Hardware Acceleration of Sobel Edge Detection for Feature Detection in Autonomous Vehicles

By:

Hassan Ahmad,
Aizah Ahmed,
Tyler Belluomini

Table of Contents

List of Figures

1 Introduction

In this project, we investigated the implementation and evaluation of Sobel edge detection on the ZedBoard FPGA, comparing a purely software-based method to a hardware-accelerated approach. Edge detection is a key procedure in image processing, playing a critical role in applications such as autonomous navigation and computer vision by delineating object boundaries. In this group project, we focused on implementing and evaluating the Sobel edge detection algorithm on the ZedBoard FPGA, examining the trade-offs and benefits of a purely software-based approach compared to a hardware accelerated solution.

To establish a baseline, we first ran the Sobel algorithm entirely in software on the ARM Cortex-A9 processor within the Zynq-7000 system on chip, measuring throughput, energy consumption, and resource utilization. These metrics served as a point of comparison when we moved to hardware acceleration. In the second phase, we used Vivado HLS 2016.3 to implement the most computationally intensive parts of the Sobel algorithm as custom IP blocks in the FPGA's programmable logic. We applied hardware specific optimizations, such as pipelining and loop unrolling, and employed AXI4-Stream interfaces for efficient data transfer between the processing system and the programmable logic.

This report provides a comprehensive analysis of both approaches, supported by detailed block diagrams, timing assessments, and power measurements. By comparing software only and hardware accelerated implementations of Sobel edge detection on the ZedBoard FPGA, we demonstrate how offloading computation to dedicated hardware can yield significant gains in performance and energy efficiency, informing future decisions on implementing real time image processing on FPGA platforms.

2 Background and Literature Review

Edge detection is a key process in image processing, especially in applications like autonomous navigation where detecting object boundaries and obstacles is crucial. The Sobel algorithm, introduced as a 3x3 isotropic gradient operator, remains an important method for highlighting areas of rapid intensity change in grayscale images [2]. However, the computational demands, especially at high resolutions, challenge real time performance in software only implementations [2]. Researchers have explored various strategies to optimize the Sobel algorithm. Enhancements in gradient calculation aim to preserve detection accuracy while reducing computational load. Some studies have refined threshold parameters using techniques such as clustering and fuzzy logic, resulting in a more robust detector suited to diverse imaging conditions [3].

Additionally, hardware acceleration has also emerged as an effective approach to address limitations in software-based edge detection. FPGA based designs leverage parallel processing capabilities to substantially increase speed and efficiency. Previous work on FPGA implementations of feature detection algorithms demonstrated the feasibility of achieving real time performance by integrating advanced processing elements and programmable logic. This provides the adaptability that dynamic autonomous systems demand [4].

Furthermore, hardware software co design approach can further improve real time edge detection by identifying tasks that consume significant processing power, such as convolution operations, and shifting them into dedicated hardware. Studies have achieved ultra-low latency and improved energy efficiency, aligning with strict requirements in autonomous vehicle applications [1]. Additionally, efficient convolution operations also play an essential role in machine learning, where similar kernels form the foundation of convolutional neural networks. The optimization of these kernels underscores their significance not only for feature extraction but also for bridging traditional edge detection methods with modern artificial intelligence applications [5].

In essence, the literature emphasizes that hardware acceleration combined with algorithmic refinements can substantially improve Sobel edge detection for real time applications. These findings support the methodologies employed in the project, including software only and hardware accelerated implementations on the Zynq 7000 platform for autonomous vehicle scenarios. Techniques such as pipelining and loop unrolling have the potential to yield significant gains in processing speed and energy efficiency [3].

3 Assumptions and Constraints

Our implementation assumes a stable and controlled operating environment with consistent power supply and clock signals to ensure reliable functionality. Input images are expected to be raw RGB format with fixed dimensions, allowing for predictable computational loads and algorithmic behavior. The Zynq-7000 platform introduces resource constraints, including limited programmable logic, memory capacity, and processing power, which require careful resource allocation and optimization strategies.

Moreover, real-time performance requirements impose additional constraints, requiring low-latency data transfers and efficient scheduling of tasks to meet stringent timing deadlines. The design also assumes adherence to standard AXI interfaces and reliable communication protocols to facilitate seamless data exchanges between the processing system and programmable logic. These constraints emphasize the importance of system-level optimizations to achieve the desired balance of performance, energy efficiency, and functional accuracy within the capabilities of the FPGA platform.

4 Methodology

4.1 High-Level Overview

Both the software-only and hardware-accelerated implementations included a shared feature for managing video output, using a multiplexer (mux_v1_0) to control the flow of data between raw and processed video streams. The multiplexer determined whether the raw input or the processed output was directed to the VGA interface. Additionally, slices (xlslice_0, xlslice_1, and xlslice_2) were used to separate the video data into RGB color channels, which were subsequently sent to the VGA output pins. This configuration ensured accurate and flexible video output handling for both implementations, enabling seamless visualization of the results on a display. The integration of these components is depicted in Figure 4, which shows the connections to the VGA interface.

The software-only implementation relied entirely on the ARM Cortex-A9 processor to process a single static image, as shown in Figure 2. Meanwhile, the hardware-accelerated design, depicted in Figure 3, introduced a custom Sobel IP block into the FPGA's programmable logic, processing live video streams from a webcam. These diagrams highlight the distinct processing paths in each approach while showcasing their shared use of output-handling components.

4.2 Software-Only Implementation

In the software-only implementation, all computations for Sobel edge detection were executed by the ARM Cortex-A9 processor within the Zynq-7000 processing system (PS). The system received a single raw RGB image as input via the AXI4-Lite interface. Subsequently, the Cortex-A9 processor performed the conversion of the image to grayscale and applied the Sobel convolution operation to detect edges in the frame. After the Sobel operation was completed, the processed frame was routed through the AXI interconnect to the multiplexer (mux_v1_0). The multiplexer then directed the processed data to the VGA interface for display.

