



**中国科学技术大学**

University of Science and Technology of China

**ENGINEERING PRACTICE ADVANCED  
(EIEN6709P)**

**FPGA IMAGE PROCESSING**

**SUBMITTED**

**BY**

**TEAM 1**

**30.06.2025**

### TEAM MEMBER

S/N	Student ID	Name
01	SL24225001	Mohamed Bachir SANOU
02	SL24225007	Yunus Isah
03	SL24225008	Muhammad Bashir Dantani

## **Abstract**

This report discusses the implementation and optimization of a Sobel edge detection algorithm for image preprocessing on an FPGA platform. With the MNIST dataset of handwritten digits as the main experimental setup, this report illustrates how hardware acceleration can be used to greatly enhance the performance and efficiency of image preprocessing tasks, which are typically regarded as bottlenecks in computer vision workflows. The Sobel filter was first developed and tested in the Python development environment, translated to C++, and implemented in Xilinx Vitis HLS for execution on an FPGA on a Xilinx ZedBoard. Optimizations at the hardware level, including loop pipelining and reduction of bit-width, were added to maximize throughput and minimize latency. The synthesized design was integrated into a system-level environment in Vivado, and host software was created in Vitis for managing data flow and execution routines.

The evaluation results show that the FPGA-based implementation provides a considerable acceleration of processing time, while also providing stable and accurate edge detection on MNIST images. Furthermore, a user-friendly graphical interface was developed to facilitate visualization of the results. This work demonstrates the practical benefits of applying FPGA-based acceleration to preprocessing tasks and highlights an extensible approach to traditional vision algorithm implementation on hardware platforms.

## Table of Contents

<b>Abstract.....</b>	<b>3</b>
<b>1. Introduction.....</b>	<b>5</b>
<b>2. Background and Motivation.....</b>	<b>5</b>
<b>3. Methodology.....</b>	<b>7</b>
<b>3.1 Project Workflow Overview.....</b>	<b>7</b>
<b>4. Prototyping Algorithms with Python.....</b>	<b>9</b>
<b>5. High-Level Synthesis (HLS) with C++.....</b>	<b>13</b>
<b>6. FPGA Hardware Design in Vivado.....</b>	<b>22</b>
<b>7. Integration of Software-Hardware in Vitis IDE (Host Code).....</b>	<b>27</b>
<b>8. GUI Development.....</b>	<b>29</b>
<b>Reference.....</b>	<b>32</b>

## **1. Introduction**

As the computer vision field keeps expanding, the performance of image preprocessing has increasingly become the key enabler of real-time and low-latency applications. As the essential initial step in the majority of vision pipelines, this preprocessing operation improves the quality of input images by minimizing noise and accentuating important features, directly affecting machine learning model accuracy. But, software implementations of these kinds of preprocessing algorithms typically lag behind the high throughput and low power demands, particularly in embedded or edge computing setups.

In this work, the implementation and optimization of a Sobel edge detection filter, a very popular preprocessing algorithm, on an FPGA platform are analyzed. The target hardware of this work is the Xilinx ZedBoard, which is known for its favorable combination of flexibility and processing power. For a test and evaluation platform, for the purposes of testing and evaluation, the MNIST dataset, which includes thousands of  $28 \times 28$  grayscale images of handwritten digits [1], is utilized as the benchmark.

The project entails an end-to-end pipeline beginning with algorithm prototyping carried out in Python, followed by translation and optimization executed in C++ with Vitis HLS, then implemented into a hardware system in Vivado and ultimately targeted onto the FPGA. A graphical user interface-based interface was also created to enable interactive testing on MNIST samples. By leveraging hardware-aware optimizations such as pipelining and loop unrolling, the design seeks to showcase considerable performance gains over conventional CPU-centric methods.

## **2. Background and Motivation**

Image preprocessing is a significant step in computer vision systems that is frequently accountable for the conversion of raw input data into a more helpful format by refining edges, moderating noise, and readying images for feature extraction or classification. Edge detection is a prevalent and efficient method used in preprocessing, which reduces image content while preserving structural information. Among the numerous approaches available, the Sobel filter is particularly popular because of its simplicity and effectiveness at emphasizing intensity gradients in images [2].

The Sobel filter detects edges by convolving the image with two  $3 \times 3$  kernels that respond maximally to edges running vertically and horizontally. Sobel filters uses small matrices called

**kernels (or filters)** to calculate the gradient across an image. A **convolution kernel** slides across the image, computing a weighted sum of pixels in a small neighborhood (e.g.,  $3 \times 3$ ).

- This allows **local analysis** of how intensity changes.
- Different kernels are designed to detect different patterns (e.g., vertical or horizontal changes).
- $G_x$ : Detects **horizontal** intensity changes (edges that are vertical).

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- $G_y$ : Detects **vertical** intensity changes (edges that are horizontal).

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel filters also use Gradient Magnitude and Direction. A gradient measures how much the pixel intensity changes in a specific direction. In image terms a high gradient means a sharp change, likely an edge and a low gradient means little change to a smooth region. We compute the gradient at each pixel to find these intensity changes.

After convolution, two images are obtained  $G_x$  and  $G_y$ . From these, we compute the magnitude:

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

Though software implementations of the Sobel filter are simple, they are confronted with scalability and processing limitations when realized with big datasets or in real-time computing setups. It is especially true in edge computing setups, where low latency, energy efficiency, and compact form factor are prime considerations. Conventional CPUs, designed for sequential workloads, are not well placed to fully exploit the intrinsic parallelism inherent in image processing applications. In contrast to this, FPGAs offer tremendous advantages by enabling massive parallelism and user-defined data paths, which facilitate real-time image processing efficiently [3]. FPGAs are programmable compared to fixed-function GPUs or ASICs, which makes it more convenient for designers to architect the hardware for specific workloads like the Sobel filter. All these advantages make them ideal to implement preprocessing algorithms in systems where performance, flexibility, and power efficiency are priority [3].

The MNIST data set [1], commonly utilized in image classification and pattern recognition research, presents itself as a suitable test case since it is simple, well-annotated, and has a standard  $28 \times 28$  grayscale format. The purpose of this project is to critically analyze the design, implementation, and optimization of the Sobel filter in FPGA hardware with MNIST as the benchmark data set.

The inspiration for this project is the bridging of the gap between algorithm-level design and successful hardware implementation. By following a disciplined development process and FPGA optimizations, the project highlights the practical benefits and challenges of applying conventional image preprocessing algorithms to reconfigurable hardware platforms.

### 3. Methodology

This section explains the end-to-end process used for implementation and optimization of the Sobel edge detection algorithm on an FPGA platform using the MNIST dataset. The method involves software prototyping, hardware implementation through High-Level Synthesis (HLS), system-level design, and host-side integration. The methodology is broken down into a number of steps to promote accuracy, efficiency, and simplicity of demonstration.

