FPGA Camera Filters

Crihălmeanu Tudor Mihai

Structure of Computer Systems Project

Technical University of Cluj-Napoca

October 2024

Contents

1 Introduction

1.1 Context

The continuous advancements in embedded systems and real-time image processing have led to increased demand for efficient and flexible hardware platforms capable of handling complex algorithms. Field Programmable Gate Arrays (FPGAs) stand out in this domain, offering parallel processing capabilities that significantly enhance performance in tasks such as image filtering. The Zybo Z7-20 development board, equipped with the Zynq 7000 SoC, provides an ideal environment for exploring these applications, integrating both programmable logic and a processing system.

This project leverages the PCam 5C camera module and the Zybo Z7-20 to implement real-time camera filters, demonstrating the power of FPGA-based solutions in image processing tasks.

1.2 Objectives

The primary objective of this project is to implement and test a set of camera filters: *Grayscale, Inverted, Edge Detecion and Gaussian Blur* - on the Zybo Z7 FPGA using the PCam 5C camera module. These filters will be executed in real-time, leveraging the Zynq 7000 processor for optimized performance. The project aims to demonstrate the FPGA's capabilities in handling computationally intensive tasks like image processing while providing an understanding of hardware-software design techniques.

- **Grayscale**: The grayscale filter converts a color image into shades of gray by removing color information and representing intensity as a single luminance value.
- **Inverted**: The inverted filter transforms an image by reversing the color values of each pixel, resulting in a negative effect where dark areas become light and vice versa.
- **Edge Detection**: The edge detection filter identifies and highlights the boundaries or edges within an image by detecting significant changes in pixel intensity, which can be useful for object recognition and analysis.
- **Gaussian Blur**: The Gaussian blur filter smooths an image by applying a Gaussian function to reduce noise and detail, creating a softening effect that helps in reducing sharp edges and providing a more visually pleasing result.

1.3 Timeline

1. **Week 1: Project Planning and Requirements Gathering** - Defining project scope and objectives, gathering technical specifications for the Zybo Z7-20 FPGA, PCam 5C camera, and Zynq 7000 processor, and researching existing implementations of camera filter algorithms.
2. **Week 2: System Setup and Environment Configuration** - Set up the development environment for the Zybo Z7-20, configure the PCam 5C camera, creating the VGA module and conducting initial tests to verify hardware functionality.

3. **Week 3: Grayscale Filter Implementation** - Designing the algorithm for the grayscale filter, implementing the filter on the FPGA using VHDL and HLS and testing the grayscale filter using the camera.
4. **Week 4: Inverted Filter Implementation** - Developing the inverted filter algorithm derived from the existing grayscale implementation, optimize the inverted filter on the FPGA, and conduct tests to ensure correct functionality and performance.
5. **Week 5: Edge Detection Filter Implementation** - Researching edge detection algorithms (e.g., Sobel, Canny), implementing the selected edge detection algorithm on the FPGA, and testing the edge detection filter.
6. **Week 6: Gaussian Blur Filter Implementation** - Designing the Gaussian blur algorithm focusing on kernel size, implementing the Gaussian blur filter on the FPGA, and validating the output using the camera.
7. **Week 7: Final Testing and Documentation** - Conducting comprehensive testing of all implemented filters, gathering results and analyzing performance metrics, and documenting the project including methodology, results, and conclusions.

2 Bibliographic Research

2.1 What is a Camera Filter?

A camera filter in digital image processing is a transformation applied to pixel values of an image, altering its visual representation. Filters can range from simple operations, such as converting an image to grayscale, to more complex transformations like edge detection or Gaussian blur. These filters are commonly implemented in hardware for real-time applications, especially when high performance is required.

2.2 How are Filters Implemented on FPGAs?

Filters are implemented on FPGAs by leveraging the parallel processing capabilities of the hardware. FPGAs consist of Configurable Logic Blocks (CLBs), Digital Signal Processing (DSP) units, and Block RAM (BRAM). Basic filters, such as Grayscale and Inverted, use simple logic operations that are mapped onto CLBs, while more complex filters like Gaussian Blur use DSP units for efficient arithmetic operations, such as convolution.

In addition to traditional hardware description languages (HDLs), High-Level Synthesis (HLS) tools are often used to implement filters on FPGAs. HLS allows developers to write algorithms in high-level languages like C/C++ and automatically generates the hardware description. This reduces development time and enables quick optimization of filters. HLS is particularly beneficial for handling complex image filters by automating parallelism and resource allocation, resulting in faster processing and real-time performance.

2.3 What are the Advantages of Hardware-Based Filters?

Hardware-based filters, particularly those implemented on FPGAs, offer significant advantages over software-based solutions. These include:

- **Low Latency**: FPGAs process data in parallel, which reduces the time required to apply filters, making them suitable for real-time applications.
- **Resource Efficiency**: Filters implemented in hardware use dedicated logic resources, freeing up the main processor for other tasks.
- **Customization**: FPGAs allow developers to optimize the implementation of filters based on the specific needs of the application, adjusting resource usage and performance.

2.4 What Are the Challenges in FPGA-Based Image Processing?

Implementing camera filters on FPGAs poses several challenges, including the need for efficient memory management and careful optimization of resource usage. Complex filters like Gaussian blur require significant computational power, and optimizing these for FPGA architectures involves balancing between the available hardware resources (e.g., DSP blocks, memory) and performance requirements.

3 Analysis

3.1 Project Proposal

The project aims to implement real-time camera filters on an FPGA using the Pcam 5C imaging module, which incorporates the Omnivision OV5640 5-megapixel color image sensor. The Pcam 5C communicates with the FPGA via a dual-lane MIPI CSI-2 interface, allowing for high-speed data transmission that supports video streaming formats such as 1080p at 30 frames per second. The camera module connects to the FPGA development board through a 15-pin flat-flexible cable (FFC), facilitating the transfer of pixel data and control signals.

To interface with the FPGA, the Pcam 5C will utilize the MIPI CSI-2 interface, which requires the FPGA to have a compatible MIPI CSI-2 controller. This will allow the FPGA to receive image data from the camera effectively. Digilent provides open-source Vivado IP cores that enable the integration of the MIPI CSI-2 interface within the FPGA, streamlining the process of capturing and processing image data. The image sensor will be configured using a Serial Camera Control Bus (SCCB), which operates similarly to I2C, to control various parameters such as exposure, white balance, and image format.

Once the image data is received by the FPGA, the project will implement a set of filters—Grayscale, Inverted, Edge Detection, and Gaussian Blur—using High-Level Synthesis (HLS). HLS tools allow developers to write filter algorithms in high-level programming languages like C or C++, which are then synthesized into hardware description language (HDL) code suitable for FPGA implementation. This approach facilitates rapid development and testing of the filters, enabling the exploration of various optimization techniques.

The filters will leverage the parallel processing capabilities of the FPGA, where each filter can operate on multiple pixels simultaneously. For example, the Gaussian Blur filter will use convolution operations that can be efficiently implemented using FPGA DSP blocks, while simpler filters like Grayscale and Inverted can be mapped onto Configurable Logic Blocks (CLBs). By utilizing HLS, the project aims to optimize the implementation of these filters, allowing for real-time performance while maintaining flexibility for future enhancements

3.2 Project Challenges and Considerations

1. **High Data Throughput Requirements:** The PCam 5C camera module is capable of delivering high-definition video at 1080p resolution at 30 frames per second. This level of throughput requires the FPGA to process a substantial amount of pixel data in real-time. Given the image resolution and frame rate, the system must handle data transfer and processing efficiently, minimizing latency in the video pipeline.

2. **Parallel Processing Needs:** Filters such as grayscale, inverted color, and Gaussian blur require pixel-by-pixel manipulation. The FPGA's architecture allows for parallel processing, which is essential for achieving the required speed for real-time performance. Implementing these filters with a focus on parallelism maximizes the FPGA's processing power and reduces the overall time needed for each frame.