The system architecture, as shown in Figure 2, relied on several interconnected components to manage the flow of data and ensure synchronization across the various stages of processing and output. To begin with, the clocking wizard (clk_wiz_0) provided the essential clock signals that synchronized the operation of the entire system. This ensured that all components, including the PS, AXI interconnect, and VGA subsystem, operated seamlessly. Furthermore, the processor system reset module (proc_sys_reset_0) facilitated proper initialization by generating reset signals for all system modules, guaranteeing a stable startup and operation.

Moreover, the AXI interconnect served as the communication backbone for this design. It acted as a bridge, enabling efficient data transfer between the processing system and the connected

peripherals, such as the AXI Video Direct Memory Access (VDMA) and video timing controller (V_TC). The processed image frame was transferred through the interconnect, which ensured smooth data flow from the PS to the VGA pipeline. Additionally, the interconnect maintained proper synchronization among all the connected modules, allowing for reliable communication throughout the system.

The AXI Video Direct Memory Access (VDMA), which further supported the system, facilitated the movement of processed frame data from the AXI interconnect to the VGA output subsystem. While the VDMA is typically used for advanced data buffering and streaming in real-time applications, in this implementation, it primarily acted as a straightforward data transfer mechanism. It ensured that the processed frame from the PS reached the VGA subsystem efficiently.

In addition to the VDMA, the video timing controller (V_TC) played an integral role in enabling proper visualization of the output frame on the VGA display. The V_TC generated horizontal and vertical synchronization signals (Hsync and Vsync), which were necessary for the VGA display to correctly interpret and render the processed frame. These timing signals, synchronized with the clocking wizard, ensured the frame was displayed with proper alignment, resolution, and stability. Furthermore, the multiplexer (mux_v1_0) provided flexibility in controlling whether the raw input data or the processed output was displayed on the VGA interface. The multiplexer, configured to prioritize processed data, directed the output to the VGA subsystem. Finally, the VGA pipeline prepared the processed frame for visualization, ensuring that the data was appropriately routed to the display pins.

Notably, this software-only implementation did not engage the programmable logic (PL) for computational tasks, resulting in minimal resource utilization. As highlighted in Figure 5, the PL fabric remained largely idle, with only essential modules like the AXI interconnect, VDMA, and V_TC being active. This underutilization of the PL limited the system's processing efficiency, as all computational tasks were sequentially executed by the ARM Cortex-A9 processor in the PS. While the system was sufficient for processing static frames, its reliance on sequential execution caused significant delays. Additionally, the absence of parallel processing made the software-only design unsuitable for real-time applications. These constraints emphasized the need for a more efficient approach that leveraged the PL to improve performance. Consequently, the limitations of this design set the stage for the development and evaluation of the hardware-accelerated implementation described later.

4.3 Hardware-Accelerated Implementation

The hardware accelerated implementation utilized the FPGA's programmable logic (PL) to process live video streams from a connected webcam in real time, addressing the computational challenges of the software-only design. Unlike the static frame processing in the software-only

implementation, this design handled continuous video streams, achieving significant performance improvements. As shown in Figure 3, video frames from the webcam were streamed into the system via the AXI4 Stream interface, managed by the AXI Video Direct Memory Access (VDMA). The VDMA ensured that incoming video frames were transferred to the PL without delay, maintaining a steady flow of data for real-time processing.

A critical component of this implementation was the custom Sobel IP block, synthesized in Vivado HLS. This IP block offloaded the computationally intensive parts of the edge detection pipeline from the ARM Cortex-A9 processor to the FPGA. It performed multiple stages of image processing, including grayscale conversion, Gaussian noise reduction, and Sobel convolution for edge detection. By consolidating these operations into a single hardware module, the custom IP block efficiently processed incoming video frames in real time.

The grayscale conversion was implemented as the first stage within the IP block. This step reduced the three RGB color channels into a single intensity channel, simplifying subsequent processing. Following grayscale conversion, the Gaussian noise reduction filter was applied to smooth the input image and minimize the effects of noise, improving the accuracy of the edge detection process. These preprocessing steps ensured that the input to the convolution stage was clean and optimized for detecting meaningful edges.

Additionally, the final stage within the IP block was the Sobel convolution operation, which calculated horizontal and vertical gradients for each pixel in the grayscale image. Using these gradients, the Sobel operator determined the intensity of edges and highlighted regions of rapid intensity change. The parallel processing capabilities of the FPGA enabled the IP block to handle multiple pixels simultaneously, significantly reducing processing time compared to the sequential nature of the software-only implementation. Once processed, the output frames were routed back through the AXI interconnect and directed to the multiplexer (mux_v1_0). This multiplexer allowed the system to toggle between displaying the raw video feed from the webcam and the edge-detected frames produced by the Sobel IP block. This flexibility was essential during testing and validation, as it enabled comparisons between the original and processed video streams. Moreover, the video timing controller (VTC) received the selected video stream and generated the required synchronization signals to ensure proper alignment and timing for VGA display. The processed frames were then displayed in real time on a connected monitor, demonstrating the system's ability to handle continuous video streams effectively.

In addition, the synchronization across the system was maintained by the clocking wizard (clk_wiz_0), which supplied clock signals to the VDMA, Sobel IP block, and VGA subsystem. The processor system reset module (proc_sys_reset_0) ensured reliable system initialization, setting all components to a stable state at startup. As shown in Figure 6, this implementation significantly utilized the FPGA’s resources. The inclusion of the Sobel IP block, VDMA, and

additional modules required a substantial portion of the PL's logic and memory. However, this resource usage was necessary to enable the real-time processing of live video streams and to perform edge detection efficiently.

This hardware accelerated design demonstrated the effectiveness of offloading computationally intensive tasks such as grayscale conversion, Gaussian noise reduction, and Sobel convolution to the programmable logic. By processing video frames directly in hardware, the system achieved real-time performance and seamless handling of live webcam input. The integration of the Sobel IP block with the AXI interconnect, VDMA, and VGA output ensured efficient data flow and reliable synchronization, making this design suitable for demanding applications such as obstacle detection and autonomous navigation.

