FPGA-Based Image Processing Pipeline Simulator in C++

Hardware Accelerator Modeling
and Simulation Report

Author
Adil Rafiq

December 27, 2025

Contents

1 Abstract

This report provides a comprehensive technical overview of a C++ based simulator designed to model an FPGA-based image processing pipeline. It bridges the gap between high-level algorithmic development and low-level Register Transfer Level (RTL) implementation. Key features include the modeling of synchronous pipeline latches, fixed-point arithmetic optimization, and hardware-software co-design. This system serves as a model for verifying the accuracy of hardware implementation.

2 Introduction

Software modeling is a critical requirement before physical implementation in the domain of IC design and FPGA development. This simulator enforces the spatial and temporal constraints of ICs, specifically modeling data movement, resource utilization, and the synchronous nature of digital logic.

3 System Architecture: The Synchronous Pipeline

The simulator is designed as a **Synchronous 4-Stage Pipeline**. In a physical FPGA, all logic gates evaluate in parallel every clock cycle.

3.1 Cycle-Accurate Simulation Loop

A common challenge in software simulation is "race conditions" where data propagates through a chain in a single software loop. To resolve this, the simulator uses a central loop acting as a system clock. Within this loop, stages are executed in **Reverse Order** (Stage 4 to Stage 1).
By processing the end of the pipeline first, we ensure that data produced by Stage N in Cycle X is only available to Stage $N + 1$ in Cycle $X + 1$. This perfectly replicates the behavior of physical hardware registers (latches).

3.2 Pipeline Stage Definitions

- **Stage 1: Frame Reader:** Simulates the hardware physical layer reading from host memory into an input register (`reg_RawData`).
- **Stage 2: ISP Block:** Consumes `reg_RawData`, performs color conversion, and latches the result into `reg_GrayData`.
- **Stage 3: DSP Engine:** Processes grayscale data using spatial filters and latches the result into `reg_ProcessedData`.
- **Stage 4: Frame Writer:** Consumes `reg_ProcessedData` and writes the final frame back to memory.

3.3 Pipeline Data Flow Diagram

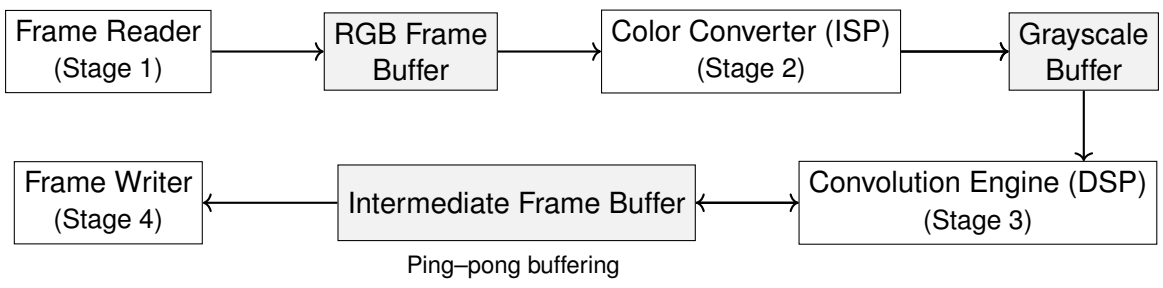

Figure 1: FPGA-based image processing pipeline.

3.4 Ping-Pong Buffering

In hardware convolution pipelines, a processing unit must never read from and write to the same memory region in a single cycle. The filters require access to neighboring pixels; overwriting input data prematurely would corrupt the computation.
To address this constraint, the simulator implements a **ping-pong buffering scheme**. Two frame buffers are alternately assigned as input and output to the DSP engine. After each filter operation, buffer roles are swapped, ensuring that every convolution stage reads from a stable data source and writes results to a separate memory region.
This approach minimizes memory usage while preserving correctness and mirrors the double-buffering strategy commonly used in FPGA image pipelines and DMA-based accelerators.

4 Modular Design and Memory Modeling

4.1 Object-Oriented Hardware Mapping

The project uses a hybrid of `structs` and `classes` to mimic hardware components:

- **Structs (Data Packets):** Types like `Pixel` and `Kernel` are implemented as `structs`. In hardware, these represent wires or data packets that carry signals across the bus.
- **Classes (IP Cores):** Modules like `ColorConverter` and `ConvolutionEngine` are implemented as `classes`. These represent independent Intellectual Property (IP) cores. They encapsulate the logic and internal state required to transform data.

