time constraints, this version remains partially implemented. The greyscale conversion has been designed and simulated but on a 60×60 pixel image due to memory issues. Further work would involve fully implementing the edge detection pipeline on a 320×240 as done in the golden measure.

B. Performance Evaluation

Performance Metric	**Description**	**Tools**
Latency	Time taken from input of first pixel to output of last pixel.	Measured via simulation waveform (Vivado).
Throughput	Pixels processed per second.	Derived from latency and clock frequency.
Resource Utilization	Use of BRAM, LUTs.	Post-synthesis and implementation reports.
Power Consumption	Dynamic and static power.	Vivado Power Estimator.
Functional Correctness	Compare with C-based golden measure.	Compare FPGA output to C-based golden measure output.

TABLE I: Performance Metrics

To evaluate the performance of the digital accelerator system, the metrics summarized in Table I will be used where applicable, as they help quantify how efficient and practical the design is for real-time or embedded image processing tasks. Latency measures how quickly a 320×240 pixel image can be processed by the accelerator, from the first input pixel to the last output pixel, and serves as an indicator of the system's suitability for time-sensitive applications.

Throughput, measured in pixels per second, indicates the number of pixels the system can process over time, which is particularly relevant for high-speed or continuous image processing scenarios. Resource utilization, in terms of LUTs, BRAMs, FFs, and DSPs, provides insight into how efficiently the design uses the available FPGA hardware.

Power consumption reflects the amount of power drawn due to logic activity and memory accesses, which is an important consideration in energy-constrained or battery-powered environments.

Lastly, functional correctness is evaluated by comparing the output images generated by the FPGA to C-based golden measure output, ensuring the system delivers correct and reliable results.

IV. Design

A. Design Overview

The digital accelerator system is designed to process 320×240 pixel images using FPGA hardware and is implemented in synthesized HDL (Verilog). The design consists of modular processing stages that support different image operations, such as median filtering (Version 1) and grayscale conversion, followed by edge detection (Version 2).

At the core of the design is a data flow architecture built around modular HDL blocks, each performing a specific task. The modules communicate via well-defined I/O ports, which support pixel streaming and synchronization through simple valid/ready handshaking.

B. Golden Measure Implementation (C Language)

The C code provided serves as the golden measure for the digital accelerator. It defines the precise algorithms and expected output for each stage of the image processing pipeline.

Image Processing Pipeline Stages: The pipeline takes an RGB image as input and produces an edge-detected grayscale image as output, with intermediate images (filtered RGB, grayscale) also produced.

Stage 1: Median Filter: This stage reduces noise in the input RGB image using a 3x3 median filter. The input is an RGB image (`unsigned char *input`, 3 bytes per pixel), and the output is the median-filtered RGB image (`unsigned char *output`, also 3 bytes per pixel). For each pixel in the image, excluding the border pixels which are copied directly, a 3x3 window of RGB pixels is collected.

The brightness of each pixel is calculated as the sum of its red, green, and blue components, i.e., $R + G + B$. The pixels in the window are sorted in ascending order based on this brightness using bubble sort. The RGB values of the median pixel (the 5th pixel in the sorted list) are then written to the output image. The implementation of this stage is shown in Listing 1.

Listing 1: Serial Implementation of the `MedianFilter` function.

```
RGB win[WINDOW_SIZE * WINDOW_SIZE];
int idx = 0;

// Collect 3x3 neighborhood
for (int dy = -1; dy <= 1; dy++) {
    for (int dx = -1; dx <= 1; dx++) {
        int ni = ((y + dy) * width + (x + dx)) * 3;
        win[idx].r = input[ni];
        win[idx].g = input[ni + 1];
        win[idx].b = input[ni + 2];
        idx++;
    }
}

// Sort pixels by brightness using bubble sort
for (int k = 0; k < 8; k++) {
    for (int l = 0; l < 8 - k; l++) {
        if (BRIGHTNESS(win[l]) > BRIGHTNESS(win[l + 1])) {
            RGB tmp = win[l];
            win[l] = win[l + 1];
            win[l + 1] = tmp;
        }
    }
}

// Assign median pixel to output
output[curr_idx]     = win[4].r;
output[curr_idx + 1] = win[4].g;
output[curr_idx + 2] = win[4].b;
```

Stage 2: Grayscale Conversion: In this stage, the median-filtered RGB image is converted into a grayscale image. The input is an RGB image (`unsigned char *input`, 3 bytes per pixel) and the output is a grayscale image (`unsigned char *output`, 1 byte per pixel). For each pixel, the grayscale value is computed using the standard luminance formula: Gray $= 0.299 \times R + 0.587 \times G + 0.114 \times B$. The implementation is shown in Listing 2.