5 Results and Analysis

5.1 Performance Metrics

The performance analysis reveals substantial gains achieved through hardware acceleration when compared to the software only approach. Both implementations were evaluated using key metrics including processing time, power consumption, and FPGA resource utilization. Figure 7 summarizes these findings, while the comparative bar chart in Figure 8 provides a clear visual indication of the performance improvements delivered by hardware acceleration.

In the software only implementation, the ARM Cortex A9 processor handled every computational task sequentially. This included grayscale conversion, Gaussian noise reduction, and Sobel convolution on a single static raw RGB frame. Although the system successfully performed each image processing step, it required approximately 142 ms to complete a single frame. This long processing time made it unsuitable for real time applications. Since the processor operated without support from the programmable logic, scaling to higher frame rates was not feasible. While processed frames could be displayed through the VGA interface, the limited throughput and elevated latency rendered this approach inadequate for dynamic scenarios such as autonomous navigation, where continuous and rapid image processing is crucial.

In contrast, the hardware accelerated implementation employed the FPGA programmable logic to perform the same image processing pipeline on a live video feed from a connected webcam. Tasks such as Sobel convolution, grayscale conversion, and Gaussian noise reduction were offloaded to a custom IP block created using Vivado HLS. By exploiting parallelism and applying optimizations like pipelining and loop unrolling, the hardware accelerated system reduced the per frame processing time to about 22 ms. This represents a speedup of approximately 6.5 times over the software only solution, enabling true real time performance. With efficient video frame transfers supported by the AXI4 Stream interface and managed by the AXI Video Direct

Memory Access (VDMA) module, the system could seamlessly deliver processed frames to the VGA interface, as shown in Figure 4. Although, the frames were loaded slowly due to the processing time, the result was a smooth, low latency visualization of incoming video data.

Power consumption measurements indicated that both solutions drew similar amounts of power. The software only design consumed 1.843 W, while the hardware accelerated implementation required slightly less at 1.825 W. Although this difference is marginal, the hardware accelerated design handled far more demanding real time workloads at virtually the same power level. This underscores the efficiency of leveraging FPGA resources rather than relying solely on the ARM processor, allowing for enhanced performance without significantly increasing power demands.

Resource utilization also differed considerably between the two approaches. The hardware accelerated system used more of the FPGA fabric to maintain a high-performance processing pipeline. It employed 10.32% of the slice LUTs, 5.43% of the slice registers, and 6.07% of the block RAM, as well as 18.64% of the DSP slices to support real time convolution operations. In comparison, the software only system depended entirely on the processor and used minimal FPGA resources, reflected by only 3.21% LUTs, 2.28% registers, and 2.86% block RAM, with no DSP slices utilized. Although the hardware design required additional FPGA resources, the resulting improvements in frame rate, ability to handle live video streams, and faster response times justified the tradeoff. Such adjustments are common in FPGA based solutions, where logic, memory, and DSP resources can be strategically allocated to meet performance goals.

Overall, our analysis clearly demonstrates that hardware acceleration delivers a level of performance the software only solution cannot achieve. The hardware accelerated implementation offered lower latency, higher frame rates, and the capacity to process live video streams continuously, all with modest power usage. These results, illustrated in Figures 7 and 8, validate the approach of offloading intensive operations to dedicated hardware components. For applications requiring real time image processing, such as autonomous navigation or surveillance, the hardware accelerated solution provides a definitive advantage over a software only design.

5.2 Error Analysis

During development and testing, we faced multiple challenges which affected our ability to conduct a detailed performance evaluation. Our main issue was configuring the UART port on our target board, which prevented the use of tools like gprof for collecting runtime profiling data. Without UART functionality, we lacked insights into instruction-level execution and function call overhead.

In addition to the UART issue, errors related to improper library imports in the Xilinx SDK environment further complicated the process. Addressing these errors required keen observations

of include paths and thorough compatibility checks between libraries. Although these difficulties did not compromise the core functionality of our design, they introduced many delays into project implementation and emphasized the importance of proactive configuration management. Without detailed runtime data from UART-based profiling, we relied on Vivado's post-implementation reports for performance insights. Moreover, these reports lacked the dynamism of runtime profiling, however they provided valuable information on resource usage, clock performance, and estimated power consumption. By leveraging these metrics, we quantified the improvements achieved through hardware acceleration, alongside qualitative observations, and established a meaningful comparison with the software-only implementation.

Moreover, intermittent synchronization issues and compatibility challenges with certain IP blocks occasionally disrupted the development workflow. Through iterative debugging and systematic parameter adjustments, these problems were eventually resolved using root cause analysis. This process highlighted the complexities in hardware-software integration and demonstrated the need for refinement throughout the development process. Despite the absence of UART-based profiling and the setbacks encountered in the SDK environment, the data from Vivado reports allowed us to perform an in-depth comparison. It aided in providing concrete results, confirming that hardware acceleration provided substantial gains in throughput and efficiency.

6 Conclusion

Our work has demonstrated the confirmation of hardware acceleration considerably enhancing performance and power in Sobel edge detection for an FPGA platform. The hardware-accelerated design achieved much better frame-per-second processing than our purely software-based solution, at approximately equal power consumption. Thereby, our approach reached all requirements on real-time capability without excessive raises in energy costs. We minimized processor load, introduced parallel processing, and reduced latency to achieve greater throughput by offloading the heavy computing tasks of grayscale conversion, Gaussian noise reduction, and Sobel convolution onto dedicated FPGA logic.