3. **Memory Bandwidth and Frame Buffering:** The camera's high data rate also impacts memory bandwidth requirements. The AXI Video DMA IP core facilitates buffered access to the FPGA's DDR memory, enabling temporary storage of image frames and providing smooth data flow between the video stream and processing stages. Double-buffering strategies may be required to ensure that frames are processed and displayed without delay.

4. **Integration of AXI Protocol and IP Cores:** Utilizing the AXI protocol allows for streamlined communication between the FPGA's processing system (PS) and programmable logic (PL). The project will leverage various AXI4-Stream IP cores for each filter and video processing task. A detailed understanding of the AXI interconnect and addressing scheme is essential to correctly route and control the data flow through the system.

5. **Synchronization and Timing Control:** Since real-time video output is required, precise timing control is critical. The FPGA's video pipeline must synchronize with the display's refresh rate to avoid lag and ensure smooth transitions between frames. The project must account for timing control between IP cores, particularly with respect to video clock signals, as well as managing any latency introduced by memory access or processing delays.

3.3 High-Level System Architecture

The FPGA design in this project involves several interconnected components that together form the video processing pipeline. The main architectural elements include:

- **Camera Input Interface:** The MIPI CSI-2 interface is used to transfer image data from the PCam 5C to the FPGA. This high-speed interface allows data to be received and formatted into an AXI4-Stream format, making it compatible with other IP cores in the video processing pipeline.

- **AXI-Stream Processing Pipeline:** Each filter—grayscale, inverted, edge detection, and Gaussian blur—operates within the AXI4-Stream framework, ensuring efficient data transfer between processing blocks. The use of the AXI4-Stream protocol facilitates parallel data handling and reduces the overhead associated with traditional data transfer methods.
- **DDR Memory Management with AXI VDMA:** The AXI Video DMA (VDMA) IP core allows the FPGA to store and retrieve image frames in DDR memory, ensuring a steady flow of frames for processing and display. This core manages buffering and access to memory resources, enabling real-time double-buffering to prevent display lag and data loss.
- **Processor Integration for Control Tasks:** The Zynq-7000 SoC integrates an ARM processor alongside the FPGA fabric, which can be used to manage and control video processing tasks, such as selecting filters or adjusting settings via a user interface. The ARM processor communicates with the AXI interconnect, coordinating data flow and filter parameters as needed.

4 Design

4.1 Camera-to-FPGA Connection Using IP Cores and AXI Protocol

The camera module, PCam 5C, connects to the Zybo Z7-20 FPGA through a MIPI CSI-2 (Camera Serial Interface-2) protocol. This protocol allows high-speed image data transfer, essential for handling real-time video feeds. Below is a step-by-step breakdown of this setup and how the data flows from the camera to the FPGA for processing.

1. **Physical and Protocol Layer IP Cores**
   - **MIPI_D_PHY_RX**: This IP core handles the physical layer of the MIPI CSI-2 interface. It converts the serial data from the camera into a parallel format that can be processed by the FPGA. Data from the PCam 5C's dual-lane MIPI CSI-2 output is de-serialized and merged into a single data stream.
   - **MIPI_CSI_2_RX**: After de-serialization, the data moves to this IP core, which implements the protocol layer for MIPI CSI-2. Here, the data stream is interpreted as image frames with pixels and lines. The formatted data is then packed into an AXI-Stream format, creating a compatible input for the subsequent video processing pipeline on the FPGA.
2. **Image Preprocessing and AXI-Stream Conversion**
   - The **AXI_BayerToRGB** core interpolates the raw Bayer data (output from the MIPI CSI-2 RX) into RGB format, suitable for typical image processing tasks.
   - **AXI_GammaCorrection** then applies gamma correction to the RGB data, adjusting the color depth to 8 bits per channel (24 bits total). The output is a high-quality RGB image, ready for additional filters.

3. **Video Processing Pipeline**

   - The filtered data is streamed through various AXI4-Stream IP blocks that handle specific filter operations. For this project, this includes cores for grayscale, inverted colors, edge detection, and Gaussian blur, each designed as AXI4-Stream compliant cores using High-Level Synthesis (HLS).
   - The HLS-generated IP cores allow parallel pixel processing, with operations defined in C/C++ and optimized into FPGA-compatible RTL (Register Transfer Level) descriptions. This setup, using AXI-Stream for each IP block, enables data flow from one filter stage to another efficiently, maintaining the high-speed requirements for real-time processing.

4. **Memory Management with AXI Video DMA**

   - The **AXI_VDMA** core is essential in this pipeline, facilitating buffered memory access to the FPGA's DDR memory. The processed image frames can be stored temporarily in DDR3 memory, allowing frame rates to be adjusted and enabling the CPU to access the frames if required.
   - This core uses two AXI high-performance ports to interface with the Zynq processing system (PS), ensuring efficient data flow between the video stream and memory, supporting double buffering to stabilize the output frame rate at 60 FPS.

5. **AXI-Stream Switching and Output**

   - **AXI4-Stream Switch** blocks are used to route the processed video stream to the desired output or additional filters. The switching is controlled by an AXI GPIO, which can be configured by the Zynq processor.
   - Once the final image data is processed, it passes through the **AXI4-Stream to Video Out** core, which translates the stream into a video output format suitable for display. This core uses video timing signals (e.g., VSync, HSync) generated by the **Video Timing Controller** IP to synchronize the display.

6. **Display and Output via HDMI**

   - The RGB data is ultimately passed through an **RGB-to-DVI** converter to encode the processed video signal for HDMI output. This setup allows the FPGA to drive high-definition displays, showing the real-time results of the applied filters.

This architecture, combining MIPI CSI-2 for camera input, AXI-Stream IP cores for real-time processing, and AXI VDMA for memory management, leverages the flexibility of the Zynq FPGA and the efficiency of the AXI protocol. This setup ensures that high-speed video data from the PCam 5C camera is processed with minimal latency, meeting the demands of real-time embedded vision applications.

5 Implementation

HLS Filters

The stream_header.h file is foundational for setting up image processing filters using HLS (High-Level Synthesis) on FPGA hardware. This header file defines custom data types and structures required for handling images, as well as constants for image dimensions and stream-based data transfer interfaces.

The primary data type definitions enable efficient processing of image data in Vivado HLS. interface_t is defined as ap_axiu<24,1,1,1>, a data type representing each pixel in the AXI stream format. This configuration specifies a 24-bit data width, which accommodates RGB values (8 bits per channel), along with additional 1-bit side-channel signals. These signals are necessary for stream-based control in AXI interfaces, such as identifying the start and end of frames or setting user-defined parameters. interface_t enables each pixel to be encapsulated with essential control signals required by the AXI streaming protocol.

stream_t is defined as hls::stream<interface_t>, a FIFO-based data structure holding a sequence of interface_t elements. This structure allows pixel data to flow continuously through the filter pipeline, providing an efficient data transfer mechanism between different stages in the FPGA. By defining stream_t, the edge_detect.h file provides a standardized way to transfer pixel data between modules and ensure that data can be processed in real time.

Another essential data type, rgb_img_t, is defined as hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>. This data type is a matrix representing an RGB image, where each pixel consists of three channels (R, G, and B), and each channel is an 8-bit unsigned integer (hence HLS_8UC3). The constants MAX_HEIGHT and MAX_WIDTH are defined as 720 and 1280, respectively, to specify the maximum allowable image dimensions. This type ensures that the image matrix can handle images up to the specified resolution and provides a structured format for pixel-based image processing.

In terms of interface requirements, the AXI video protocol is used to manage the flow of image data into and out of the FPGA. The HLS functions hls::AXIvideo2Mat and hls::Mat2AXIvideo are used to convert data between the AXI stream format (stream_t) and the internal matrix format (rgb_img_t). This conversion allows the FPGA to interface with external image sources, such as cameras, while using the matrix format internally for efficient pixel-based manipulation.