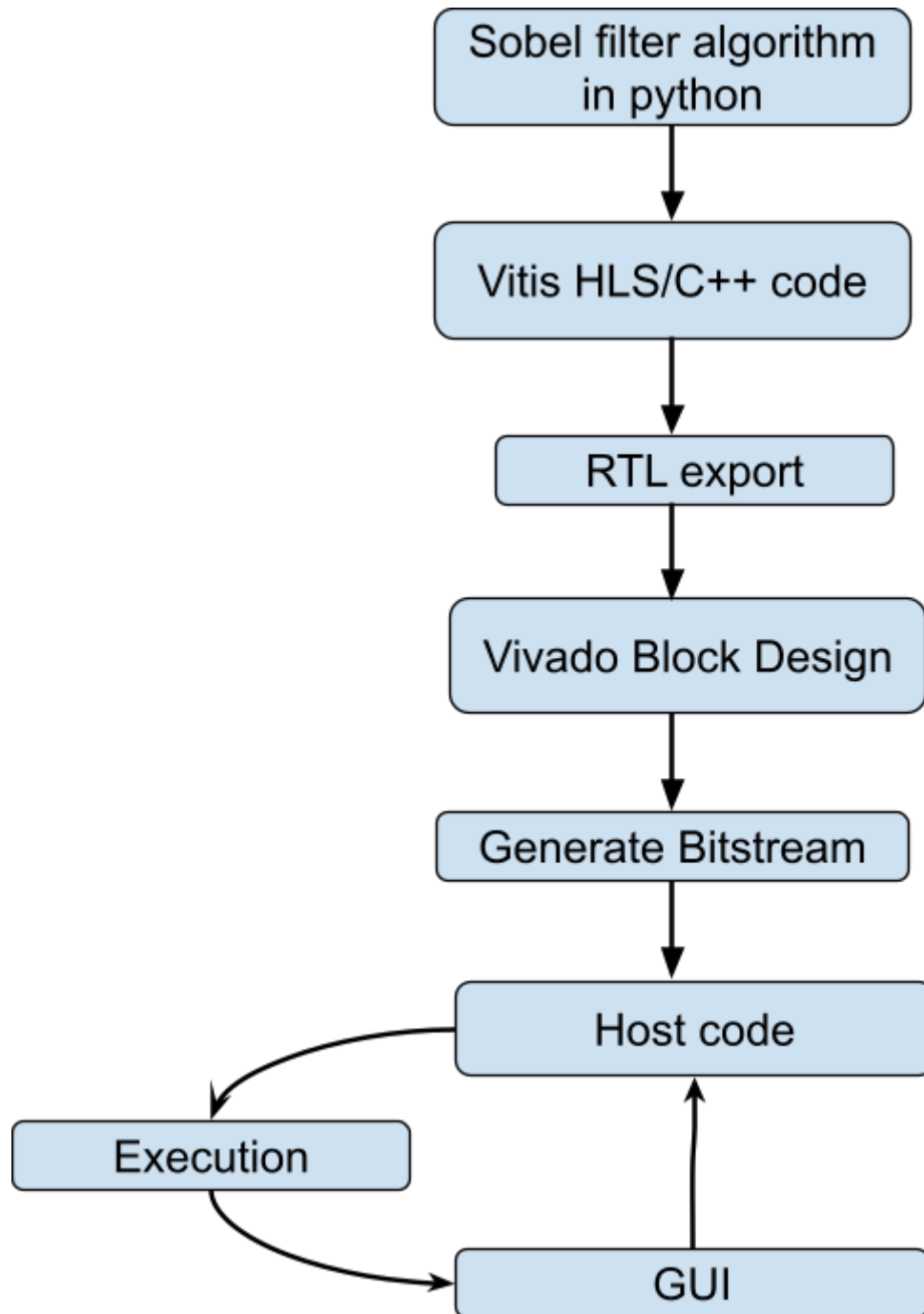
#### 3.1 Project Workflow Overview

The process of development was organized into a series of basic stages:

- i. **Algorithm Prototyping:** Sobel edge detection was initially developed and tested in Python. This step ensured that the underlying logic appropriately detected edges in  $28 \times 28$  grayscale MNIST images and provided a helpful baseline to which the FPGA implementation could be compared.
- ii. **HLS Conversion and Optimization:** The verified Python algorithm was then translated to C++ for synthesis using Xilinx Vitis HLS. FPGA-specific optimization directives such as loop pipelining, loop unrolling, and bit-width reduction were applied to enhance throughput and reduce resource utilization. The design was particularly tuned for the fixed image sizes intrinsic to the MNIST dataset. We evaluated the result of the test bench with the python result.
- iii. **FPGA Hardware Design:** The IP core created with HLS was then incorporated into a Vivado block design. The ZYNQ Processing System (PS) was set up to communicate with the Programmable Logic (PL), thereby enabling correct interaction via AXI interfaces. After system verification, the bitstream was created and exported for deployment on the Xilinx ZedBoard.
- iv. **Software-Hardware Integration:** Host-code software was developed using the Xilinx Vitis IDE platform to establish communication between the ARM processor on the ZedBoard and the Sobel accelerator in the PL. The software handled operations such as reading MNIST image data, writing pixel arrays to the accelerator, reading processed outputs, and handling user interaction.
- v. **GUI Development:** A simple graphical user interface was realized to enable demonstrations. The interface supports importing MNIST images, display of the input



and the output generated through edge detection, and filtering parameter adjustment. The GUI plays a central role in illustrating processing potential of the FPGA and provides a more interactive experience for demonstration purposes.



**Workflow**

#### **4. Prototyping Algorithms with Python**

The primary objective of this stage was to realize and verify the Sobel edge detection filter in a software framework prior to moving on to the hardware design. Python was selected since it is easy, has rich image processing library support, and allows visualization. The MNIST handwritten digit dataset was employed to verify the efficacy of the filter on low-resolution grayscale images.

##### **Methodology:**

With Python and the help of libraries such as NumPy, Matplotlib, and OpenCV, the MNIST digit images, which have a size of  $28 \times 28$  pixels, were loaded and processed. The Sobel operator in horizontal and vertical directions was applied to detect gradient changes and hence to highlight the edges of the digits.

Each picture went through the following process:

- i. Image loading and normalization to ensure consistent pixel intensity scaling.
- ii. Convolution with Sobel kernels in the X and Y directions.
- iii. Calculation of the gradient magnitude to produce a final edge map.
- iv. Visualization of both the original and processed images to assess the accuracy of edge detection.

## Python Code Snippet:

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.ndimage import convolve

def sobel_filter(image, threshold=None):

    # 1 Sobel kernel,

    sobel_x = np.array([[ -1,  0,  1],
                        [-2,  0,  2],
                        [-1,  0,  1]])

    sobel_y = np.array([[ -1, -2, -1],
                        [ 0,  0,  0],
                        [ 1,  2,  1]])

    image = image.astype(np.float32)

    grad_x = convolve(image, sobel_x, mode='constant', cval=0.0) #
horizontal gradient

    grad_y = convolve(image, sobel_y, mode='constant', cval=0.0) #
vertical gradient

    grad_mag = np.hypot(grad_x, grad_y)

    max_val = grad_mag.max()

    if max_val > 0:
```

```

        grad_mag /= max_val

    if threshold is not None:
        grad_mag = (grad_mag > threshold).astype(np.float32) #
Binarization

    return grad_mag, grad_x, grad_y

def load_mnist(csv_path):
    """
    load dataset
    """
    data = pd.read_csv(csv_path)
    labels = data.iloc[:, 0].values

    images = data.iloc[:, 1:].values.reshape(-1, 28, 28) / 255.0 #
Normalize pixels 0 and 1

    return images, labels

def plot_results(original, filtered, title="Filtered image"):

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.title("Image Originale")
    plt.imshow(original, cmap='gray')
    plt.axis('off')

```

```

plt.subplot(1, 2, 2)

plt.title(title)

plt.imshow(filtered, cmap='gray')

plt.axis('off')


plt.show()


# main function
if __name__ == "__main__":

    images, labels = load_mnist("test/mnist_train.csv")

    print(len(images))

    sample_idx = 3

    original_image = images[sample_idx]

    filtered_image, grad_x, grad_y = sobel_filter(original_image,
threshold=0.2)

    plot_results(original_image, filtered_image, "Sobel filter")

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1).imshow(grad_x, cmap='gray').set_title('Gradient
X')

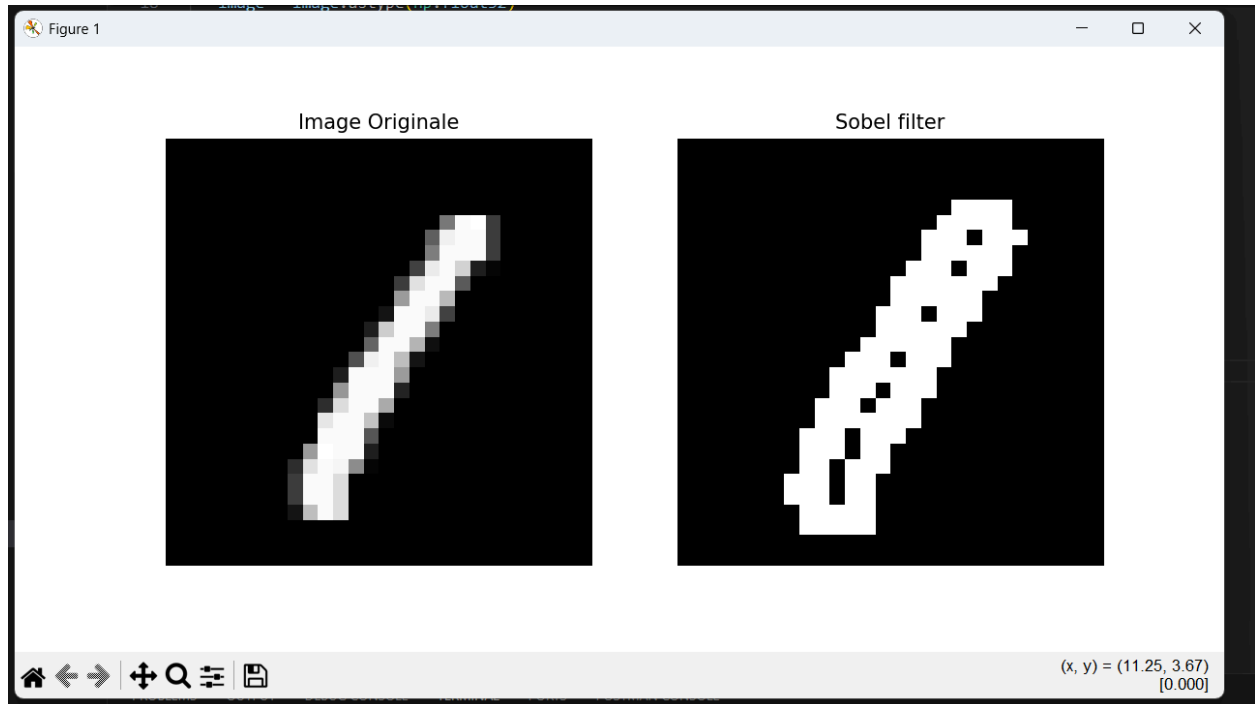
    plt.subplot(1, 3, 2).imshow(grad_y, cmap='gray').set_title('Gradient
Y')

    plt.subplot(1, 3, 3).imshow(filtered_image,
cmap='gray').set_title('Magnitude du Gradient')

```