Listing 2: Serial Implementation of the `ConvertToGreyscale` function.

```
uint8_t r = input[idx];
uint8_t g = input[idx + 1];
uint8_t b = input[idx + 2];

// Convert using luminance formula
output[y * width + x] = (uint8_t)(
    0.299 * r + 0.587 * g + 0.114 * b
);
```

Stage 3: Sobel Edge Detection: This stage detects edges by applying Sobel operators to the grayscale image. The input is a grayscale image (`unsigned char *grey`, 1 byte per pixel), and the output is an edge map (`unsigned char *edges`, 1 byte per pixel), where the intensity indicates edge strength. For each pixel (excluding the borders), a 3x3 window is extracted. Horizontal (G_x) and vertical (G_y) Sobel kernels are convolved with the window to compute gradient components. The edge magnitude is then calculated as $\sqrt{G_x^2 + G_y^2}$, and clamped to the range [0, 255]. The implementation is provided in Listing 3.

Listing 3: Serial Implementation of the `SobelEdgeDetection` function.

```
int sumX = 0, sumY = 0;

// Apply Sobel kernels
for (int j = -1; j <= 1; j++) {
    for (int i = -1; i <= 1; i++) {
```

```
        int pix = grey[(y + j) * width + (x + i)];
        sumX += gx[j + 1][i + 1] * pix;
        sumY += gy[j + 1][i + 1] * pix;
    }
}

// Compute edge magnitude
int mag = (int)sqrt((double)(sumX * sumX + sumY * sumY));
if (mag > 255) mag = 255;

edges[y * width + x] = (uint8_t)mag;
```

C. Verilog HDL Implementation

Block Diagram

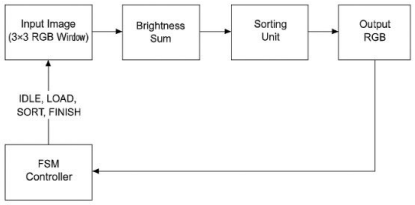

Fig. 2: Block Diagram

- **Input Image (3×3 RGB Window):** A 3×3 block of pixels is input for processing. Each pixel consists of 8-bit R, G, and B channels. These values are stored in registers and represent the local neighborhood of the center pixel.
- **Brightness Sum:** For each of the 9 pixels, the R + G + B values are computed to determine brightness. This results in 9 brightness values (brightness[0] to brightness[8]), each 24 bits wide, stored in temporary registers for the sorting stage.
- **Sorting Unit:** A bubble sort algorithm is used to sort the 9 brightness values in ascending order. During swaps, the corresponding R, G, and B values are also swapped in parallel to maintain alignment. After sorting, the median pixel is located at index 4.
- **Median Extraction (Output RGB):** The pixel at index 4 (median brightness) is selected as the output RGB value. This becomes the filtered value for the center pixel in the image.
- **FSM Controller:** A finite state machine (FSM) governs the flow across 4 states:
  - **IDLE:** Waits for the start signal and passes through the unmodified 3×3 input window.
  - **LOAD:** Captures RGB inputs and calculates brightness values.
  - **SORT:** Executes the bubble sort across brightness values.
  - **FINISH:** Extracts and writes the median RGB value to output, then asserts the done signal.

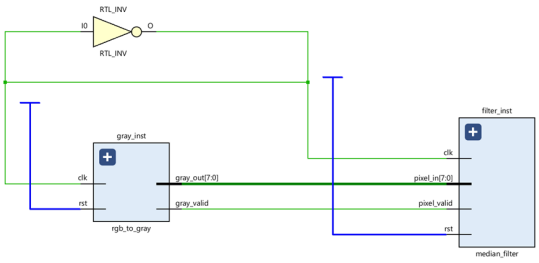

Fig. 3: RTL schematic : Grayscale converter and Median filter Pipeline

RTL Schematic

D. Input Data Consideration: Grayscale vs RGB

The input image is 320 × 240 pixels in size. In RGB format, each pixel occupies 3 bytes (one byte per component), resulting in a total image size of 3 × 320 × 240 = 230,400 bytes (230.4 KB). In contrast, grayscale images use only 1 byte per pixel, yielding a more compact size of 1 × 320 × 240 = 76,800 bytes (76.8 KB).

This distinction is significant for our design decisions. Grayscale images are more memory-efficient and ideal for resource-constrained systems. However, RGB images carry more color detail and are useful for more advanced image analysis.

E. On-Chip Image Storage

The accelerator stores input images in block RAM (BRAM) during processing. RGB images are stored with a 24-bit width, while grayscale images use an 8-bit width. The system leverages line buffers and a sliding window approach to reduce memory usage while enabling spatial operations like filtering.

For the median filter, a bubble sort algorithm is used to compute the median value within a windowed region. The access pattern is sequential and aligned to the line-buffered structure.