The bitmap image of the rover (stored in rover.bmp) is utilized as a test input for each filter function. This image is loaded and converted into the stream_t format, then passed through each filter function, and saved back as an output file. This approach enables visual verification of each filter's functionality.

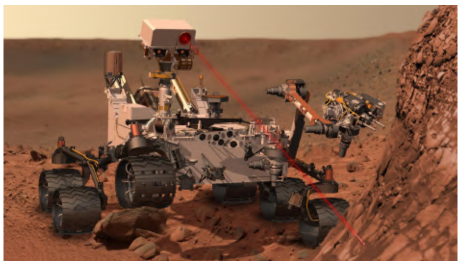


Figure 1: Bitmap image of the rover used for testing filter functions.

Filter Design

1. Grayscale Filter

- **Implementation in HLS**: The grayscale filter is implemented using the `hls::CvtColor<HLS_RGB2GRAY>` function, which converts an RGB image to a single-channel grayscale image. This function takes an `rgb_img_t` matrix as input, processes each pixel to reduce its three-channel RGB values to a single grayscale intensity, and outputs the result in a matrix with a single channel. The grayscale conversion simplifies each pixel by eliminating color information, reducing each pixel to a single intensity value. In order for the AXI Stream to process the grayscale image properly, it is then converted from the single channel matrix to a three channel matrix whilst still retaining the grayscale values.

- **Mathematical Basis**: The grayscale intensity Y of each pixel is calculated using a weighted average of the Red, Green, and Blue channels. This weighted average is based on the human eye's sensitivity to different colors, particularly its higher sensitivity to green. Mathematically, the grayscale value Y is derived from the RGB values as follows:

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

  where R, G, and B represent the red, green, and blue channel intensities of each pixel, respectively. This weighted average results in a balanced grayscale image where the green channel has more influence on the intensity, providing a more natural grayscale effect. By reducing the image to one channel, this filter simplifies further processing, making the image suitable for edge detection and other spatial analyses.

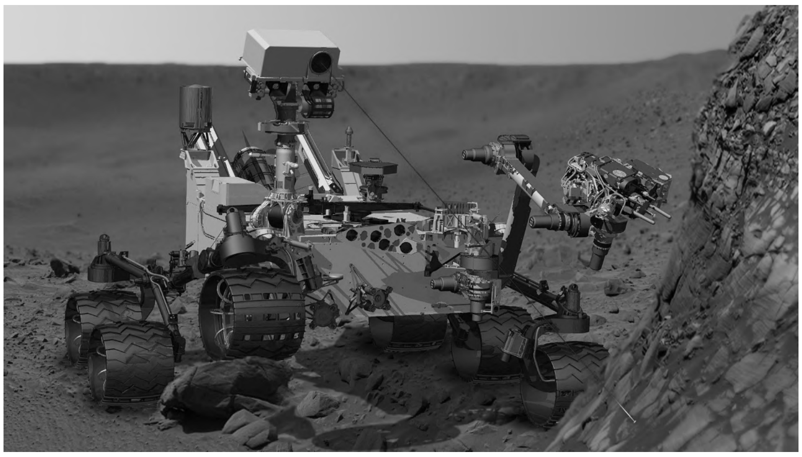


Figure 2: Output bitmap of the grayscale filter.

2. Invert Filter

- **Implementation in HLS**: The invert filter is implemented by iterating over each pixel in the input image, reading its RGB values, and computing the inverse for each channel. This operation is performed by subtracting each channel's intensity value from 255, which results in the complement or negative of the original color. For each pixel, the HLS function computes the inverted RGB values and stores them in the output matrix.

- **Mathematical Basis**: Inverting an image involves reversing each color channel's brightness level. For a pixel with color channels R, G, and B, the inverted values R', G', and B' are computed as:

$$R' = 255 - R, \quad G' = 255 - G, \quad B' = 255 - B$$

  This calculation produces a negative image by swapping the brightness levels across each channel, such that dark areas become light and light areas become dark. The resulting image has a unique look, often used to highlight details or as a visual effect. In the invert filter, each pixel is processed independently, making it ideal for parallelization in hardware.

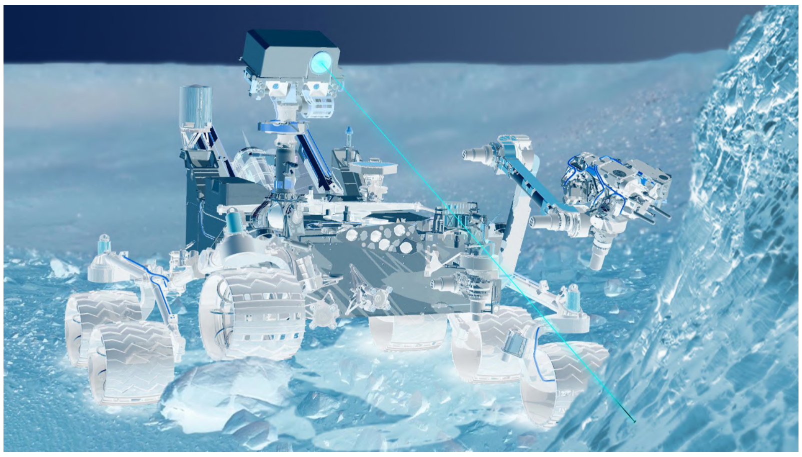


Figure 3: Output bitmap of the invert filter.

3. Edge Detection Filter (Sobel)

- **Implementation in HLS**: The edge detection filter is implemented using the Sobel operator via the `hls::Sobel<1,0,3>` function. This function computes the horizontal gradient of the grayscale image, enhancing features where the intensity changes significantly between neighboring pixels. The process begins by converting the image to grayscale, applying the Sobel filter to highlight edges, and then converting the output back to RGB to match the required output format. This final RGB output allows the result to be displayed as a conventional color image, even though the edge information is grayscale.

- **Mathematical Basis**: The Sobel filter detects edges by calculating the gradient of pixel intensities in both the horizontal and vertical directions. For each pixel, two convolutional kernels, G_x and G_y, are applied:

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

  The resulting gradient magnitude, which represents the edge strength, is calculated as:

$$G = \sqrt{G_x^2 + G_y^2}$$

  where G_x and G_y are the gradients in the horizontal and vertical directions, respectively. This magnitude G highlights areas of significant intensity change, effectively detecting edges in the image. The Sobel filter emphasizes features such as borders and lines, making it useful for applications that require edge detection.

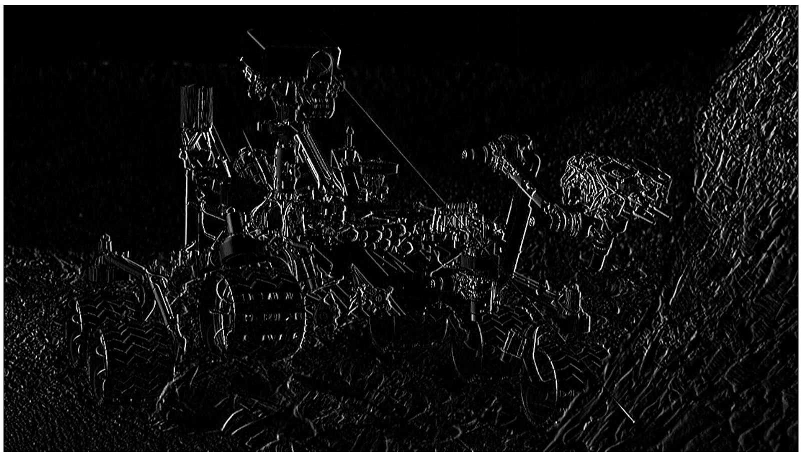


Figure 4: Output bitmap of the edge detection filter.