```
plt.show()
```

## Outcome:



Both original and processed image visualizations in order to ascertain the degree of accuracy in edge detection.

## 5. High-Level Synthesis (HLS) with C++

The objective in this stage was to port the Sobel edge detection algorithm previously verified in Python to C++, then synthesize it into hardware using Xilinx Vitis High-Level Synthesis (HLS) to enable the implementation of the algorithm as a custom hardware accelerator on the FPGA that was optimized for the fixed image dimensions of the MNIST dataset.

## Methodology:

The Sobel edge detection algorithm originally developed in Python was re-implemented in C++ with hardware design considerations, such as efficient memory access, appropriate data types, and modular code structure. The implementation was organized into three core files: **sobel.cpp**, containing the main functional logic; **sobel.h**, defining data types and function prototypes; and **sobel\_tb.cpp**, serving as the testbench for simulation and verification. The C++ design was synthesized into a Register Transfer Level (RTL) description using Xilinx Vitis HLS, targeting the programmable logic (PL) of the ZedBoard based on the **xc7z020clg484-1** Zynq-7000 SoC.

To improve the performance and hardware efficiency of the design, several high-level synthesis (HLS) optimization directives were employed:

- i. **Loop Pipelining:** Enabled concurrent processing of adjacent pixels by overlapping iterations of loops, thus reducing initiation intervals and overall latency. For example we use *#pragma HLS PIPELINE II=1* and *#pragma HLS PIPELINE II=2* to reduce latency and allow concurrent processing.
- ii. **Loop Unrolling:** Applied to inner convolution loops to increase parallelism and throughput, especially during the horizontal and vertical gradient calculations. We used *#pragma HLS UNROLL* that significantly boosts throughput during the 3×3 Sobel kernel gradient calculations.
- iii. **Fixed-Point Arithmetic:** Floating-point operations were replaced with fixed-point equivalents where precision loss was negligible, leading to reduced hardware complexity and faster synthesis.
- iv. **Array Partitioning and Dataflow:** These directives allowed simultaneous memory access and concurrent execution of independent processing stages, facilitating higher parallelism and better resource utilization. The input image was handled as a two-dimensional matrix, and careful management of boundary conditions ensured valid memory access during kernel operations. Intermediate values, such as the horizontal and vertical gradients, were buffered internally to enable efficient pipelined processing throughout the filter stages.

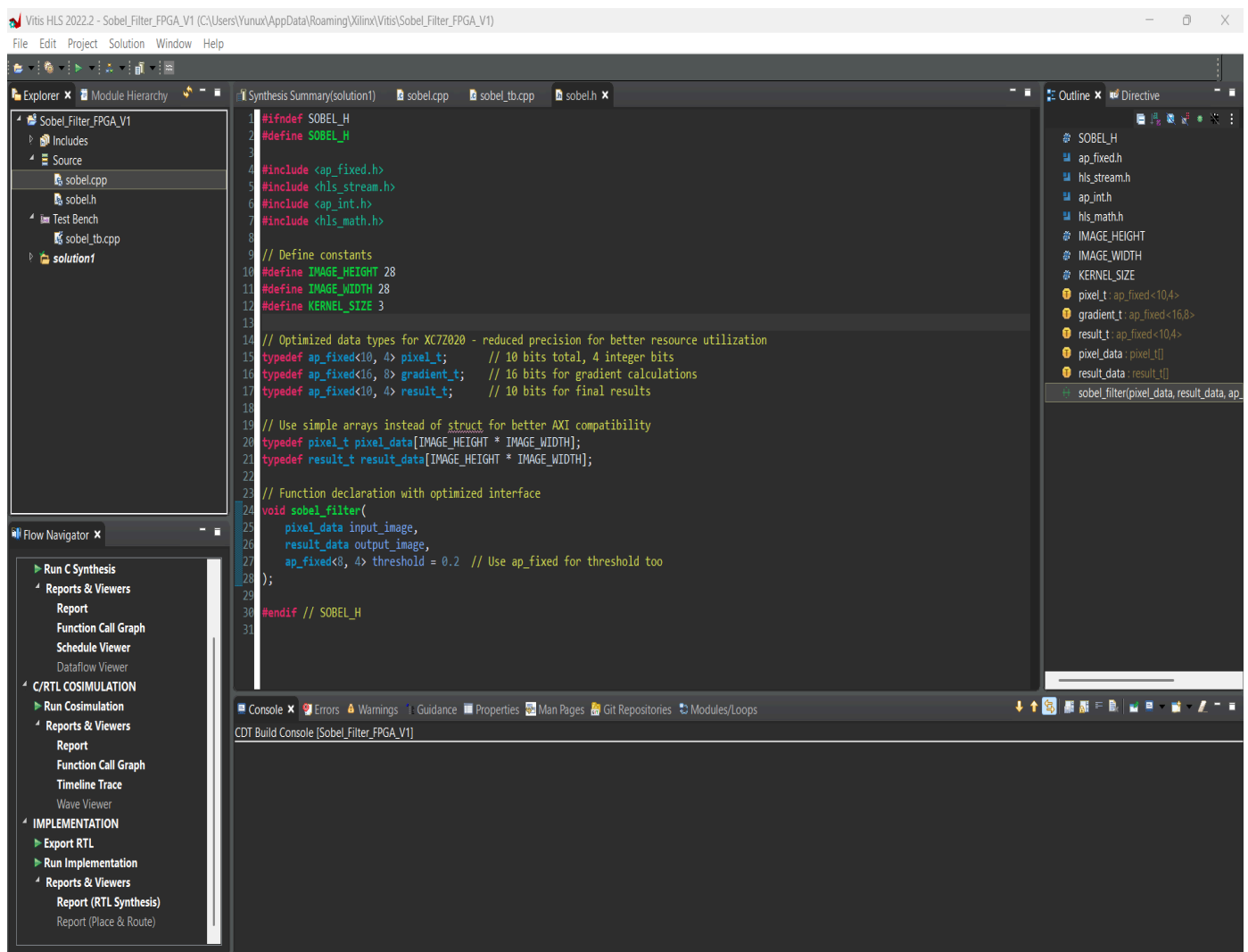
The code below allows **parallel access to multiple elements** in the same row, important for Sobel 3×3 filtering

```
#pragma HLS ARRAY_PARTITION variable=Gx complete dim=0
#pragma HLS ARRAY_PARTITION variable=Gy complete dim=0
```