Although we faced certain development challenges, including the inability to configure the UART port and issues with SDK library imports, we adapted our methods accordingly. Without detailed runtime profiling, we relied on Vivado post-implementation reports for valuable insights into resource usage, power consumption, and clock performance. Other synchronization issues and refinements of IP compatibility further improved the stability and showed how iterative testing and careful refinement of a design are important. Looking ahead, our work points to several avenues for future improvements. Integration of advanced filtering techniques, studying complex algorithms for feature detection, and integrating machine learning-based solutions can further enhance our system. Additionally, optimizing resource usage and power consumption, through

partial reconfiguration or energy-aware strategies, will allow our hardware-accelerated methodology to be applied to a broader range of real-time image processing and autonomous navigation scenarios. In summary, our results validate the effectiveness of hardware acceleration for Sobel edge detection and pave the way for more sophisticated and adaptive embedded systems.

7 References

[1] D. K. Mandal et al., "Visual Inertial Odometry At the Edge: A Hardware-Software Co-design Approach for Ultra-low Latency and Power," 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), Florence, Italy, 2019, pp. 960-963, doi: 10.23919/DATE.2019.8714921.

[2] I. Sobel and G. Feldman, "A 3x3 isotropic gradient operator for image processing," presented at the Stanford Artificial Intelligence Project (SAIL), 1968. [Online]. Available: https://www.researchgate.net/publication/285159837_A_33_isotropic_gradient_operator_for_image_processing

[3] S. El-Sayed, "A novel approach to improve Sobel edge detector," Procedia Computer Science, vol. 12, pp. 91-97, 2012. [Online]. Available: https://www.researchgate.net/publication/306072430_A_Novel_Approach_to_Improve_Sobel_Edge_Detector/fulltext/57adffd908aeb2cf17bda10c/A-Novel-Approach-to-Improve-Sobel-Edge-Detector.pdf

[4] T. Imsaengsuk and S. Pumrin, "Feature Detection and Description based on ORB Algorithm for FPGA-based Image Processing," 2021 9th International Electrical Engineering Congress (iEECON), Pattaya, Thailand, 2021, pp. 420-423, doi: 10.1109/iEECON51072.2021.9440232.

[5] A. Jain, "All about convolutions, kernels, features in CNN," 12 February 2024. [Online]. Available: https://medium.com/@abhishekjainindore24/all-about-convolutions-kernels-featuresin-cnn-c656616390a1.