F. Resource Considerations

Our design carefully balances the utilization of key FPGA resources to achieve optimal performance within hardware constraints:

Block RAMs (BRAMs) serve as the primary storage elements for image data and intermediate results. We strategically allocate BRAMs to implement line buffers and window buffers while minimizing unnecessary data duplication.

Look-Up Tables (LUTs) implement the combinational logic required for pixel processing operations, including comparisons, sorting networks for median computation, and

gradient calculations for edge detection.

Flip-Flops (FFs) are employed throughout the design for pipelining and temporary storage of both data and control signals, helping maintain timing closure while maximizing throughput.

Digital Signal Processors (DSPs) are primarily utilized in Version 2 for the multiply-accumulate operations needed during Sobel edge detection, efficiently handling the gradient computations without consuming general-purpose logic resources.

G. Testbench and Integration

A testbench is used to instantiate and simulate the accelerator modules, defining input patterns, expected behavior, and inter-module signaling. The modular design allows for straightforward integration and future adaptation to host-controlled or autonomous processing systems.

V. Planned Experimentation

A. Verilog Expermintation

To evaluate the effectiveness and correctness of the designed accelerator, a series of experiments will be conducted in simulation using Vivado. The test setup involves loading pre-prepared image data from .mem files into the FPGA simulation environment and observing the processed output using waveform inspection and memory dumps. These experiments are executed through Tcl scripts or manual simulation runs using the Vivado GUI.

For **Version 1 (Median Filter)**, a noise-injected grayscale image is processed by the HDL median filter. The output is extracted from simulation memory and compared with a reference output generated by a golden measure written in C. Commands such as `launch_simulation`, `run all`, and `write_mem` are used during simulation, while synthesized resource utilization is obtained via `report_utilization`.

Functional simulation ensures logic correctness, and power analysis is planned using the Vivado Power Estimator.This experimental setup ensures thorough validation of both functional and performance characteristics of the system before considering hardware deployment.

B. Golden Measure Experimentation

To evaluate the correctness of the HDL image processing implementation, a software-based golden measure was developed and executed using a GUI-based image processing pipeline (as shown in Figure 4). This GUI enables loading an image, applying a median filter, converting to grayscale, and performing edge detection. It also provides timing data for each stage of the pipeline.

The original image is passed through each processing stage, and the resulting outputs—filtered image, grayscale image, and edge-detected image—are displayed. The final outputs

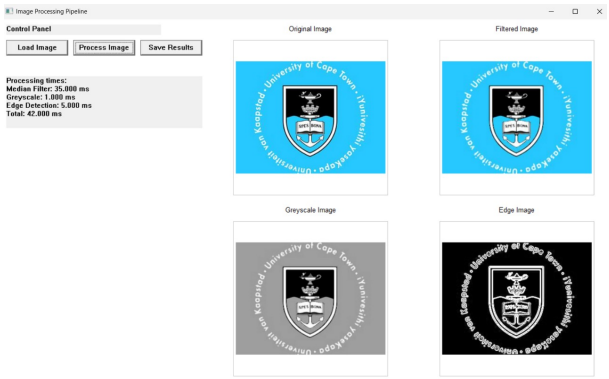

Fig. 4: Image Processing GUI (Golden Measure)

generated by this software pipeline are saved and used as reference results (i.e., the golden measure) against which the Verilog simulation results are compared.

To comprehensively evaluate the system's behavior, multiple test 320×240 pixel images were used.

VI. Results and Discussion

The Verilog implementation was validated in Vivado using a 60×60 RGB image. A testbench delivered pixel data sequentially, and the output was assessed against a C-based reference (golden measure).

Functional Verification Outputs from both Version 1 (Median Filter) and Version 2 (Grayscale) were compared to the golden reference on a pixel-by-pixel basis. The results showed a high degree of similarity, verifying that the Verilog design functions correctly as intended.

Simulation Output Analysis

Figure 5 displays the simulation waveform captured in Vivado, illustrating the activity of output signals during valid clock cycles. The waveform confirms that pipeline stages are properly synchronized and that buffer management and control handshaking are implemented effectively.

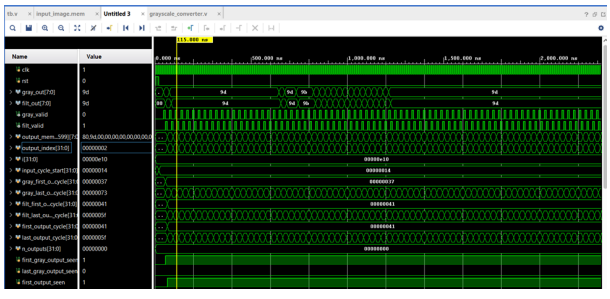

Fig. 5: Simulation waveform showing pixel output timing

Additionally, performance metrics observed during simulation are shown in Figure 6. For Version 2, processing a full image required approximately 260,000 clock cycles. At an operating frequency of 100 MHz, this translates to

roughly 2.6 milliseconds per image, equating to an estimated throughput of 123 frames per second—making the system suitable for real-time applications.