```
#pragma HLS ARRAY_PARTITION variable=local_input cyclic factor=2 dim=2
#pragma HLS ARRAY_PARTITION variable=local_output cyclic factor=2 dim=2
```

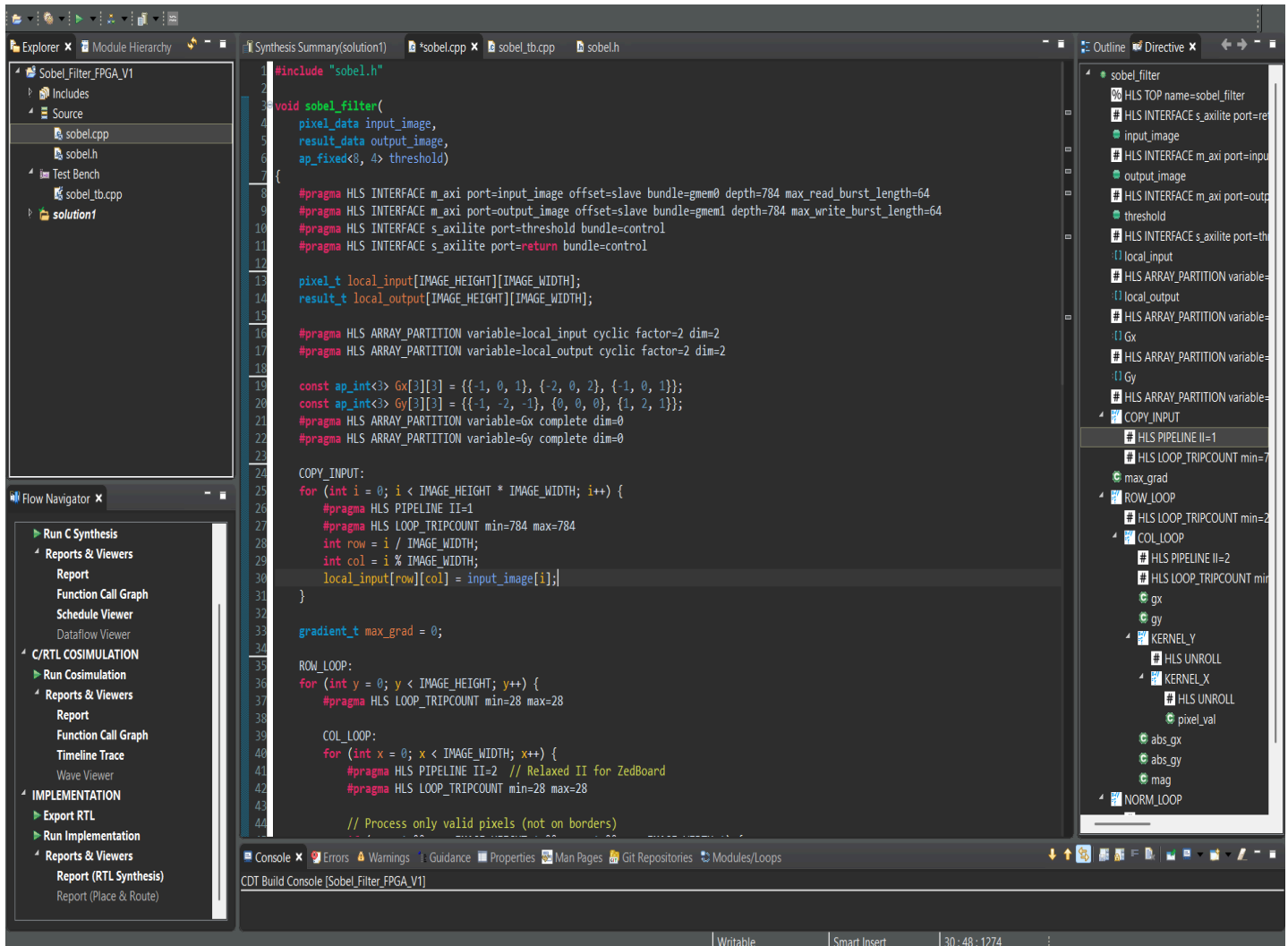
To illustrate the HLS-based implementation, key excerpts from each of the three C++ source files are presented below.

**(a) sobel.h - Header File:** This file defines data types, image dimensions, and the function prototype for the Sobel filter.





**(b) sobel.cpp – Main Function Implementation:** This file contains the actual implementation of the Sobel edge detection algorithm, written for synthesis using HLS-friendly constructs.



(c) **sobel\_tb.cpp – Testbench for Verification:** This testbench file was used to simulate and verify the correctness of the Sobel filter before synthesis.

The screenshot displays an IDE with the following components:

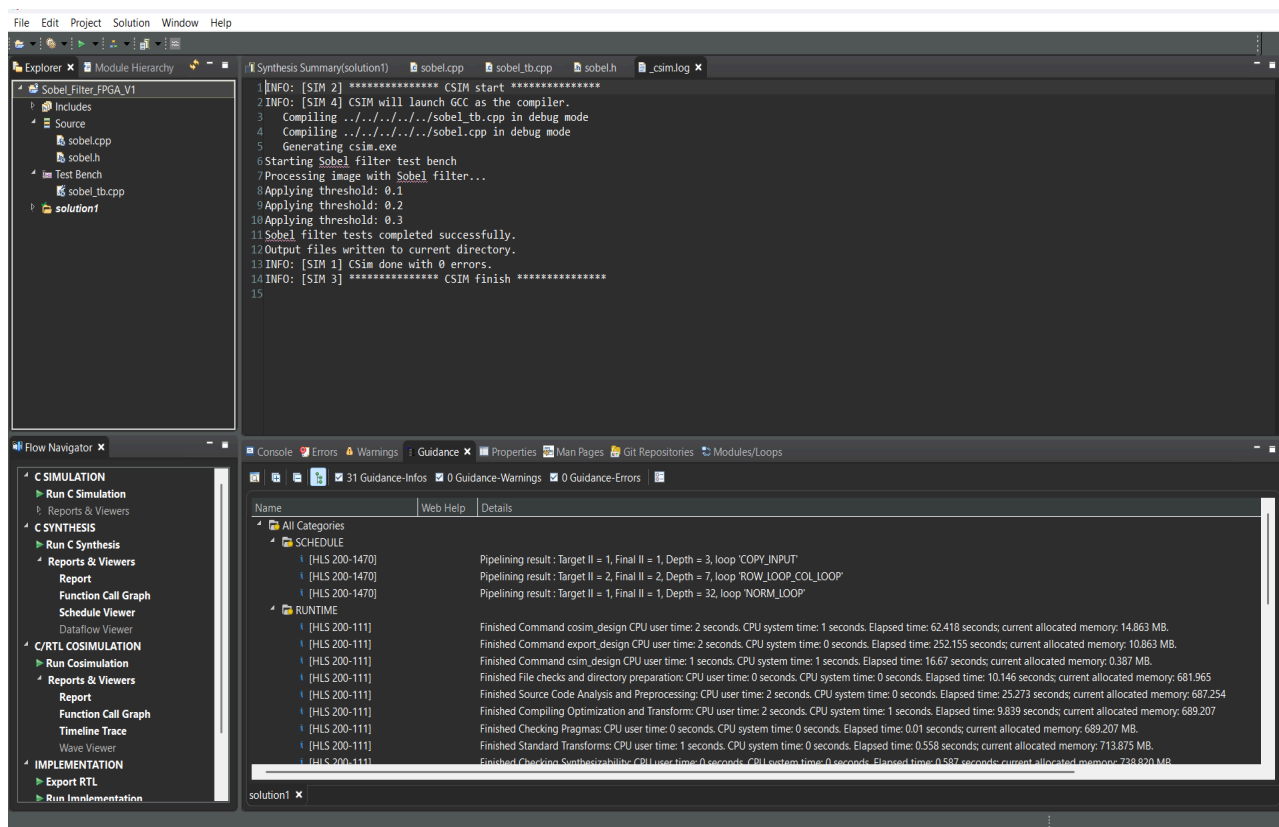
- Explorer:** Shows the project structure for 'Sobel\_Filter\_FPGA\_V1', including 'Includes', 'Source', and 'Test Bench' folders. The 'Test Bench' folder contains 'sobel\_tb.cpp' and 'solution1'.
- Flow Navigator:** Lists various analysis and synthesis tools, including 'Run C Synthesis', 'Reports & Viewers', 'C/RTL COSIMULATION', and 'IMPLEMENTATION'.
- Main Editor:** Displays the code for 'sobel\_tb.cpp'. The code includes headers for 'sobel.h', 'iostream', 'fstream', 'cstring', and 'iomanip'. It defines two functions: 'read\_image\_from\_file' and 'write\_image\_to\_file'. The 'read\_image\_from\_file' function attempts to open a file and read pixel data into an array. The 'write\_image\_to\_file' function writes the pixel data back to a file with fixed precision. The code uses 'IMAGE\_HEIGHT' and 'IMAGE\_WIDTH' constants.
- Outline:** Provides a summary of the code structure, listing the functions and their parameters.
- Console:** Shows the output of the CDT Build Console for 'Sobel\_Filter\_FPGA\_V1'.