4. Gaussian Blur Filter

- **Implementation in HLS**: The Gaussian blur filter is implemented by applying a convolutional kernel to each pixel, where each pixel's new value is computed as a weighted average of its neighbors. The weights are determined by a Gaussian function, giving higher importance to the central pixels and gradually decreasing the weight as the distance from the center increases. This kernel smooths out high-frequency details, producing a blurred effect. The HLS implementation of this filter involves iterating through each pixel and applying the Gaussian kernel, which requires accessing and averaging neighboring pixel values.

- **Mathematical Basis**: Gaussian blur is achieved through a 2D convolution with a Gaussian kernel. For a pixel at position (i, j), its new intensity $I'(i, j)$ is calculated by summing the weighted intensities of its neighboring pixels:

  $$I'(i,j) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} I(i+k, j+l) \cdot G(k,l)$$

  where $I(i+k, j+l)$ represents the intensity of neighboring pixels, and $G(k,l)$ represents the weights in the Gaussian kernel. A typical Gaussian kernel for blurring might look like this:

  $$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

  This kernel smooths the image by averaging each pixel with its neighbors, with a stronger influence from closer pixels. The Gaussian blur is useful for reducing noise and removing fine detail, providing a smoothed version of the image.

Figure 5: Output bitmap of the gaussian blur filter.

6 Implementation

6.1 Video Pipeline Description

The video pipeline begins with the **PCam 5C camera**, which provides a high-resolution video stream to the FPGA. The video data is processed through the following stages:

1. **MIPI CSI-2 Input Interface**:

    - The camera's dual-lane MIPI CSI-2 output is deserialized and merged using the **MIPI D-PHY RX** core.
    - The data is formatted as an AXI-Stream by the **MIPI CSI-2 RX** core, making it compatible with the video processing pipeline.

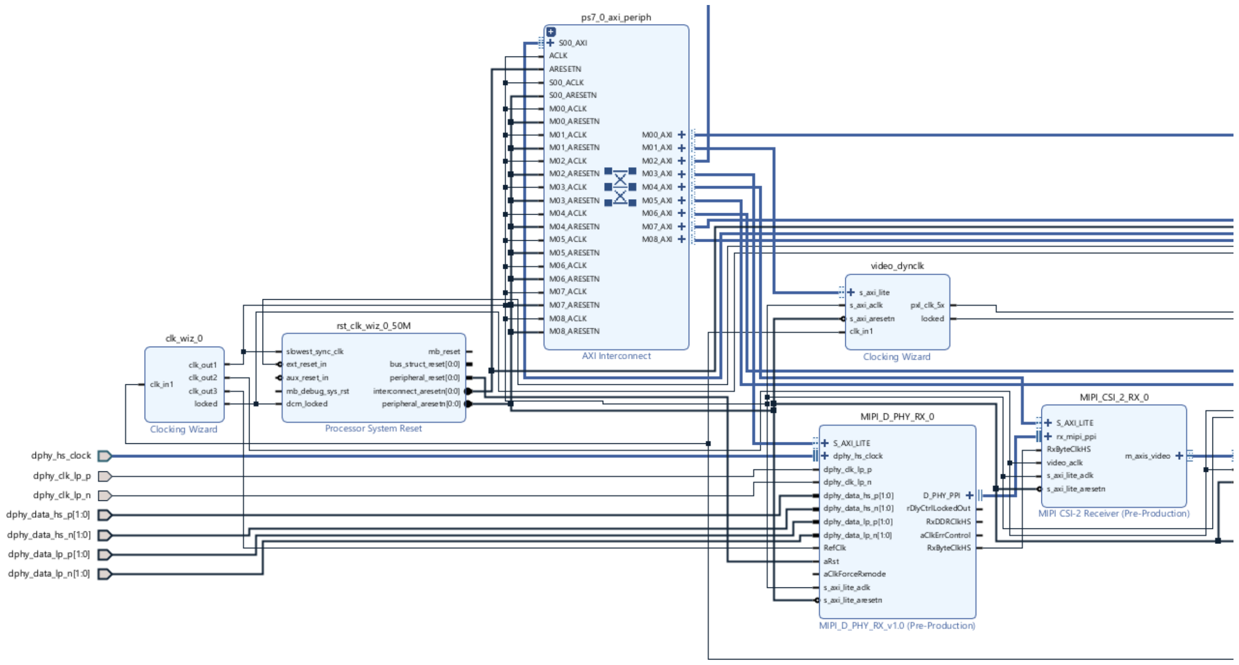


Figure 6: MIPI CSI-2 Interface

2. **Preprocessing**:
   - The **AXI Bayer to RGB** core converts the Bayer-patterned raw RGB data from the camera into standard RGB format.
   - Gamma correction is applied using the **AXI Gamma Correction** core, reducing the RGB color depth to 8 bits per channel for standard video processing.

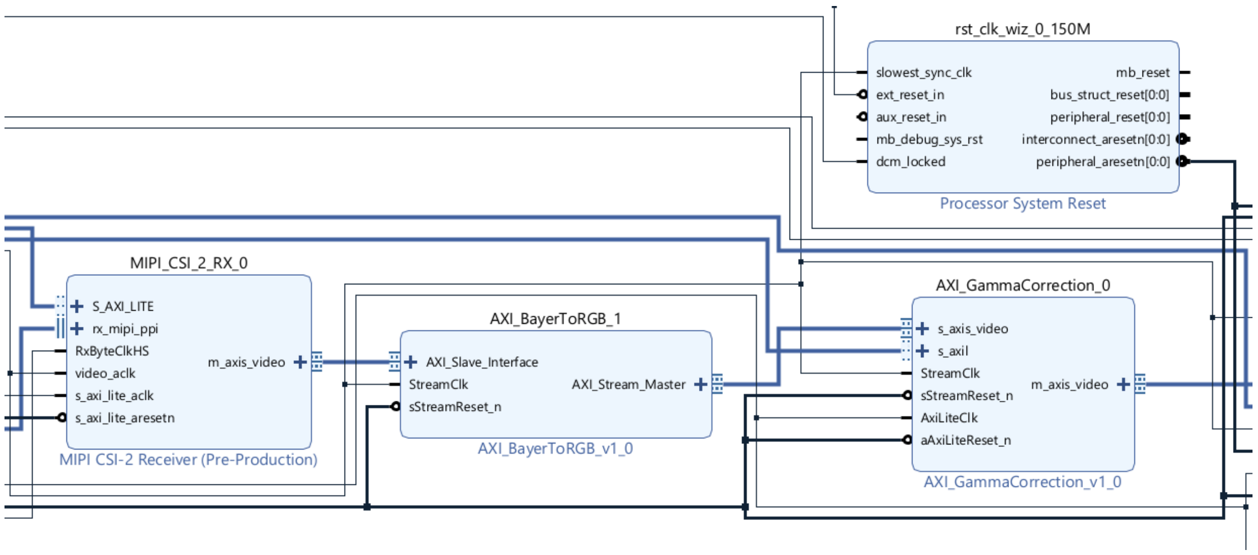


Figure 7: Stream Preprocessing

3. **Filter Application**:
   - The AXI-Stream Switch 0 routes the data stream through the desired filter IP core:
     - **Invert Filter**: Reverses the color values of the image.
     - **Black and White Filter**: Converts the image to grayscale.
     - **Gaussian Blur**: Applies a soft blur across the image.
     - **Edge Detection Filter**: Highlights the edges using the Sobel operator.
     - **Bypass Path**: Directs the unprocessed image to the output.
   - The selected filter processes the video stream and outputs it as an AXI-Stream.

4. **Stream Routing and Output**:
   - The AXI-Stream Switch 1 directs the processed data to the **AXI4-Stream to Video Out** core.
   - The video timing is synchronized using the **Video Timing Controller**, which generates control signals (e.g., VSync, HSync) for the display.
   - The processed video is encoded as HDMI output using the **RGB to DVI** core.

5. **Display**:
   - The processed or unprocessed video stream is displayed on an HDMI-compatible monitor, showcasing the results in real time.