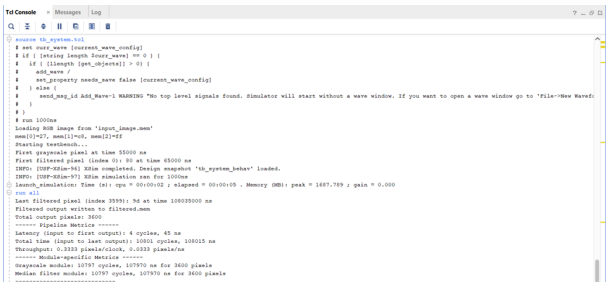

Fig. 6: Simulation performance metrics

Performance Speedup Analysis

While the full-scale image processed by the golden reference was 320×240 pixels, the Verilog simulation was constrained to a smaller 60×60 image due to simulation limitations. To enable a fair comparison, execution times were normalized based on image size, assuming linear complexity with respect to the number of pixels.

The scaling factor is calculated as:

$$\frac{320 \times 240}{60 \times 60} = \frac{76,800}{3,600} = 21.3$$

Execution time measurements:

- **Golden Reference (320×240)**: 37 milliseconds
- **Verilog (60×60)**: 108,015 nanoseconds = 0.108015 milliseconds

Estimated Verilog processing time for a full 320×240 image is:

$$T_{\text{verilog scaled}} = 0.108015 \times 21.3 \approx 2.3 \text{ milliseconds}$$

Thus, the performance speedup is:

$$\text{Speedup} = \frac{T_{\text{golden}}}{T_{\text{verilog scaled}}} = \frac{37}{2.3} \approx 16.1\times$$

Conclusion: After scaling for image size, the Verilog implementation demonstrates an approximate **16.1× speedup** over the software-based golden reference. This result is based on linear scaling assumptions and should be interpreted cautiously, as actual performance may be influenced by simulation constraints and hardware-level parallelism.

VII. Conclusion

The Verilog-based implementation successfully demonstrated median filtering and grayscale conversion, producing correct output that aligns with the expected results from the golden reference. Despite being tested on a smaller image due to simulation constraints, performance estimates adjusted for image size suggest a significant speedup of approximately **16.1**× over the software-based golden reference.

This highlights the potential of FPGA acceleration in image processing applications, particularly for noise reduction and grayscale conversion tasks. The output images confirm the functional design's correctness, while the performance analysis demonstrates the advantages of hardware parallelism.

Although the Sobel filter was not completed, the current results lay a strong foundation for continued development. Implementing the edge detection stage, transitioning to hardware testing, and standardizing input sizes will be critical next steps. Ultimately, incorporating a real-time display system will further enhance the usability and completeness of the design.

VIII. Appendix

A. GitHub Repository Link

Please find all codes used for this project at the following GitHub repository:
https://github.com/ALECIA13/EEE4120F--YODA

B. Vivado Schematic

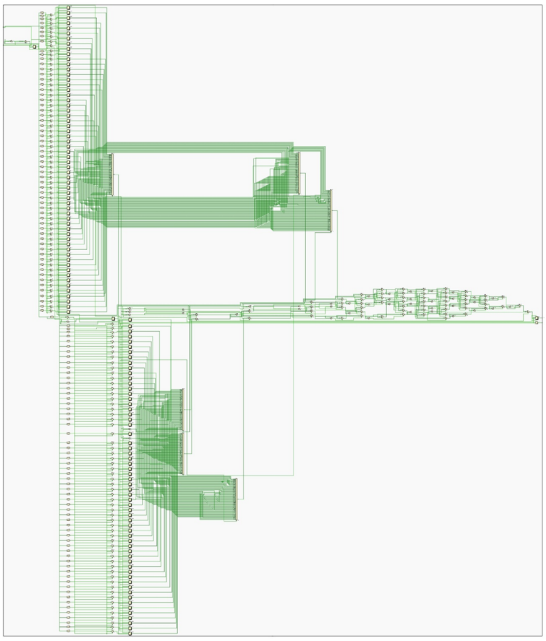

Fig. 7: Vivado Schematic

REFERENCES

[1] S. A. Villar, S. Torcida, and G. G. Acosta, "Median filtering: a new insight," *Journal of Mathematical Imaging and Vision*, vol. 58, pp. 130–146, 2017.

[2] R. Kasturi and B. Chanda, "Machine vision, chapter 5: Edge detection," https://cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter5.pdf, [Accessed 18-05-2025].