```
1 #include "sobel.h"
2 #include <iostream>
3 #include <fstream>
4 #include <cstring>
5 #include <iomanip>
6
7 // Function to read an image from a file
8 bool read_image_from_file(const char* filename, pixel_data input_image) {
9     std::ifstream file(filename);
10    if (!file.is_open()) {
11        std::cerr << "Error: Could not open file " << filename << std::endl;
12        return false;
13    }
14
15    for (int i = 0; i < IMAGE_HEIGHT * IMAGE_WIDTH; i++) {
16        float pixel_value;
17        if (!(file >> pixel_value)) {
18            std::cerr << "Error reading pixel at index " << i << std::endl;
19            file.close();
20            return false;
21        }
22        input_image[i] = pixel_value;
23    }
24
25    file.close();
26    std::cout << "Successfully loaded image from " << filename << std::endl;
27    return true;
28 }
29
30 void write_image_to_file(const char* filename, result_data output_image) {
31     std::ofstream file(filename);
32     if (!file.is_open()) {
33         std::cerr << "Error: Could not open file " << filename << " for writing" << std::endl;
34         return;
35     }
36
37     // Write image data with fixed precision
38     file << std::fixed << std::setprecision(6);
39     for (int i = 0; i < IMAGE_HEIGHT; i++) {
40         for (int j = 0; j < IMAGE_WIDTH; j++) {
41             int index = i * IMAGE_WIDTH + j;
42             if (j > 0) file << " ";
43             file << float(output_image[index]);
44         }
45     }
```

## HLS Simulation, Verification, RTL Export and Report Analysis:

Following the building and creation of the Sobel filter algorithm using C++, a full series of simulation and synthesis steps were carried out using Xilinx Vitis HLS to establish correctness, examine performance, and ready the design for implementation in hardware. This phase merges the results of the next phases: C Simulation, C Synthesis, Co-Simulation, and RTL Export.

**C Simulation:** The first half involved simulating the C++ version to verify that the algorithm was giving accurate results when applied to sample MNIST images. The testbench (sobel\_tb.cpp) read in a sample image, executed the filter, and printed out the output values for comparison.



## C Simulation Report

**C Synthesis:** The second step was to take the validated C++ implementation and convert it to Register Transfer Level (RTL) hardware in Vitis High-Level Synthesis (HLS). In this step, synthesis reports were created that gave us estimates of latency, initiation interval (II), and FPGA resource utilization. This is to verify the validity and correctness of the algorithm before hardware synthesis.

The screenshot displays the Vitis IDE interface with the 'Synthesis Summary Report of 'sobel\_filter'' open. The report is divided into several sections: General Information, Timing Estimate, Performance & Resource Estimates, Performance Pragma, and HW Interfaces.

**General Information**

- Date: Sat Jun 21 15:31:38 2025
- Version: 2022.2 (Build 3670227 on Oct 13 2022)
- Project: Sobel\_Filter\_FPGA\_V1
- Solution: solution1 (Vivado IP Flow Target)
- Product family: zynq
- Target device: xc7z020-clg484-1

**Timing Estimate**

Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

**Performance & Resource Estimates**

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSB	FF	LUT	URAM
sobel_filter	-	-	-	-	3189	3.189E4	-	3190	-	no	8	0	4016	7833	0
sobel_filter_Pipeline_COPY_INPUT	-	-	-	-	787	7.870E3	-	787	-	no	0	0	57	170	0
sobel_filter_Pipeline_ROW_LOOP_COL_LOOP	-	-	-	-	1574	1.574E4	-	1574	-	no	0	0	599	1410	0
sobel_filter_Pipeline_NORM_LOOP	-	-	-	-	815	8.150E3	-	815	-	no	0	0	1442	1025	0

**Performance Pragma**

Modules & Loops	Target TI(cycles)	TI(cycles)	TI me
sobel_filter	-	-	-
sobel_filter_Pipeline_COPY_INPUT	-	-	-
sobel_filter_Pipeline_ROW_LOOP_COL_LOOP	-	-	-
sobel_filter_Pipeline_NORM_LOOP	-	-	-

**HW Interfaces**

Console | Errors | Warnings | Guidance | Properties | Man Pages | Git Repositories | Modules/Loops

31 Guidance-Infos | 0 Guidance-Warnings | 0 Guidance-Errors

## C Synthesis Report

**Co-Simulation:** Co-simulation was run to ensure that the RTL model produced the same behavior as the original C++ model. The comparison was done by comparing the output of both simulations at the bit level. This is to provide bit-for-bit correspondence in between the hardware and software models.

The screenshot displays the Vivado IDE interface with the 'Co-simulation Report(solution1)' open. The report is titled 'Cosimulation Report for 'sobel\_filter'' and is divided into several sections: General Information, Cosim Options, and Performance Estimates.

**General Information:**

- Date: Sat Jun 21 15:33:33 CST 2025
- Version: 2022.2 (Build 3670227 on Oct 13 2022)
- Project: Sobel\_Filter\_FPGA\_V1
- Status: Pass
- Solution: solution1 (Vivado IP Flow Target)
- Product family: zynq
- Target device: xc7z020-clg484-1

**Cosim Options:**

- Tool: Vivado XSIM
- RTL: Verilog

**Performance Estimates:**

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
sobel_filter	4658	4658	4658	4629	4629	4629
sobel_filter_Pipeline_COPY_INPUT	4658	4658	4658	789	789	789
sobel_filter_Pipeline_ROW_LOOP_COL_LOOP	4658	4658	4658	1572	1572	1572
sobel_filter_Pipeline_NORM_LOOP	4658	4658	4658	2255	2255	2255

The bottom of the interface shows the 'Console' tab with 31 Guidance-Infos, 0 Guidance-Warnings, and 0 Guidance-Errors. The 'Flow Navigator' on the left lists various synthesis and implementation steps, with 'Run C Synthesis' and 'Run Cosimulation' being the most relevant for this report.

## Co-Simulation Report

**RTL Export:** Upon successful verification and synthesis, the final step was exporting the design into an IP core. The IP core, along with Verilog/VHDL files, interface definitions, and configuration metadata, was packaged for use in Vivado. This is to develop the synthesized architecture to incorporate in the programmable logic (PL) of the Zynq-7000 system-on-chip (SoC) on the ZedBoard platform. The exported IP was later imported into Vivado for system-level integration and bitstream generation.

### Generated Files:

- i. RTL (Verilog/VHDL) code
- ii. IP metadata (component.xml)

The screenshot shows the Vivado IDE interface with the 'Export Report for sobel\_filter' window open. The window is divided into several sections:

- General Information:**
  - Report date: Sat Jun 21 15:44:34 +0800 2025
  - Project: Sobel\_Filter\_FPGA\_V1
  - Solution: solution1
  - Device target: xc7z020-clg484-1
  - Implementation tool: Xilinx Vivado v.2022.2
- Run Constraints & Options:**

Name	Value
Design Constraints & Options	
RTL Synthesis Options	
Reporting Options	
- Resource Usage:**

	Verilog
SLICE	0
LUT	2698
FF	3423
DSP	0
BRAM	11
URAM	0
LATCH	0
SRL	360
CLB	0
- Final Timing:**

	Verilog
CP required	10.000
CP achieved post-synthesis	6.311

Timing met
- Resources:**

Name	LUT	FF	DSP	BRAM	URAM	SRL	Pragma	Impl	Latency	Variable	Source

The bottom of the window shows a console with 31 Guidance-Infos, 0 Guidance-Warnings, and 0 Guidance-Errors.

### Export Report

## 6. FPGA Hardware Design in Vivado

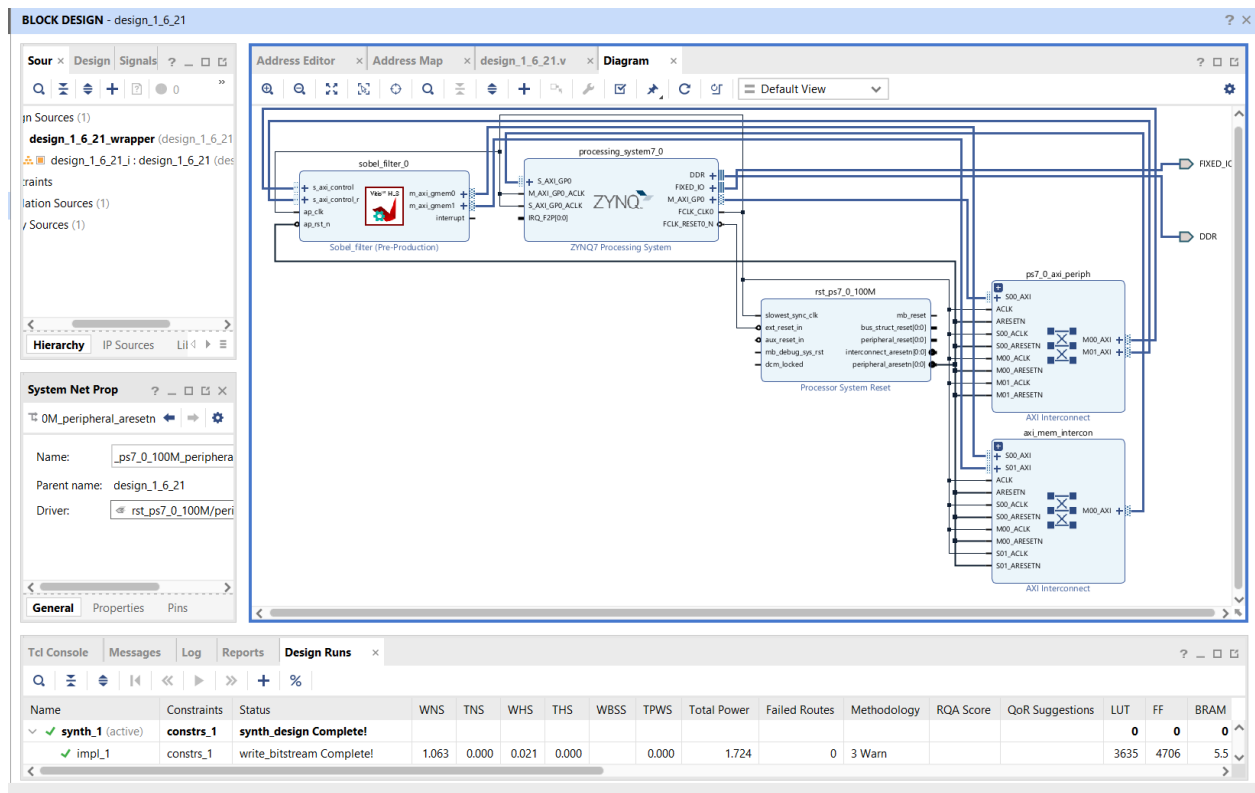
The goal of this phase was to integrate the custom Sobel filter synthesized as an IP core using Vitis HLS into a complete hardware platform using Xilinx Vivado. The platform was built on the **ZedBoard (xc7z020clg484-1)**. The procedure that was used to implement and design the hardware system is as follows:

- i. **Vivado Project Creation:** A new project in Vivado was created, and the ZedBoard (ZYNQ-7000 SoC, part number XC7Z020CLG484-1) was targeted. Default board presets were enabled to simplify the configuration of peripherals and the processing system.
- ii. **Importing the Sobel HLS IP Core:** The custom Sobel edge detection module, previously exported from Vitis HLS, was added to the Vivado IP catalog and instantiated within the block design.
- iii. **Including the ZYNQ7 Processing System (PS7):** The ZYNQ7 PS block was included to serve as the central controller. Configuration settings were updated to:
  - Enable UART1 for serial communication with a host PC.
  - Enable DDR3 RAM for storing MNIST image data.
- iv. **Connected the Sobel IP to the PS7 via AXI-Lite and AXI interfaces:** AXI Interconnects were used to route communication between the PS and the Sobel IP block. Proper memory address ranges were assigned to the IP core using the Address Editor to ensure software-level access via Vitis.
- v. **Validated the design and generated the bitstream:** Once the block design was completed, the system was validated using Vivado's integrated tools to ensure all connections were correctly made and that the design met interface and timing requirements. Any missing connections or unassigned addresses were resolved during this phase.

After successful validation:

- The block design was wrapped in a top-level HDL wrapper.
- Synthesis and Implementation were performed.
- The final bitstream file (.bit) was generated and exported along with the hardware platform description (.xsa) for use in the Vitis software development environment (Hostcode).

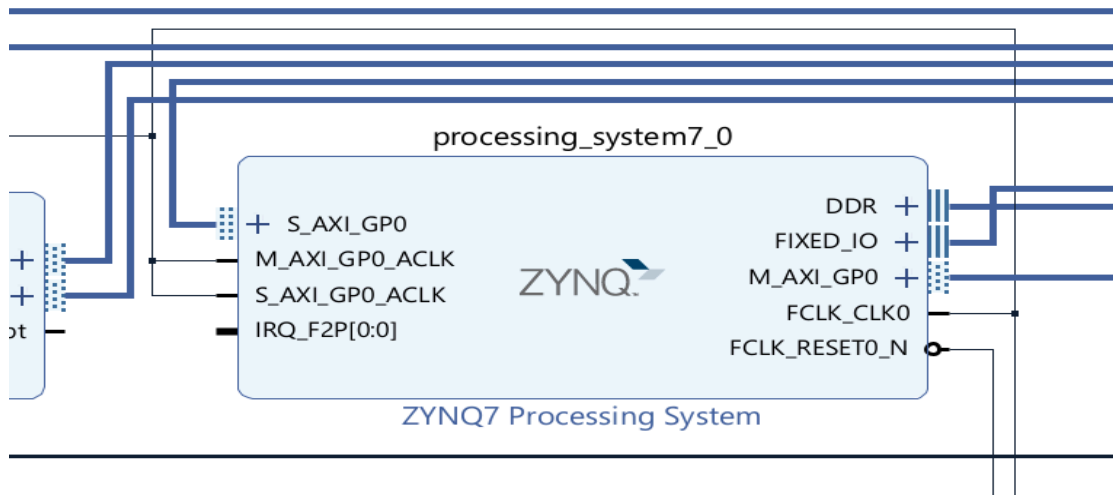
## Block Design:



## IP Block Diagram

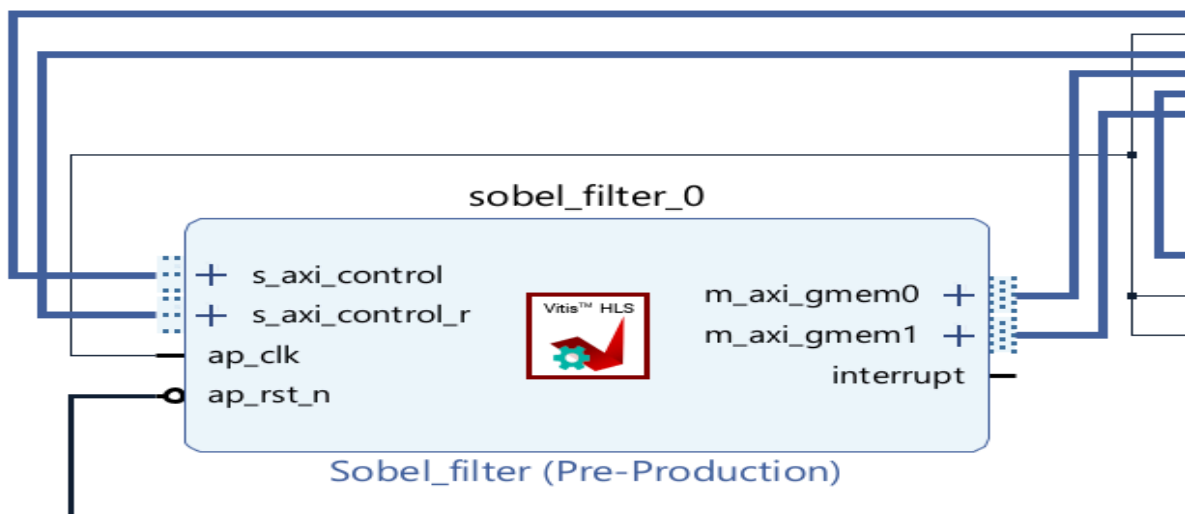


- i. **processing\_system7\_0 – ZYNQ Processing System (PS7):** Manages control, memory access, and data I/O.



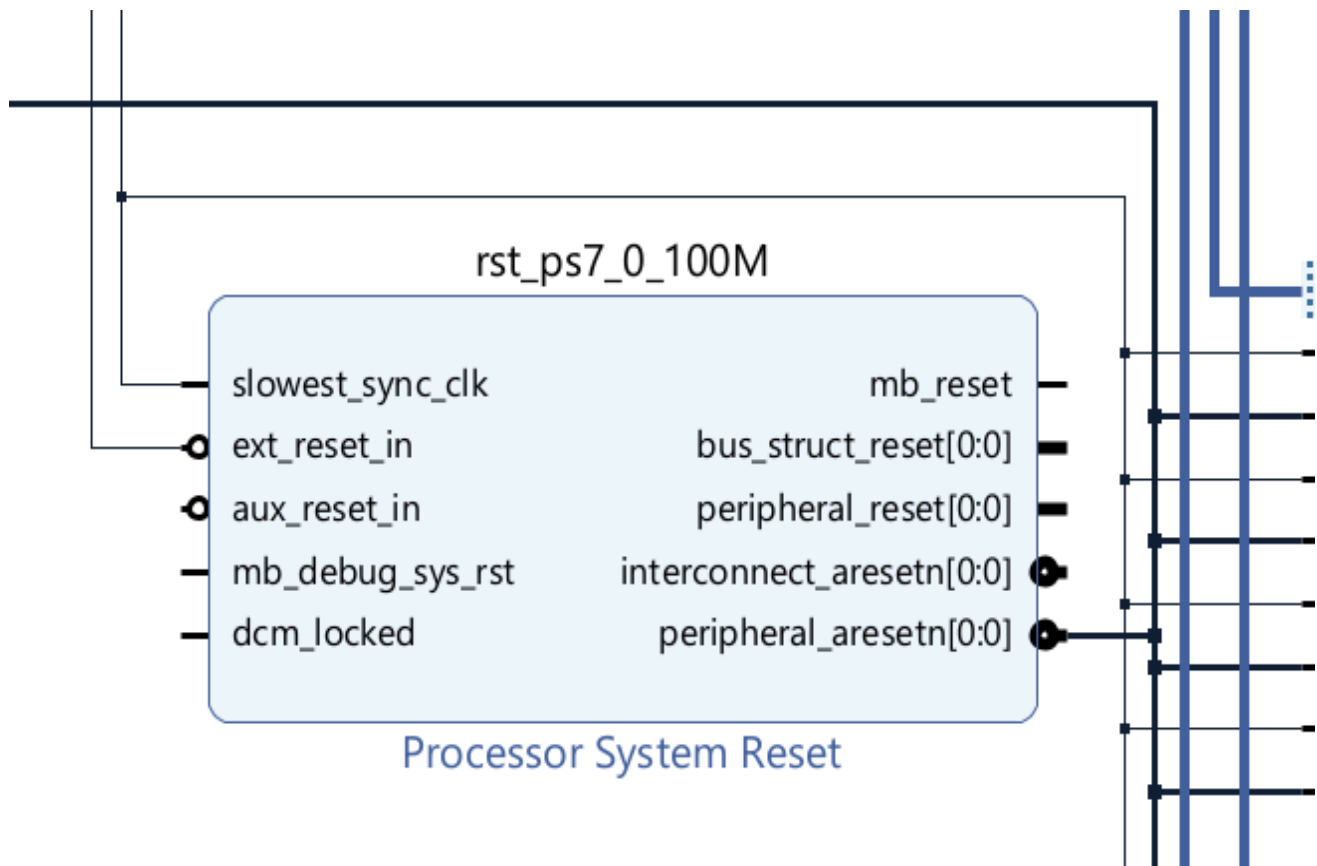
**ZYNQ Processing System (PS7)**

- ii. **sobel\_filter\_0 - Custom HLS IP Core:** Performs edge detection on images using the Sobel filter.



**Custom HLS IP Core**

- iii. **rst\_ps7\_0\_100M** – **Processor System Reset Block:** Handles system reset synchronization.

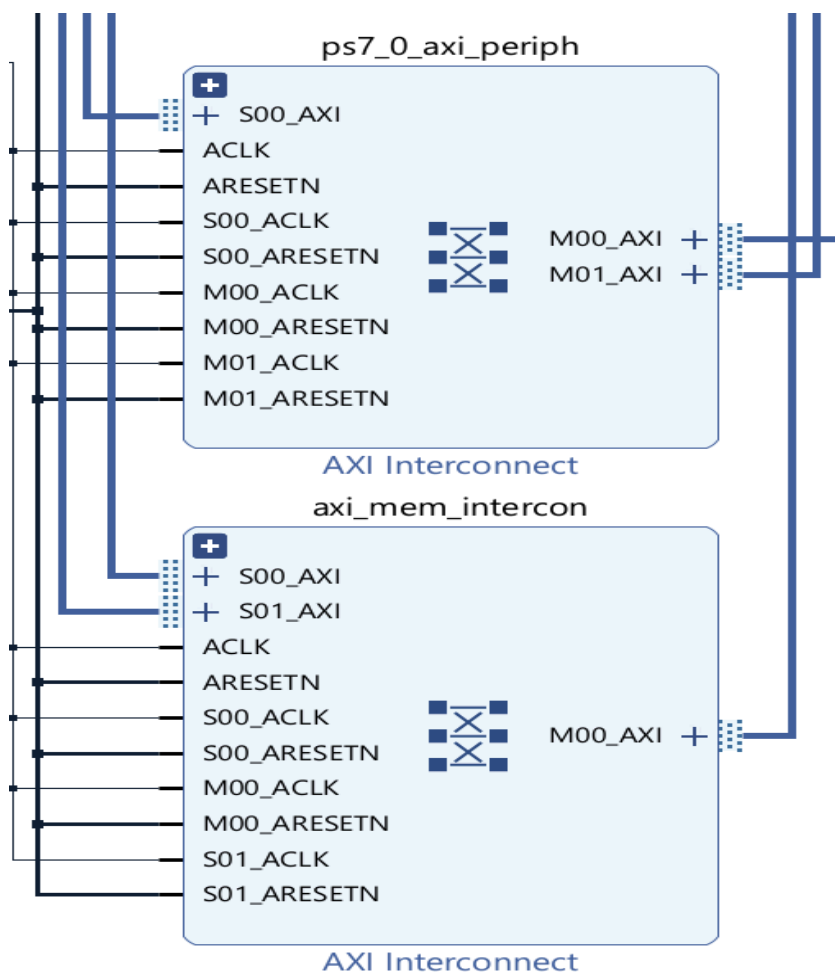


**Processor System Reset Block**

iv. **AXI Interconnect Blocks:** There are two AXI interconnects in this design:

- Ps7\_0\_axi\_periph: Routes AXI-Lite transactions from PS7 to the Sobel IP control interface.
- Axi\_mem\_intercon: Routes AXI full transactions for image data transfer between DDR and the Sobel IP (m\_axi\_gmem0).

They enable seamless communication between PS and PL, with address decoding and port arbitration handled internally.



**AXI Interconnect Blocks**

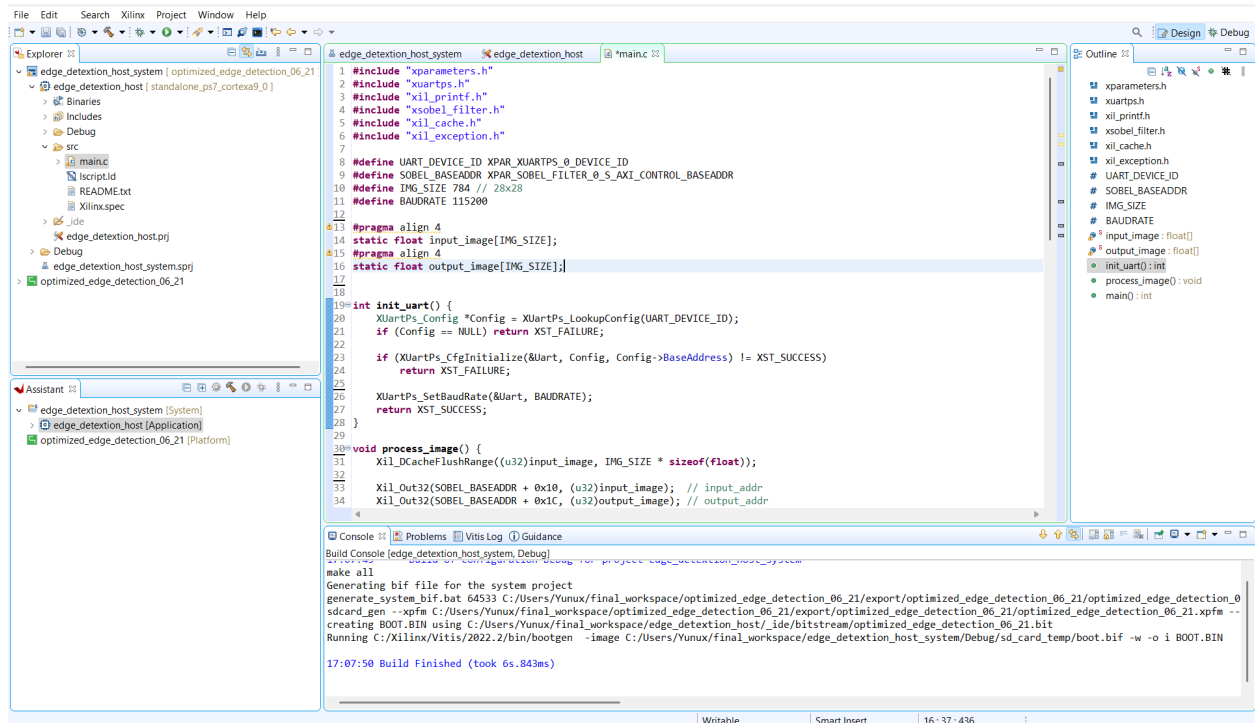
## 7. Integration of Software-Hardware in Vitis IDE (Host Code)

The main goal of this phase was to create host-side software that is designed to facilitate data communication between the ARM Processing System and the Sobel hardware accelerator realized in the programmable logic (PL). The software reads MNIST images, loads them to the FPGA, initializes the Sobel filter module, and gets back resulting edge maps for visualization

### Process:

- i. **Exporting the Hardware Platform:** The hardware design, including the Sobel IP core and processing system, was exported from Vivado as an **.xsa file**. This file describes the platform's hardware structure and is used to configure the Vitis workspace.
- ii. **Creating the Vitis Application Project:** A new application project was created in the Vitis IDE, targeting the exported hardware platform. The project was based on a C application template designed for bare-metal execution on the Zynq ARM core.
- iii. **Writing the Host Application Code:** The host application was redesigned to facilitate PC-to-FPGA communication via UART while maintaining hardware acceleration. The software performs these critical tasks:
  - i. **Image Acquisition via UART:** MNIST images are transmitted from the PC GUI as serialized 32-bit floating-point arrays. The host code reconstructs these into 28×28 matrices in aligned DDR memory.
  - ii. **Memory Management:** Image buffers are allocated in DDR with explicit cache control. The ARM processor flushes input data to ensure coherence before acceleration and invalidates output buffers to retrieve processed results.
  - iii. **Sobel IP Configuration:** The accelerator is programmed through AXI-Lite registers:
    - Input/output buffer addresses are set to shared DDR locations
    - Threshold values are configured in normalized fixed-point format
    - Processing is initiated via a start command
  - iv. **Execution Monitoring:** The host polls the IP's status register to detect completion, ensuring synchronization between software and hardware execution.
  - v. **Result Retrieval:** Processed edge maps are read from DDR and transmitted back to the PC via UART, enabling visualization and validation on the host machine.

## Code Snippet:



The screenshot displays the Xilinx IDE interface. The main editor shows the source code for 'edge\_detection\_host\_system' in the file 'main.c'. The code includes headers for 'xparameters.h', 'xuartps.h', 'xil\_printf.h', 'xsobel\_filter.h', 'xil\_cache.h', and 'xil\_exception.h'. It defines constants for UART device ID, Sobel base address, image size (784x28), and baud rate (115200). The code implements functions for initializing UART, processing the image using Sobel filter, and printing the result.