Appendix A – Figures & Data

Figure 1: Contribution Breakdown

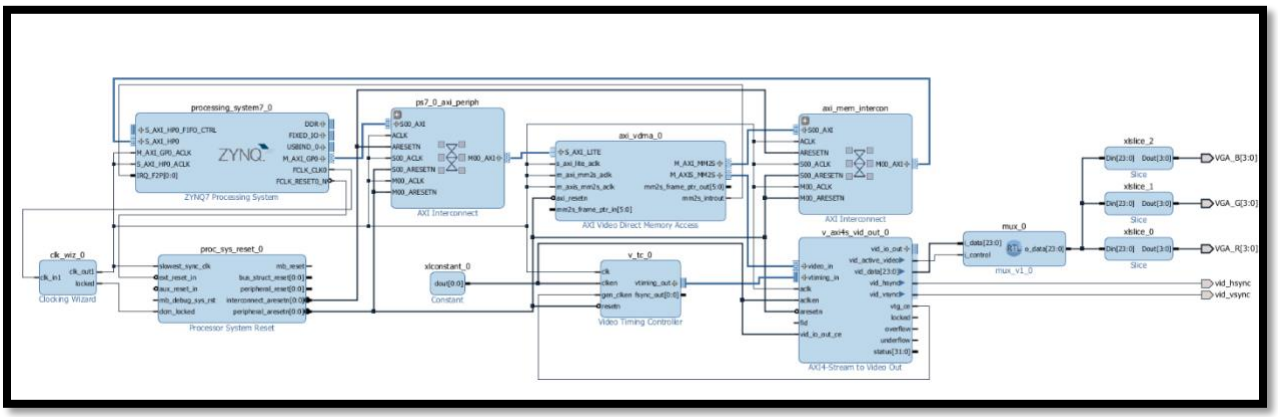

Figure 2: System architecture of the software-only implementation

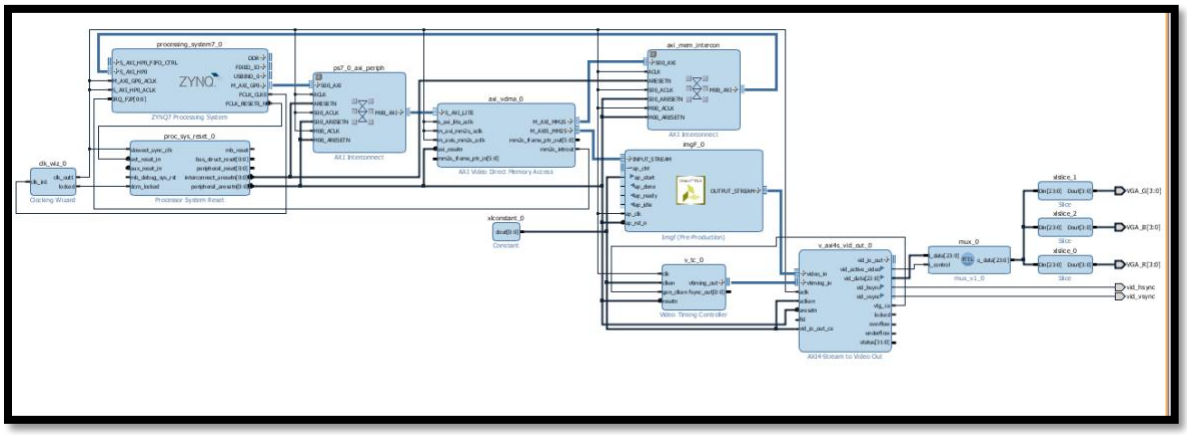

Figure 3: System architecture of the hardware-accelerated implementation

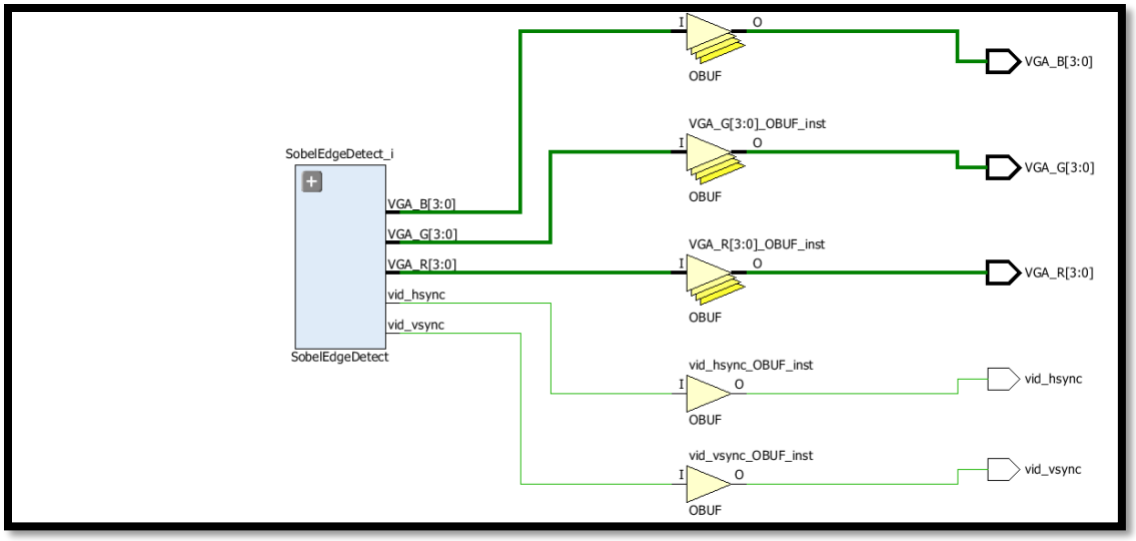

Figure 4: Diagram of the VGA interface and video output handling

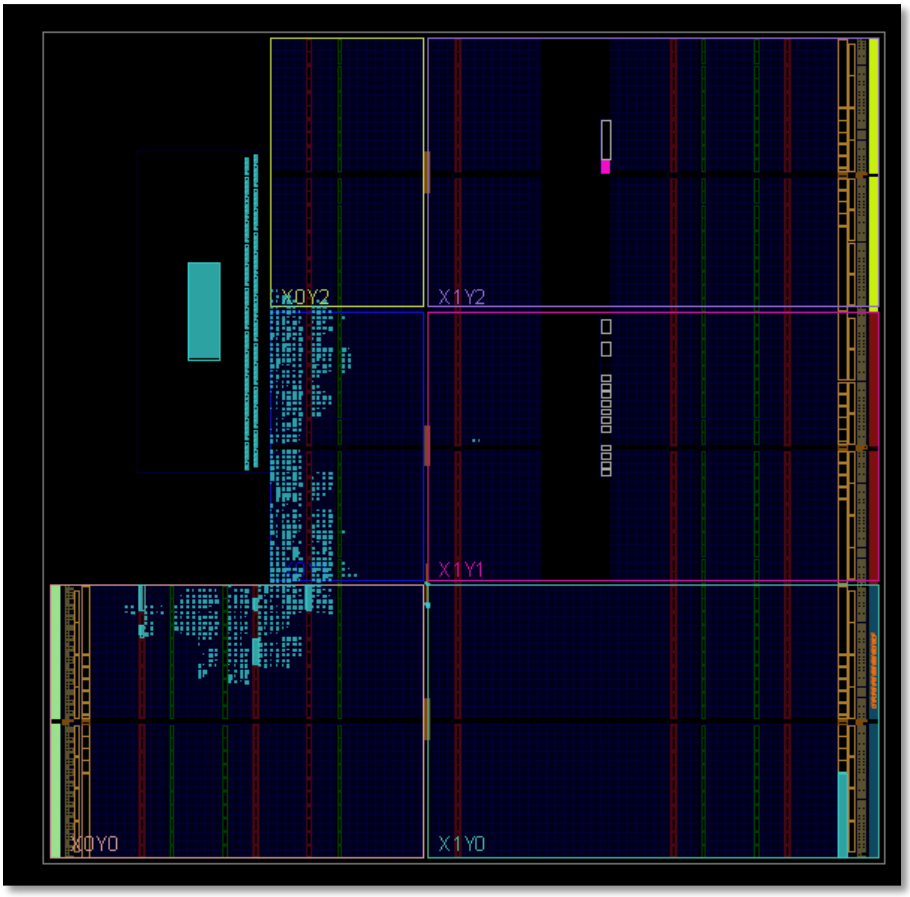

Figure 5 Area utilization for the software-only implementation

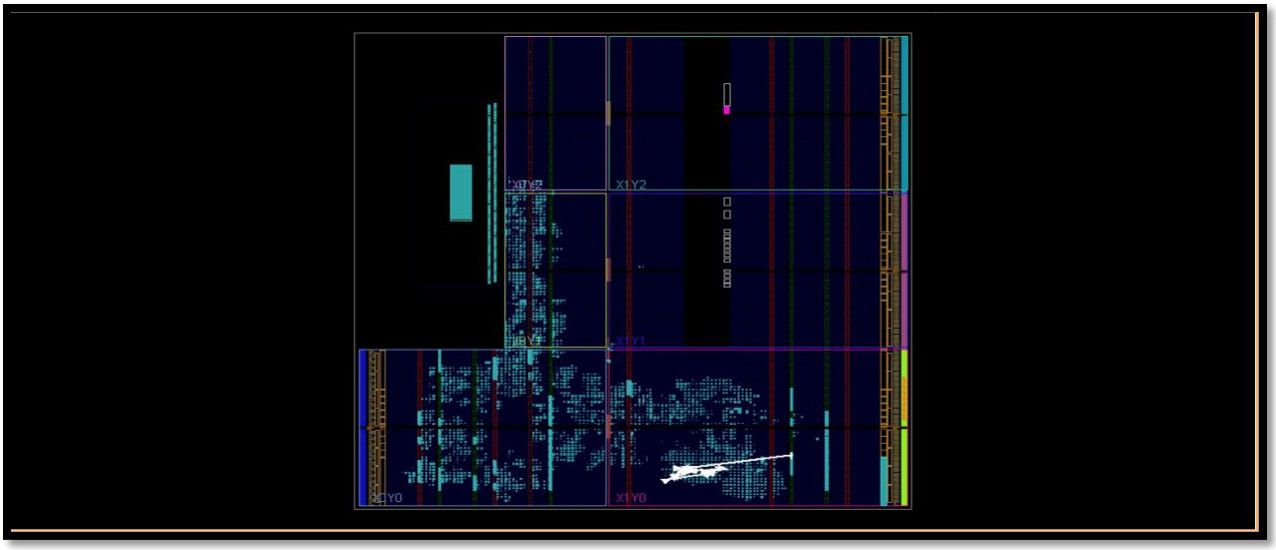

Figure 6: Utilization for the hardware-accelerated implementation