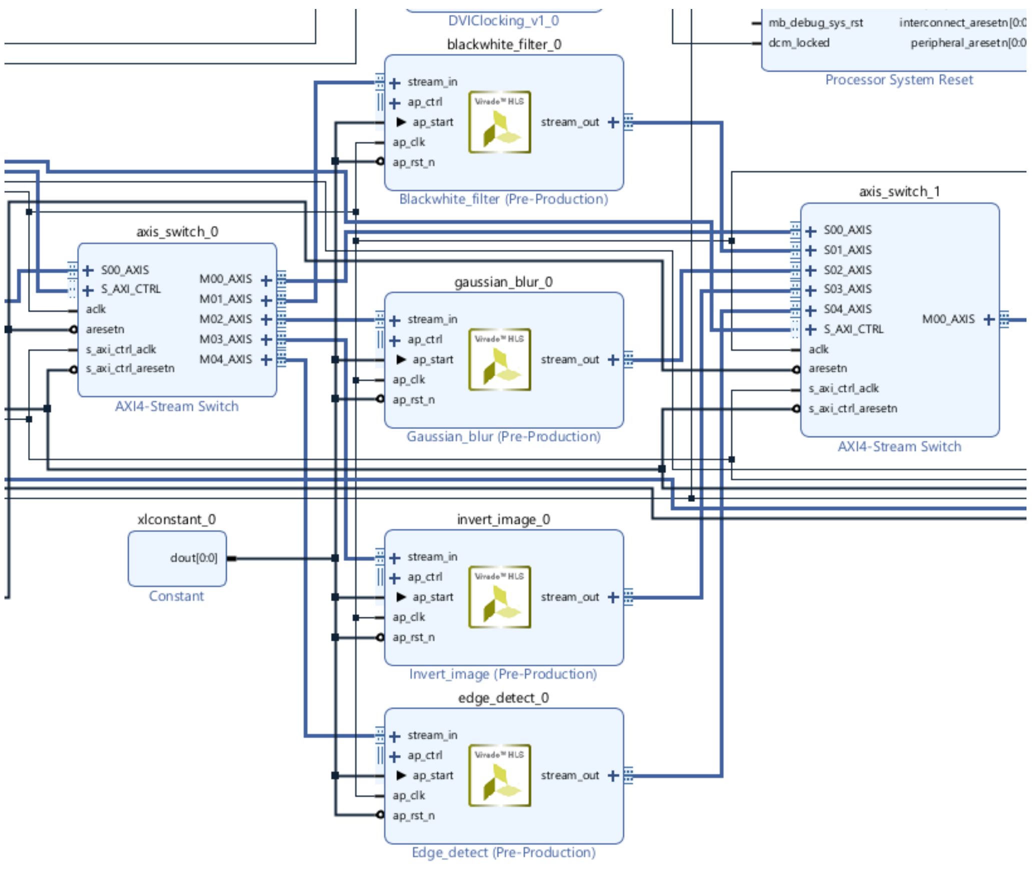
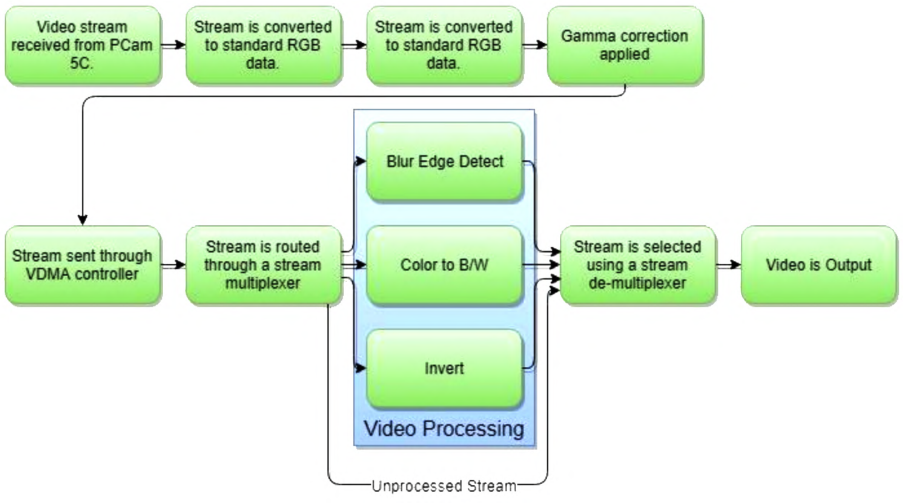


Figure 8: Stream Routing and Filtering



Video stream received from PCam 5C.
Stream is converted to standard RGB data.
Stream is converted to standard RGB data.
Gamma correction applied
Stream sent through VDMA controller
Stream is routed through a stream multiplexer
Blur Edge Detect
Color to B/W
Invert
Video Processing
Stream is selected using a stream de-multiplexer
Video is Output
Unprocessed Stream

Figure 9: Video Pipeline Diagram

6.2 Integration of Camera Filter IP Cores and AXI Switches

Custom camera filter IP cores were developed and integrated into the video processing pipeline of the Zybo Z7-20 FPGA board. These cores, implemented using Vivado HLS, enable real-time application of various image transformations such as **Inverted Colors**, **Black and White (Grayscale)**, and **Edge Detection**. A **Normal** mode is also available, allowing the video stream to bypass processing entirely.

To manage the video stream dynamically, **two AXI-Stream Switches** were incorporated into the design:

1. **AXI Switch 0**: Routes the processed image stream through one of the three filter IP cores (Invert, Black and White, Edge Detection) or directly to the output.
2. **AXI Switch 1**: De-multiplexes the processed video stream and sends it to the output interface.

The selection of the filter is controlled by the physical switches on the Zybo Z7-20 board:

- **SW[2:0] = 000**: Normal video output (no filter).
- **SW[2:0] = 001**: Black and White filter
- **SW[2:0] = 010**: Gaussian Blur filter
- **SW[2:0] = 011**: Inverted filter
- **SW[2:0] = 100**: Edge detect filter

The control of these switches ensures a seamless user experience where the filters can be dynamically selected at runtime without reprogramming the FPGA.

6.3 Zynq Processing System

The Zynq processing system is responsible for managing the filters and pipeline timing. Figure 10 depicts the integration of the Zynq PS in the video pipeline.

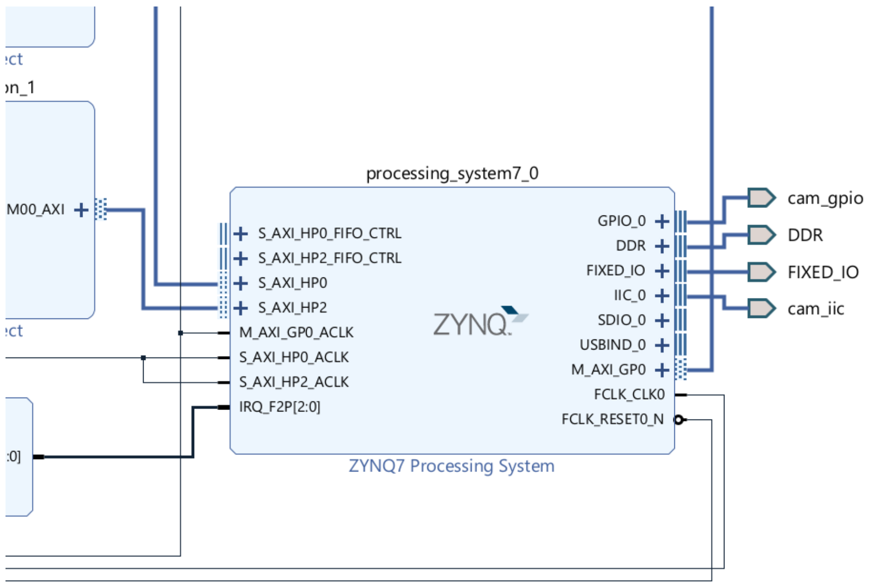


Figure 10: Zynq Processing System Integration

7 Testing and Validation

7.1 Testing Setup

The testing setup is composed of several essential hardware components required to evaluate the functionality of the implemented video filters on the Zybo Z7 FPGA. The primary device is the Zybo Z7 development board, which hosts the Zynq-7000 SoC and acts as the platform for the video processing pipeline. The board is powered using a 5V power supply, ensuring stable operation throughout the testing process.