```
1 #include "xparameters.h"
2 #include "xuartps.h"
3 #include "xil_printf.h"
4 #include "xsobel_filter.h"
5 #include "xil_cache.h"
6 #include "xil_exception.h"
7
8 #define UART_DEVICE_ID XPAR_XUARTPS_0_DEVICE_ID
9 #define SOBEL_BASEADDR XPAR_SOBEL_FILTER_0_S_AXI_CONTROL_BASEADDR
10 #define IMG_SIZE 784 // 28x28
11 #define BAUDRATE 115200
12
13 #pragma align 4
14 static float input_image[IMG_SIZE];
15 #pragma align 4
16 static float output_image[IMG_SIZE];
17
18
19 int init_uart() {
20     XUartPs_Config *Config = XUartPs_LookupConfig(UART_DEVICE_ID);
21     if (Config == NULL) return XST_FAILURE;
22     if (XUartPs_CfgInitialize(&Uart, Config, Config->BaseAddress) != XST_SUCCESS)
23         return XST_FAILURE;
24     XUartPs_SetBaudRate(&Uart, BAUDRATE);
25     return XST_SUCCESS;
26 }
27
28 void process_image() {
29     Xil_DCacheFlushRange((u32)input_image, IMG_SIZE * sizeof(float));
30     Xil_Out32(SOBEL_BASEADDR + 0x10, (u32)input_image); // input_addr
31     Xil_Out32(SOBEL_BASEADDR + 0x1C, (u32)output_image); // output_addr
32 }
```

The console output shows the build process for the 'edge\_detection\_host\_system' project. It includes commands for generating the system BIF file, creating the BOOT.BIN, and running the bootgen tool to generate the boot.bif file.