Metric	Software-Only Implementation	Hardware-Accelerated Implementation
Processing Time per Frame	142 ms	22 ms
Total On-Chip Power (W)	1.843	1.825
Slice LUT Utilization (%)	3.21%	10.32%
Slice Registers Utilization (%)	2.28%	5.43%
DSP Utilization (%)	0%	18.64%
Block RAM Utilization (%)	2.86%	6.07%

Figure 7: Performance Metrics Comparison Between Software-Only and Hardware-Accelerated Implementations

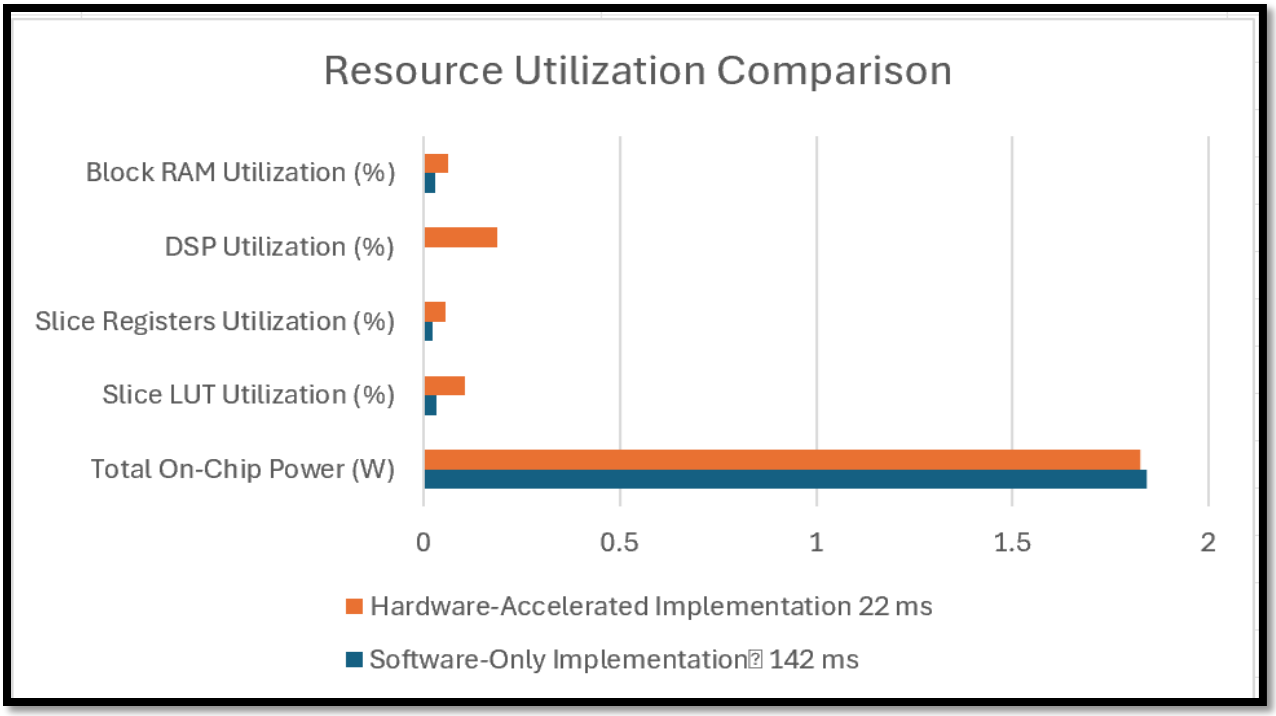

Figure 8: Graph of Resource Utilization Comparison for Software-Only and Hardware-Accelerated Implementations