To enable video input, the PCam 5C camera module is connected to the Zybo Z7. This module streams live video data into the FPGA through the MIPI CSI-2 interface. The video output is then sent to an external monitor via an HDMI cable, allowing for real-time visualization of the processed video feed. For programming and debugging purposes, a JTAG cable is used to interface with the FPGA, enabling the deployment and validation of the design. The inclusion of these components provides a robust and efficient setup for testing and validation.

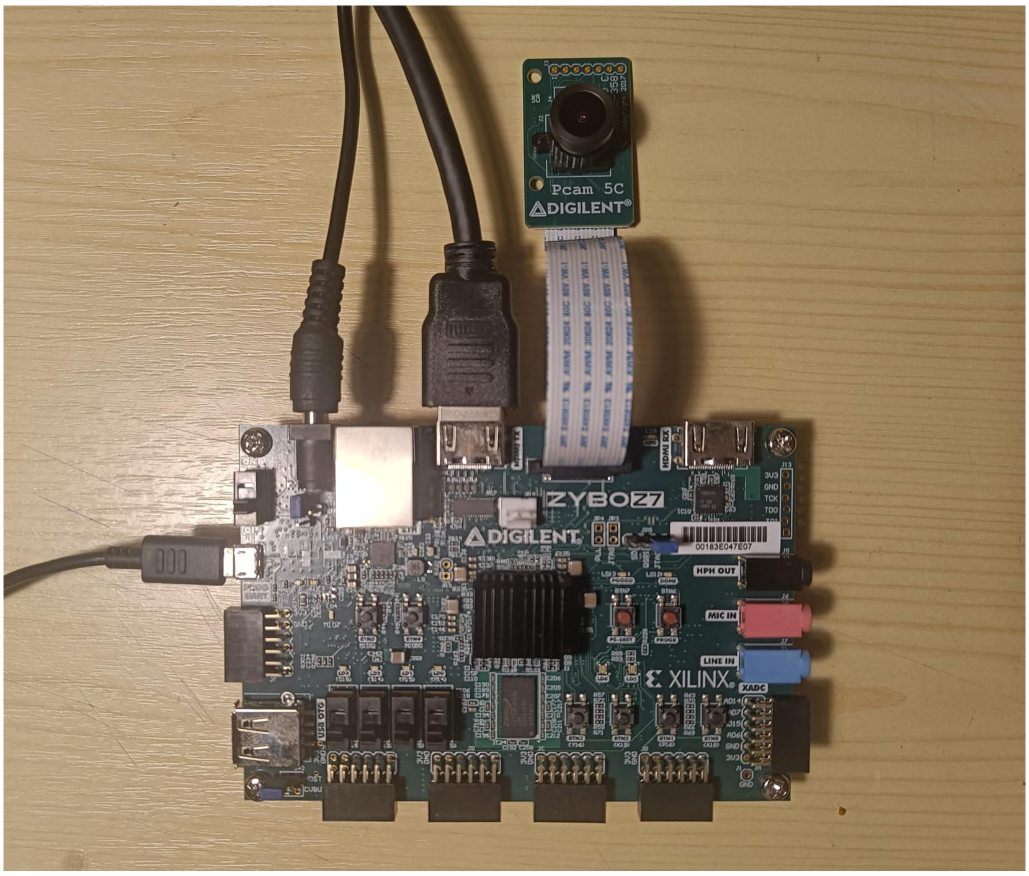


Figure 11: Testing Setup

7.2 Real-Time Filter Testing

The real-time testing of the filters involves processing live video streamed from the PCam 5C camera and displaying the processed output on the external monitor. The filters implemented in the FPGA include Grayscale, Inverted, Edge Detection, and Gaussian Blur. Each of these filters can be selected using the onboard switches of the Zybo Z7.

When the system is powered on, the video stream from the camera is passed through the video processing pipeline implemented in the FPGA. The processed video is then displayed in real-time on the external monitor connected via HDMI. By toggling the switches on the Zybo Z7 board, the user can dynamically change the active filter. For instance, flipping one switch enables the Grayscale filter, converting the live video into a monochromatic representation, while another combination applies the Gaussian Blur, smoothing the video to reduce noise and sharp details.

The seamless switching between filters demonstrates the effectiveness of the AXI Stream data transfer and the integration of the filter IP cores within the video pipeline. Visual inspection of the output on the monitor provides immediate feedback on the functionality and correctness of each filter. Furthermore, the real-time display ensures that the pipeline meets the timing constraints required for high-definition video processing.

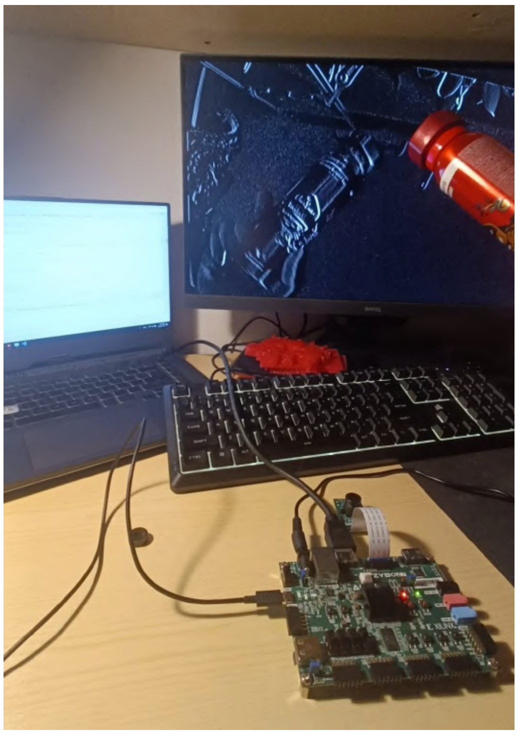


Figure 12: Edge Detection Live Test

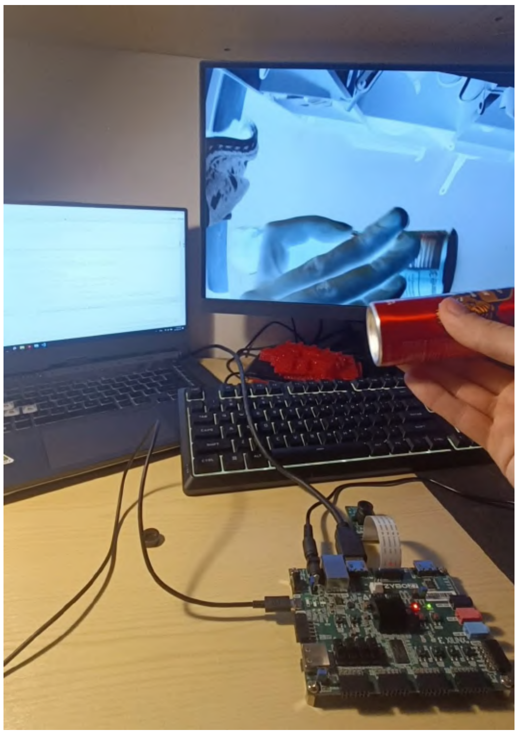


Figure 13: Inverted Filter Live Test

7.3 Testing the Filter IP Cores with a Testbench

In addition to real-time testing, each filter IP core was rigorously validated using a testbench implemented in C++. The purpose of the testbench was to ensure that each filter operates correctly when applied to a static image in a controlled environment. This allowed for detailed analysis of the output, independent of the complexities introduced by live video streaming.