4.2 Templates and Bus Width Flexibility

The `FrameBuffer` class uses C++ **Templates**. In hardware, memory controllers are often generic. By using `FrameBuffer<T>`, we simulate a unified memory interface that can handle different bus widths—such as 24-bit RGB or 8-bit Grayscale—using the same underlying logic.

4.3 Dynamic Allocation as Physical RAM

To simulate DDR RAM, the system utilizes `malloc` and `free`.

- `malloc`**:** This simulates the **Memory Arbiter** reserving a contiguous block of physical addresses. The resulting pointer acts as the **Base Address Register** used by DMA engines.
- `free`**:** This simulates the release of a memory bank or the draining of a FIFO after a frame has been successfully transmitted.

5 Mathematical Optimization for Silicon

The simulator supports two distinct modes of calculation, toggled via preprocessor macros (`USE_FLOAT` or `USE_FIXED_POINT`).

5.1 Floating-Point Reference

In floating-point mode, the ISP and DSP blocks use standard IEEE-754 math. This serves as a reference to determine the maximum possible image quality. Requires more complex calculations and Floating-Point Units (FPUs).

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{1}$$

5.2 Fixed-Point Hardware Accuracy

In fixed-point mode, the system uses integer coefficients scaled by 256 (2^8). The division is replaced by a right shift, which only involves bit-rewiring in hardware.

$$Y = (77 \cdot R + 150 \cdot G + 29 \cdot B) \gg 8 \tag{2}$$

6 DSP Architecture and Filter Hierarchy

The **Convolution Engine** is the main complex block in the system, managing multiple filters through a unified 3x3 Multiply-Accumulate (MAC) structure.

6.1 Pre-defined Filter Kernels

Filters are stored in `kernel.h` as static coefficient matrices. These represent the hard-wired constants in the hardware's look-up tables.

- **Box Blur (Always-On):** This filter is implemented as a mandatory pre-processing step. In real-world sensors, high-frequency noise is common; the Box Blur acts as a "noise floor" reduction unit, smoothing the signal before complex enhancement.
- **Gaussian and Sharpen:** These two filters share the same internal **Convolution Block**. Since both are 3x3 spatial operations, they utilize the same MAC logic units. The only difference is the coefficients loaded from the registers, demonstrating a reconfigurable hardware design.

6.2 The Sobel Block

Unlike the standard filters, **Sobel Edge Detection** is implemented in a separate logic block within the engine. This is because Sobel requires dual kernels (G_x and G_y) to be calculated simultaneously followed by a magnitude calculation:

$$|G| \approx |G_x| + |G_y| \tag{3}$$

7 Design Constraints and Simplifications

To maintain clarity and focus, several real-world hardware complexities were intentionally abstracted:

- Line buffers and sliding window generators are modeled implicitly rather than cycle-by-cycle.
- Memory access latency is assumed to be uniform and deterministic.
- Control logic for arbitration and interrupts is not modeled.

These simplifications allow the simulator to emphasize architectural correctness and numerical accuracy while remaining extensible for future enhancements.

8 Simulation Environment and Results

8.1 Linux Setup

The project was developed and tested on an Ubuntu 24.04 LTS environment.

8.2 Code Implementation

The following figures provide snippets of the main source code and the makefile build system.