```
Build Console [edge_detection_host_system, Debug]
17:07:50 64533 C:/Users/Yunux/final_workspace/edge_detection_host_system
make all
Generating bif file for the system project
generate_system_bif.bat 64533 C:/Users/Yunux/final_workspace/edge_detection_host_system/export/edge_detection_06_21/export/edge_detection_06_21/export/edge_detection_06_21.xpfm --
creating BOOT.BIN using C:/Users/Yunux/final_workspace/edge_detection_host_system/ide/bitstream/edge_detection_06_21.bif
Running C:/Xilinx/Vitis/2022.2/bin/bootgen -image C:/Users/Yunux/final_workspace/edge_detection_host_system/Debug/sd_card_temp/boot.bif -w -o i BOOT.BIN
17:07:50 Build Finished (took 6s.843ms)
```

## Hostcode

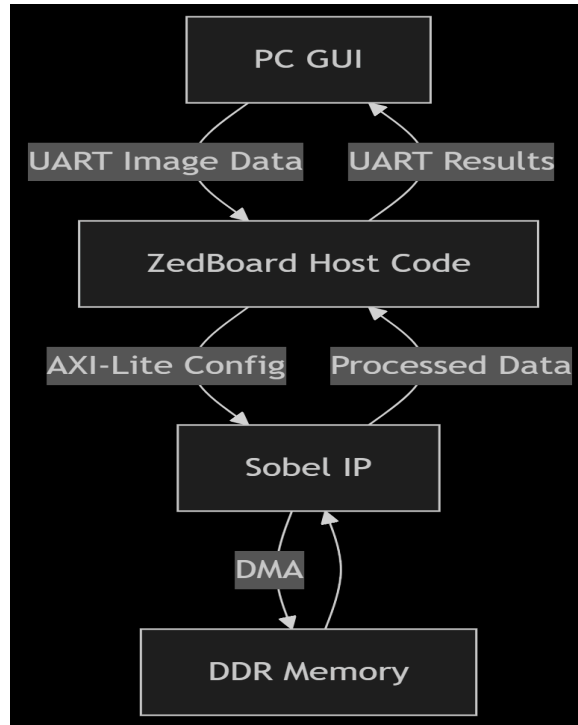
## 8. GUI Development

The goal of this stage was to design a simple and interactive graphical user interface (GUI) that allows users to select MNIST images, configure Sobel filter parameters, and visualize both the original and processed images. The GUI acts as a bridge between the user and the FPGA system, simplifying demonstration and testing.

### Process:

A lightweight GUI was developed in Python using the Tkinter library, which is well-suited for rapid prototyping and cross-platform compatibility. The interface was designed to integrate seamlessly with the host software and hardware setup. The GUI provides the following core functionalities:

- i. **Dataset Access and Image Selection:** The application loads the MNIST dataset from local storage and presents a user-friendly way to browse or randomly select a digit image (28×28 pixels).
- ii. **Parameter Configuration:** Users can adjust key processing parameters such as the **Sobel threshold**, allowing control over edge sensitivity. This value is later passed to the FPGA through the host code.
- iii. **Initiating FPGA Processing:** Upon clicking the **"Run FPGA"** button, the selected image is sent to the FPGA via the host software. The GUI communicates with the Vitis-built host application, which in turn configures the Sobel IP and retrieves the output.
- iv. **Result Visualization:** The GUI displays the edge-detected result returned by the FPGA. Images are rendered using the Matplotlib library, which integrates smoothly with Tkinter to deliver clear, high-quality visual output.



Execution Flow

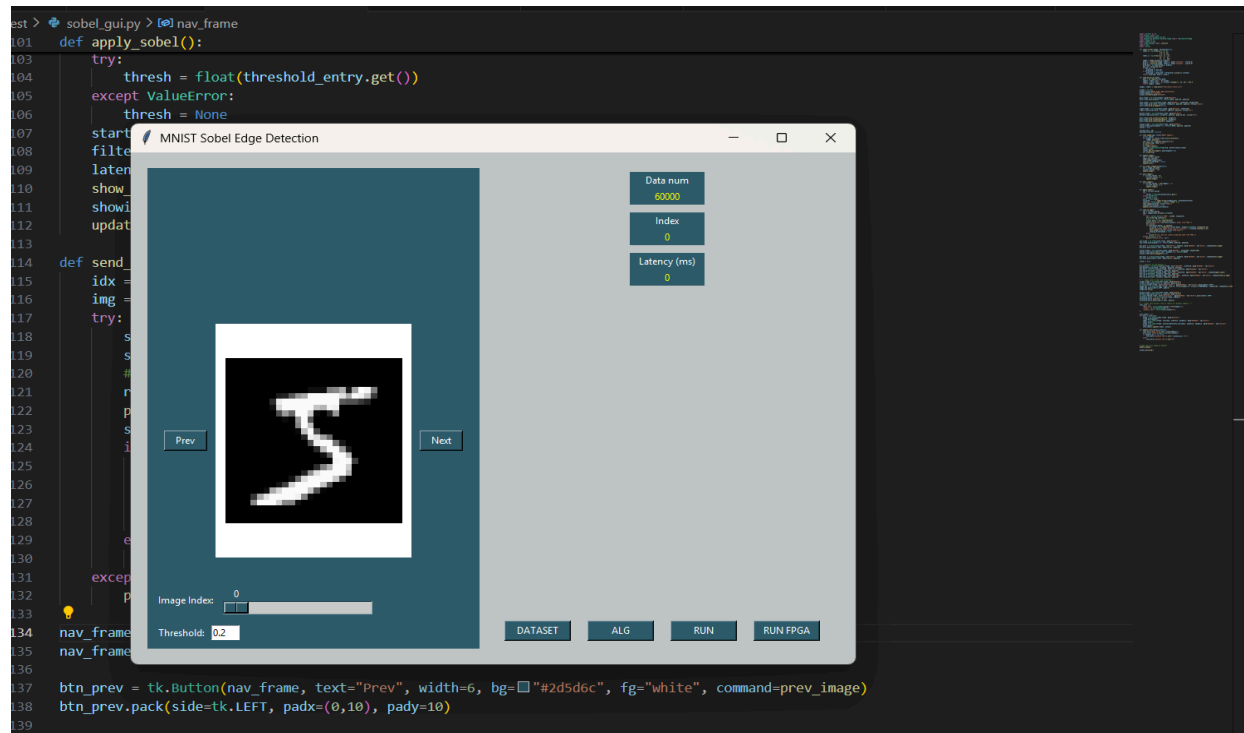
## Code Snippet (Python + Tkinter):