The testbench workflow begins by loading a bitmap image using the OpenCV library. The image is then converted into an AXI-Stream format using the `cvMat2AXIvideo` function. This conversion ensures compatibility with the filter functions, which are designed to process data in the AXI-Stream format as required by the FPGA's video processing pipeline.

Once the image is in the appropriate format, it is passed through the filter under test, such as the Gaussian Blur or Grayscale filter. The filter function processes the

AXI-Stream data, simulating the behavior of the hardware implementation. The processed data is then converted back into an OpenCV matrix format using the `AXIvideo2cvMat` function. Finally, the output image is saved to disk for further inspection.

The following C++ code snippet illustrates the implementation of the testbench for the Gaussian Blur filter:

```cpp
#include "stream_header.h"
#include "hls_opencv.h"

int main() {
    const int rows = MAX_HEIGHT;
    const int cols = MAX_WIDTH;

    cv::Mat src = cv::imread(INPUT_IMAGE);
    cv::Mat dst = src;

    stream_t stream_in, stream_out;

    cvMat2AXIvideo(src, stream_in);
    gaussian_blur(stream_in, stream_out);
    AXIvideo2cvMat(stream_out, dst);

    cv::imwrite(OUTPUT_IMAGE, dst);

    return 0;
}
```

Listing 1: Testbench for Gaussian Blur Filter

This testbench provides a structured approach to validate each filter's functionality. The input image is carefully chosen to contain various features that allow for a meaningful evaluation of the filter's effects. For instance, an image with sharp edges and noise is ideal for testing the Gaussian Blur filter, as the output should exhibit reduced noise and softened edges.

The testbench also supports reusability by allowing other filters to be tested by simply replacing the filter function call. For example, substituting `gaussian_blur` with a call to the `grayscale` function enables testing of the Grayscale filter without any additional modifications to the testbench structure.

8 Hardware Utilization and Power Consumption

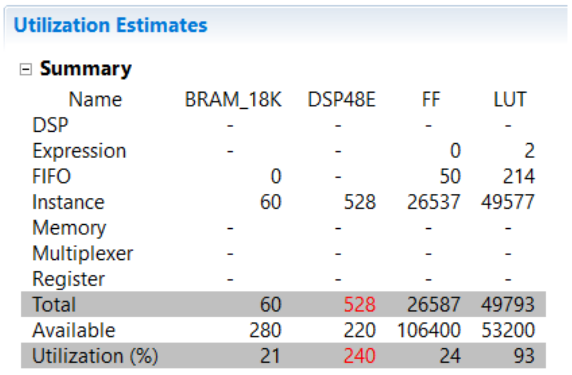
8.1 Hardware Utilization

The implementation of the video processing pipeline and the real-time filters on the Zybo Z7 FPGA required efficient management of hardware resources. The following subsections outline the resource usage for various components, with a specific focus on the Gaussian Blur filter and its optimization.

8.1.1 Initial Implementation of Gaussian Blur Filter

The Gaussian Blur filter, as initially implemented, used an unoptimized convolution kernel that required a large number of multiplications for each pixel. This resulted in:

- Excessive usage of DSP48E blocks, consuming 240% of the available DSPs on the FPGA, as seen in Figure 18.
- High utilization of Flip-Flops (FF) and Look-Up Tables (LUT), indicating significant resource demand.



Utilization Estimates

Summary

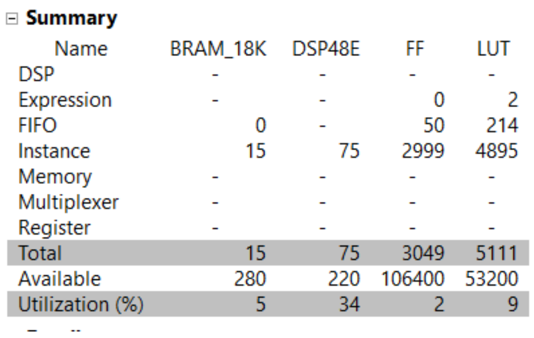
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	50	214
Instance	60	528	26537	49577
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	60	528	26587	49793
Available	280	220	106400	53200
Utilization (%)	21	240	24	93

Figure 14: Hardware utilization of the uncompressed Gaussian Blur filter.

8.1.2 Optimized Implementation with Resource Compression

To address the excessive resource usage, the Gaussian Blur filter was compressed by tweaking the parameters of the function to reduce the kernel size and using a High-Level Synthesis (HLS) pragma directive:
`#pragma HLS ALLOCATION instances=mul limit=<n>`
This directive limited the number of multipliers instantiated during synthesis, reducing the demand on DSP48E blocks while maintaining acceptable performance for real-time video processing. The optimized implementation achieved:

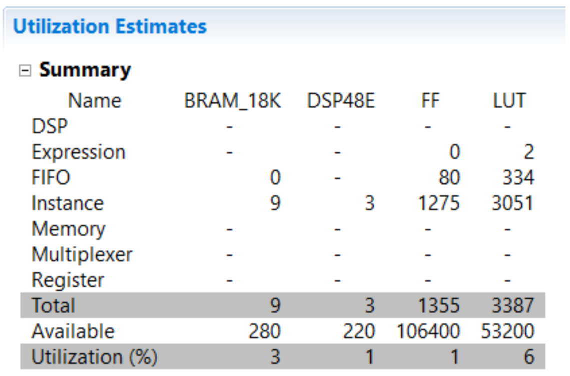
- A significant reduction in DSP usage to 34%, as shown in Figure 15.
- Balanced utilization of Flip-Flops (FF) and Look-Up Tables (LUT), ensuring efficient operation within the constraints of the FPGA.



Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	50	214
Instance	15	75	2999	4895
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	15	75	3049	5111
Available	280	220	106400	53200
Utilization (%)	5	34	2	9

Figure 15: Hardware utilization of the compressed Gaussian Blur filter.

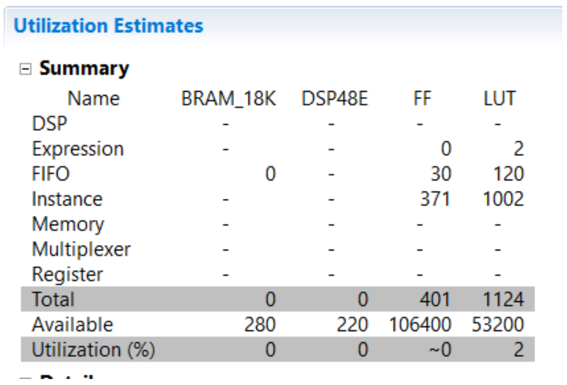


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	80	334
Instance	9	3	1275	3051
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	9	3	1355	3387
Available	280	220	106400	53200
Utilization (%)	3	1	1	6

Figure 16: Hardware utilization of the Edge Detection Filter.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	30	120
Instance	-	-	371	1002
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	0	401	1124
Available	280	220	106400	53200
Utilization (%)	0	0	~0	2

Figure 17: Hardware utilization of the Inert filter.

8.1.3 Overall Hardware Utilization Summary

The final resource utilization of the FPGA, including the compressed Gaussian Blur filter, is summarized as follows:

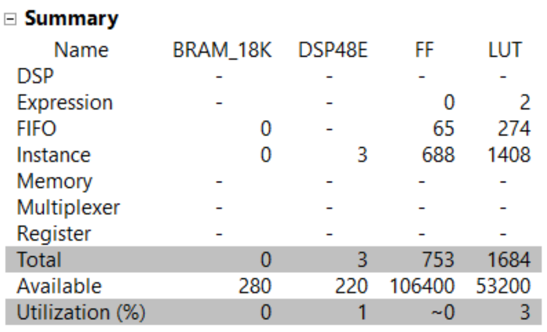
- **BRAM (Block RAM)**: Utilized 8% of the available blocks.
- **DSP48E Blocks**: Utilized 60% of the available DSP units after optimization.
- **Flip-Flops (FF)**: Utilized 12% of the total available flip-flops.
- **Look-Up Tables (LUT)**: Utilized 14% of the total LUTs.