Appendix B – Implemented Code

Software Implementation

```
/*
 * vdmaTest.c
 *
 *  Created on: Nov. 29, 2024
 *      Author: grp 14
 */
#include "xparameters.h"
#include "xaxivdma.h"
#include "xscugic.h"
#include "sleep.h"
#include <stdlib.h>
#include "xil_cache.h"
#include "xil_cache.h"
#include "imageData.h"      //Holds image data in a separate header file

#define HSize 1920
#define VSize 1080
#define FrameSize HSize*VSize*3
#define imgHSize 512
#define imgVSize 512

static XScuGic Intc;


static int SetupIntrSystem(XAxiVdma *AxiVdmaPtr, u16 ReadIntrId);
int drawImage(u32 displayHSize,u32 displayVSize,u32 imageHSize,u32 imageVSize,u32 hOffset, u32 vOffset,char
*imagePointer);

unsigned char Buffer[FrameSize];

int main(){
    int status;
    int Index;
    u32 Addr;
    XAxiVdma myVDMA;
    XAxiVdma_Config *config = XAxiVdma_LookupConfig(XPAR_AXI_VDMA_0_DEVICE_ID);
    XAxiVdma_DmaSetup ReadCfg;
    status = XAxiVdma_CfgInitialize(&myVDMA, config, config->BaseAddress);
    if(status != XST_SUCCESS){
        xil_printf("DMA Initialization failed");
```

```
    }
    ReadCfg.VertSizeInput = VSize;
    ReadCfg.HoriSizeInput = HSize*3;
    ReadCfg.Stride = HSize*3;
    ReadCfg.FrameDelay = 0;
    ReadCfg.EnableCircularBuf = 1;
    ReadCfg.EnableSync = 1;
    ReadCfg.PointNum = 0;
    ReadCfg.EnableFrameCounter = 0;
    ReadCfg.FixedFrameStoreAddr = 0;
    status = XAxiVdma_DmaConfig(&myVDMA, XAXIVDMA_READ, &ReadCfg);
    if (status != XST_SUCCESS) {
        xil_printf("Write channel config failed %d\r\n", status);
        return status;
    }

    Addr = (u32)&(Buffer[0]);


    for(Index = 0; Index < myVDMA.MaxNumFrames; Index++) {
        ReadCfg.FrameStoreStartAddr[Index] = Addr;
        Addr +=  FrameSize;
    }

    status = XAxiVdma_DmaSetBufferAddr(&myVDMA, XAXIVDMA_READ,ReadCfg.FrameStoreStartAddr);
    if (status != XST_SUCCESS) {
        xil_printf("Read channel set buffer address failed %d\r\n", status);
        return XST_FAILURE;
    }

    XAxiVdma_IntrEnable(&myVDMA, XAXIVDMA_IXR_COMPLETION_MASK, XAXIVDMA_READ);

    SetupIntrSystem(&myVDMA, XPAR_FABRIC_AXI_VDMA_0_MM2S_INTROUT_INTR);

    drawImage(HSize,VSize,imgHSize,imgVSize,(HSize-imgHSize)/2,(VSize-imgVSize)/2,imageData);


    status = XAxiVdma_DmaStart(&myVDMA,XAXIVDMA_READ);
    if (status != XST_SUCCESS) {
        if(status == XST_VDMA_MISMATCH_ERROR)
            xil_printf("DMA Mismatch Error\r\n");
        return XST_FAILURE;
    }

    while(1){
```

```
    }
}
/*****************************************************************************/
 /* Call back function for read channel
******************************************************************************/

static void ReadCallBack(void *CallbackRef, u32 Mask)
{
    /* User can add his code in this call back function */
    xil_printf("Read Call back function is called\r\n");
}
/*****************************************************************************/
/*
 * The user can put his code that should get executed when this
 * call back happens.
 *
******************************************************************************/
static void ReadErrorCallBack(void *CallbackRef, u32 Mask)
{
    /* User can add his code in this call back function */
    xil_printf("Read Call back Error function is called\r\n");

}
static int SetupIntrSystem(XAxiVdma *AxiVdmaPtr, u16 ReadIntrId)
{
    int Status;
    XScuGic *IntcInstancePtr =&Intc;

    /* Initialize the interrupt controller and connect the ISRs */
    XScuGic_Config *IntcConfig;
    IntcConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
    Status =  XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig, IntcConfig->CpuBaseAddress);
    if(Status != XST_SUCCESS){
        xil_printf("Interrupt controller initialization failed..");
        return -1;
    }

    Status = XScuGic_Connect(IntcInstancePtr,ReadIntrId,(Xil_InterruptHandler)XAxiVdma_ReadIntrHandler,(void
*)AxiVdmaPtr);
    if (Status != XST_SUCCESS) {
        xil_printf("Failed read channel connect intc %d\r\n", Status);
        return XST_FAILURE;
    }
    XScuGic_Enable(IntcInstancePtr,ReadIntrId);
```

```
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,(void
*)IntcInstancePtr);
    Xil_ExceptionEnable();

    /* Register call-back functions
     */
    XAxiVdma_SetCallBack(AxiVdmaPtr, XAXIVDMA_HANDLER_GENERAL, ReadCallBack, (void *)AxiVdmaPtr, XAXIVDMA_READ);

    XAxiVdma_SetCallBack(AxiVdmaPtr, XAXIVDMA_HANDLER_ERROR, ReadErrorCallBack, (void *)AxiVdmaPtr, XAXIVDMA_READ);

    return XST_SUCCESS;
}


int drawImage(u32 displayHSize, u32 displayVSize, u32 imageHSize, u32 imageVSize, u32 hOffset, u32 vOffset, char
*imagePointer) {
    unsigned char grayscale[imgVSize][imgHSize];
    unsigned char sobelImage[imgVSize][imgHSize];

    // Convert to grayscale
    for (int i = 0; i < imgVSize; i++) {
        for (int j = 0; j < imgHSize; j++) {
            unsigned char R = *imagePointer++;
            unsigned char G = *imagePointer++;
            unsigned char B = *imagePointer++;
            grayscale[i][j] = (unsigned char)(0.3 * R + 0.59 * G + 0.11 * B);
        }
    }

    // Sobel edge detection
    int Gx, Gy;
    int sobelKernelX[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    int sobelKernelY[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};
    for (int i = 1; i < imgVSize - 1; i++) {
        for (int j = 1; j < imgHSize - 1; j++) {
            Gx = Gy = 0;
            for (int ki = -1; ki <= 1; ki++) {
                for (int kj = -1; kj <= 1; kj++) {
                    Gx += sobelKernelX[ki + 1][kj + 1] * grayscale[i + ki][j + kj];
                    Gy += sobelKernelY[ki + 1][kj + 1] * grayscale[i + ki][j + kj];
                }
            }
            int magnitude = abs(Gx) + abs(Gy);
            sobelImage[i][j] = (magnitude > 255) ? 255 : magnitude;
```

```
        }
    }

    // Fill Buffer for VGA output with the Sobel edge-detected image
    for (int i = 0; i < VSize; i++) {
        for (int j = 0; j < HSize; j++) {
            if (i < vOffset || i >= vOffset + imageVSize || j < hOffset || j >= hOffset + imageHSize) {
                // Fill outside the image bounds with black
                Buffer[(i * HSize * 3) + (j * 3)] = 0x00;        // Red
                Buffer[(i * HSize * 3) + (j * 3) + 1] = 0x00;    // Green
                Buffer[(i * HSize * 3) + (j * 3) + 2] = 0x00;    // Blue
            } else {
                // Map Sobel result to the Buffer
                int imgX = j - hOffset;
                int imgY = i - vOffset;
                unsigned char pixelValue = sobelImage[imgY][imgX] / 16; // Original output format
                Buffer[(i * HSize * 3) + (j * 3)] = pixelValue;        // Red
                Buffer[(i * HSize * 3) + (j * 3) + 1] = pixelValue;    // Green
                Buffer[(i * HSize * 3) + (j * 3) + 2] = pixelValue;    // Blue
            }
        }
    }
    // Flush the cache to ensure proper display
    Xil_DCacheFlush();
    return 0;
}
```