```

101 def apply_sobel():
102     start = time.time()
103     filtered, _, _ = sobel_filter(images[idx], threshold=thresh)
104     latency = (time.time() - start) * 1000 # ms
105     show_image(filtered, title="sobel")
106     showing_filtered[0] = True
107     update_info(latency_ms=latency)
108
109 def send_to_fpga():
110     idx = current_idx[0]
111     img = images[idx].astype(np.float32)
112     try:
113         with serial.Serial('COM3', 115200, timeout=5) as ser:
114             # Send image
115             ser.write(img.tobytes())
116             ser.flush()
117             # Receive result
118             result_bytes = ser.read(28*28*4)
119
120             if len(result_bytes) == 28*28*4:
121                 filtered = np.frombuffer(result_bytes, dtype=np.float32).reshape(28,28)
122                 show_image(filtered, title="FPGA Sobel")
123                 showing_filtered[0] = True
124             else:
125                 print(f"Received {len(result_bytes)} bytes (expected {28*28*4})")
126     except Exception as e:
127         print(f"communication error: {e}")
128
129 nav_frame = tk.Frame(left_frame, bg="#2d5d6c")
130 nav_frame.pack(expand=True, fill=tk.BOTH, padx=20, pady=20)
  
```

The code snippet shows the logic for sending image data to the FPGA and receiving results back. It uses the serial module to communicate with the FPGA via a serial port (COM3). The image data is converted to bytes and sent to the FPGA. The FPGA returns the processed image data as bytes, which are then converted back to a numpy array and displayed using the show\_image function. The code also includes error handling for communication errors.

## GUI Code



GUI displaying the edge-detected result



## Reference

- [1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed., Pearson, 2018.
- [3] Y. Zhang and V. K. Prasanna, “High-performance and energy-efficient image processing using FPGAs,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 23, no. 11, pp. 1810–1823, 2013.