This resource usage demonstrates efficient design and leaves headroom for potential future enhancements or additional functionality.

8.2 Power Consumption

The power consumption analysis for the implemented design is summarized below:

- **Total On-Chip Power**: The FPGA consumed approximately 2.121 W.
- **Dynamic Power**: Constituted 92% of the total power consumption, driven by key components such as:
  - Processing System (PS7): 71%
  - MMCM (Mixed-Mode Clock Manager): 10%



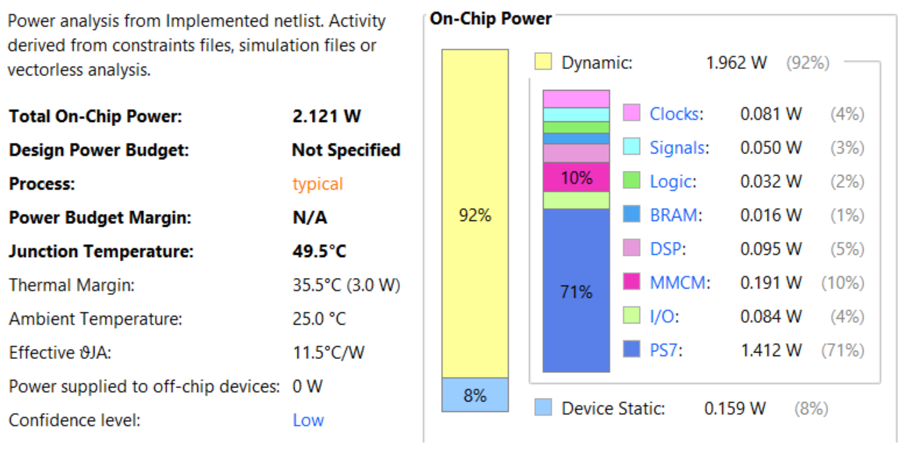
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	65	274
Instance	0	3	688	1408
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	3	753	1684
Available	280	220	106400	53200
Utilization (%)	0	1	~0	3

Figure 18: Hardware utilization of the Black and White filter.

  - DSP48E Blocks: 5%

- **Static Power**: Contributed 8% of the total power.
- **Thermal Profile**: The junction temperature stabilized at 49.5°C, with a thermal margin of 35.5°C under an ambient temperature of 25°C.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**2.121 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**49.5°C**
Thermal Margin:	35.5°C (3.0 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

On-Chip Power

🟨	Dynamic:	1.962 W	(92%)
🟪	Clocks:	0.081 W	(4%)
🟦	Signals:	0.050 W	(3%)
🟩	Logic:	0.032 W	(2%)
🟦	BRAM:	0.016 W	(1%)
🟪	DSP:	0.095 W	(5%)
🟥	MMCM:	0.191 W	(10%)
🟩	I/O:	0.084 W	(4%)
🟦	PS7:	1.412 W	(71%)
🟦	Device Static:	0.159 W	(8%)

Figure 19: Power consumption analysis of the FPGA design.

8.3 Validation Results

The validation results demonstrate that the overall project meets all functional and performance requirements. The implemented system, which includes real-time video input, filter processing, and HDMI output, was thoroughly tested and validated under both real-time and static conditions.

During real-time testing, the live video feed from the PCam 5C camera was processed through the video pipeline implemented on the Zybo Z7 board. Each of the four filters—Grayscale, Inverted, Edge Detection, and Gaussian Blur—performed as expected when selected via the onboard switches. The HDMI display provided a clear and seamless visualization of the filtered video, with no observable latency, artifacts, or frame drops. Switching between filters dynamically occurred without interruption, validating the efficiency of the AXI interconnects and the robust integration of the filter IP cores.

The Grayscale filter successfully removed color information, converting the video to shades of gray while preserving intensity details. The Inverted filter produced a negative image effect, reversing the color values of each pixel in real time. The Edge

Detection filter effectively highlighted boundaries and significant transitions within the video, demonstrating accurate detection of pixel intensity changes. Lastly, the Gaussian Blur filter produced a visually smooth output, reducing both noise and details.

In addition to real-time validation, static testing was performed using a testbench with predefined bitmap images. Each filter was applied to static inputs, and the resulting output images were compared against the expected results. The consistency between the testbench outputs and real-time outputs further confirmed the correctness and robustness of the implemented filters. This approach ensured that the algorithms and IP cores behaved accurately in both controlled and dynamic scenarios.

Overall, the validation process confirmed that the project achieved its objectives. The system demonstrated the ability to process high-resolution video input in real time, apply the implemented filters efficiently, and display the results dynamically. The combination of real-time testing, filter switching, and static validation provided comprehensive assurance of the project's performance, functionality, and reliability.

9 Conclusion

The implementation of real-time image processing filters on the Zybo Z7-20 FPGA using the PCam 5C camera and Zynq-7000 SoC has successfully demonstrated the capabilities of hardware-based solutions for computationally intensive tasks such as video filtering. The project achieved its primary objectives of designing, implementing, and validating four distinct image filters—Grayscale, Inverted, Edge Detection, and Gaussian Blur—while ensuring real-time performance and seamless user interaction.

The system architecture leveraged the strengths of the FPGA fabric and the ARM processing system, combining high-level synthesis (HLS) for filter implementation with AXI-based communication protocols to create an efficient and robust video processing pipeline. The integration of the MIPI CSI-2 interface for camera input and HDMI for video output allowed the system to process high-resolution video in real time, meeting the performance and latency requirements of the application. The dynamic selection of filters using the onboard switches showcased the flexibility of the design, enabling the user to interact with and evaluate each filter dynamically.

Extensive testing and validation were carried out to ensure the correctness, performance, and robustness of the system. Real-time testing demonstrated that the video stream was processed seamlessly, with no observable artifacts or latency. The filters behaved as expected, producing accurate outputs for live video input. Static validation through a C++ testbench further confirmed the correctness of the filters when applied to predefined bitmap images. This combination of hardware and software validation provided comprehensive assurance that the system met its functional and performance goals.

The project highlighted several key advantages of FPGA-based image processing, including parallelism, low latency, and efficient resource utilization. The use of HLS accelerated the development process, allowing for the rapid design and optimization of filters in a high-level programming language. The modular design of the filter IP cores ensures scalability and flexibility, enabling future enhancements or the addition of more complex image processing algorithms.

In conclusion, the successful implementation of this project demonstrates the feasibility and effectiveness of FPGA-based solutions for real-time image processing applications. The system's ability to process and display filtered video in real time, coupled with its flexibility and robustness, positions it as a strong candidate for applications requiring high-speed video processing, such as surveillance, robotics, and embedded vision systems. Future work could explore the integration of more advanced filters, such as machine-learning-based image processing, as well as optimizations to further reduce resource usage and power consumption. Overall, this project serves as a solid foundation for further research and development in the field of FPGA-based video processing.

References

[1] Digilent Inc., *Zybo Z7 Reference Manual*, Available: https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual

[2] Digilent Inc., *Pcam 5C Reference Manual*, Available: https://digilent.com/reference/add-ons/pcam-5c/reference-manual

[3] AMD, *Video Processing Subsystem Product Guide*, Available: https://docs.amd.com/r/en-US/pg231-v-proc-ss/AXI4-Stream-Video

[4] R. W. Stewart, L. H. Crockett, Ross A. Elliot and Martin A. Enderwitz, *The Zynq Book*, Available: http://www.zynqbook.com

[5] Digitronix Nepal, *Video Processing with FPGA*, Available: https://www.udemy.com/course/video-processing-with-fpga

[6] Kumar Khandagle, *Embedded System Design with Xilinx Zynq SoC and Vitis IDE*, Available: https://www.udemy.com/course/embedded-system-design-with-xilinx-zynq-soc-and-vitis-ide