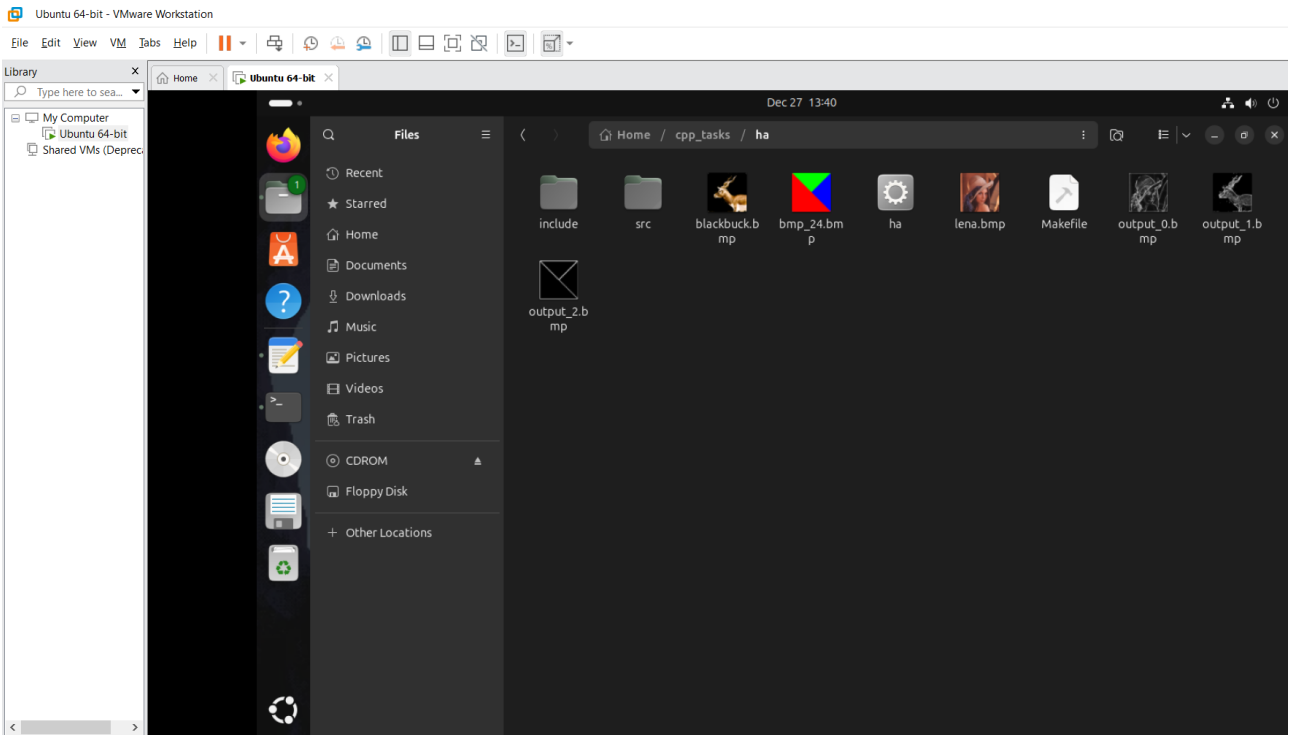

Figure 2: Linux environment and project directory.

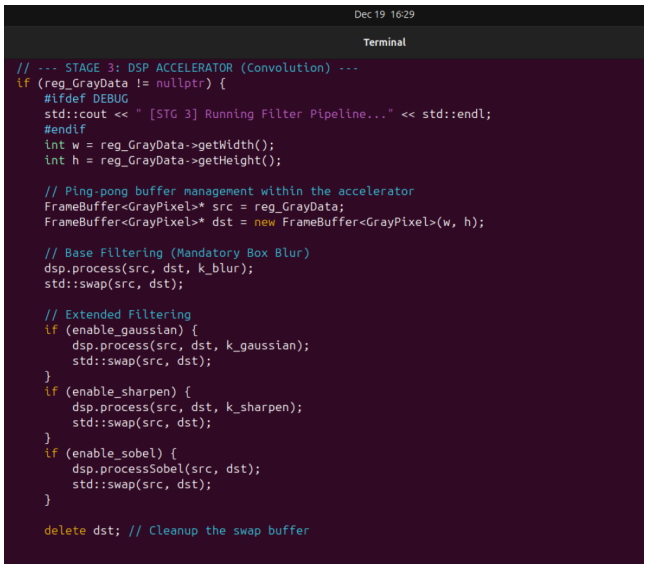

Figure 3: Main code (main.cpp).