Hardware Implementation

Main File

```
#include "cvt_colour.hpp"


void imgF(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM)//, int rows, int cols)
{


#pragma HLS INTERFACE axis port=INPUT_STREAM
#pragma HLS INTERFACE axis port=OUTPUT_STREAM


RGB_IMAGE  img_0(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE img_1(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE  img_2(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE  img_2a(MAX_HEIGHT, MAX_WIDTH);
```

```
GRAY_IMAGE  img_2b(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE  img_3(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE  img_4(MAX_HEIGHT, MAX_WIDTH);
GRAY_IMAGE  img_5(MAX_HEIGHT, MAX_WIDTH);
RGB_IMAGE  img_6(MAX_HEIGHT, MAX_WIDTH);
;

#pragma HLS dataflow
hls::AXIvideo2Mat(INPUT_STREAM, img_0);
hls::CvtColor<HLS_BGR2GRAY>(img_0, img_1);
hls::GaussianBlur<3,3>(img_1,img_2);
hls::Duplicate(img_2,img_2a,img_2b);
hls::Sobel<1,0,3>(img_2a, img_3);
hls::Sobel<0,1,3>(img_2b, img_4);
hls::AddWeighted(img_4,0.5,img_3,0.5,0.0,img_5);
hls::CvtColor<HLS_GRAY2RGB>(img_5, img_6);

hls::Mat2AXIvideo(img_6, OUTPUT_STREAM);
}
```

Header File

```
#include  "hls_video.h"
#include <ap_fixed.h>

#define MAX_WIDTH  512
#define MAX_HEIGHT 512

typedef hls::stream<ap_axiu<24,1,1,1> >           AXI_STREAM;
typedef hls::Mat<MAX_HEIGHT,   MAX_WIDTH,   HLS_8UC3> RGB_IMAGE;
typedef hls::Mat<MAX_HEIGHT,   MAX_WIDTH,   HLS_8UC1> GRAY_IMAGE;



void imgF(AXI_STREAM& INPUT_STREAM, AXI_STREAM& OUTPUT_STREAM);//int rows, int cols);
```

Test Bench for Hardware Block

```
#include <hls_opencv.h>
#include "cvt_colour.hpp"
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;
```

```
int main (int argc, char** argv) {
    cout << "Opening video capture..." << endl;

    // Open the default camera (usually the first camera connected)
    VideoCapture cap(0);

    if (!cap.isOpened()) {
        cerr << "Error: Could not open the camera." << endl;
        return -1;
    }

    cout << "Camera opened successfully at resolution " ;

    // Create a window to display the processed video
    namedWindow("Processed Video", WINDOW_AUTOSIZE);

    AXI_STREAM src_axi, dst_axi;

    while (true) {
        Mat frame;
        cap >> frame; // Capture a new frame from the camera

        if (frame.empty()) {
            cerr << "Error: Captured empty frame." << endl;
            break;
        }

        // Ensure the frame size matches MAX_WIDTH and MAX_HEIGHT
        if (frame.cols != MAX_WIDTH || frame.rows != MAX_HEIGHT) {
            cerr << "Actual frame size: cols = " << frame.cols << ", rows = " << frame.rows << endl;
            cerr << "Error: Frame size does not match MAX_WIDTH and MAX_HEIGHT." << endl;
            break;
        }

        // Convert the Mat frame to an IplImage header
        IplImage* ipl_frame = cvCreateImageHeader(cvSize(frame.cols, frame.rows), IPL_DEPTH_8U, frame.channels());
        ipl_frame->imageData = (char*)frame.data;

        // Convert IplImage to AXI Stream
        IplImage2AXIvideo(ipl_frame, src_axi);

        // Apply the Sobel filter using the image_filter function
        imgF(src_axi, dst_axi);
```

```
        // Create an output IplImage to store the processed frame
        IplImage* dst = cvCreateImage(cvGetSize(ipl_frame), ipl_frame->depth, ipl_frame->nChannels);

        // Convert AXI Stream back to IplImage
        AXIvideo2IplImage(dst_axi, dst);

        // Convert the IplImage to a Mat for display
        Mat dst_mat = cvarrToMat(dst, true); // true indicates data copy

        // Display the processed frame
        imshow("Processed Video", dst_mat);

        // Release the allocated IplImage and header
        cvReleaseImage(&dst);
        cvReleaseImageHeader(&ipl_frame);

        // Exit the loop when 'q' or 'ESC' is pressed
        char c = (char)waitKey(1);
        if (c == 'q' || c == 27) break;
    }

    // Release the video capture object and close display windows
    cap.release();
    destroyAllWindows();

    cout << "Done." << endl;
    return 0;
}
```