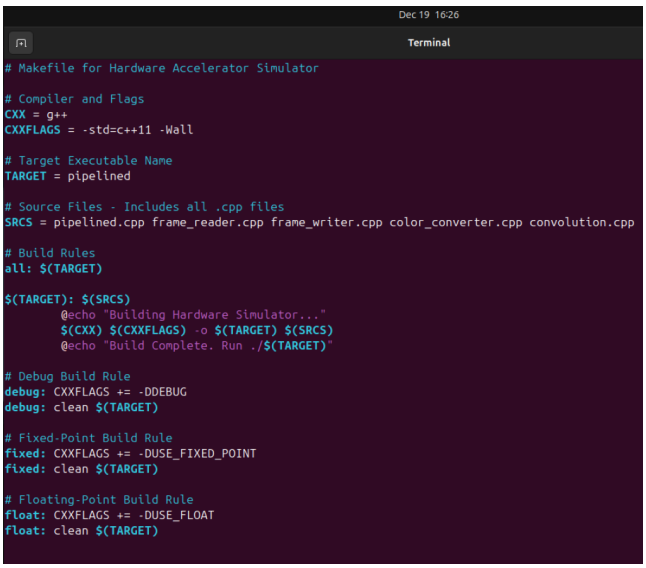

Figure 4: Project Makefile.

8.3 Execution Outputs

The simulator was verified across three primary build modes.

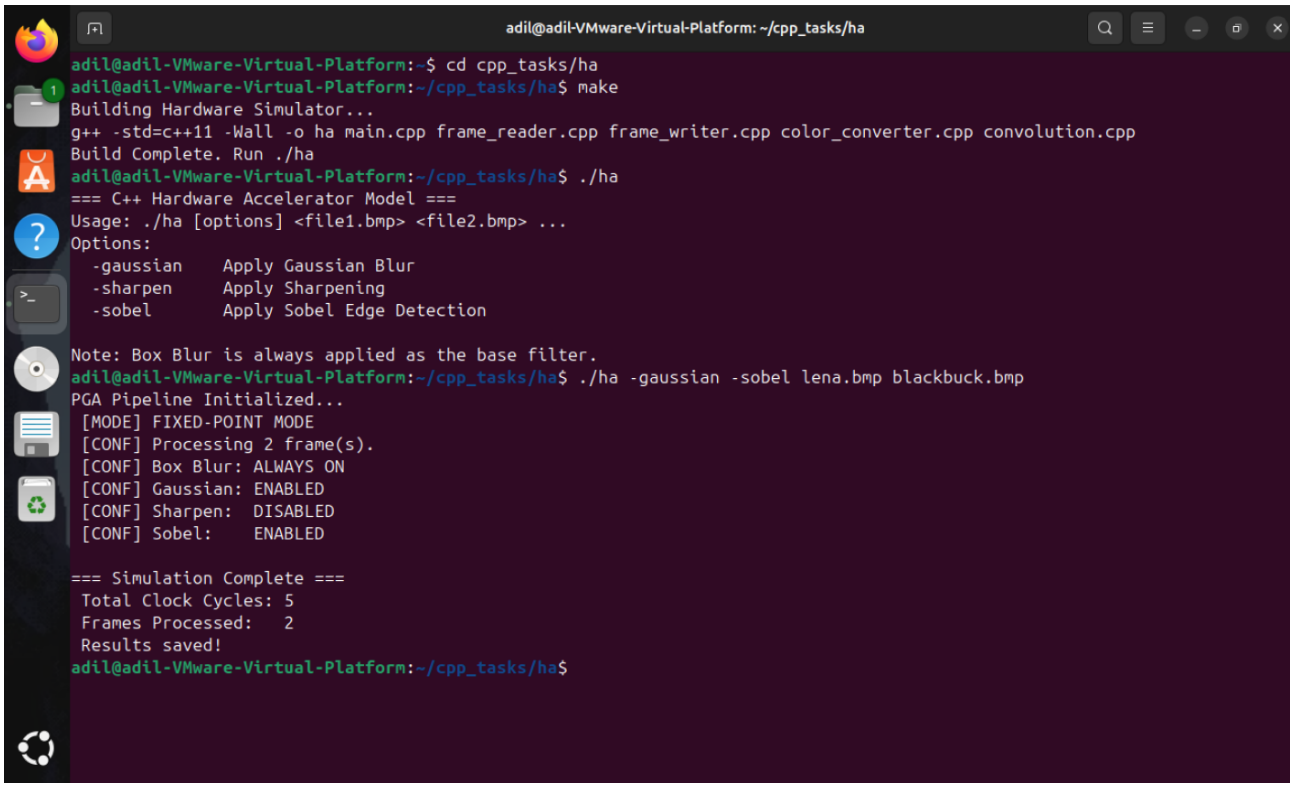

Figure 5: Normal (Fixed-Point) execution output.

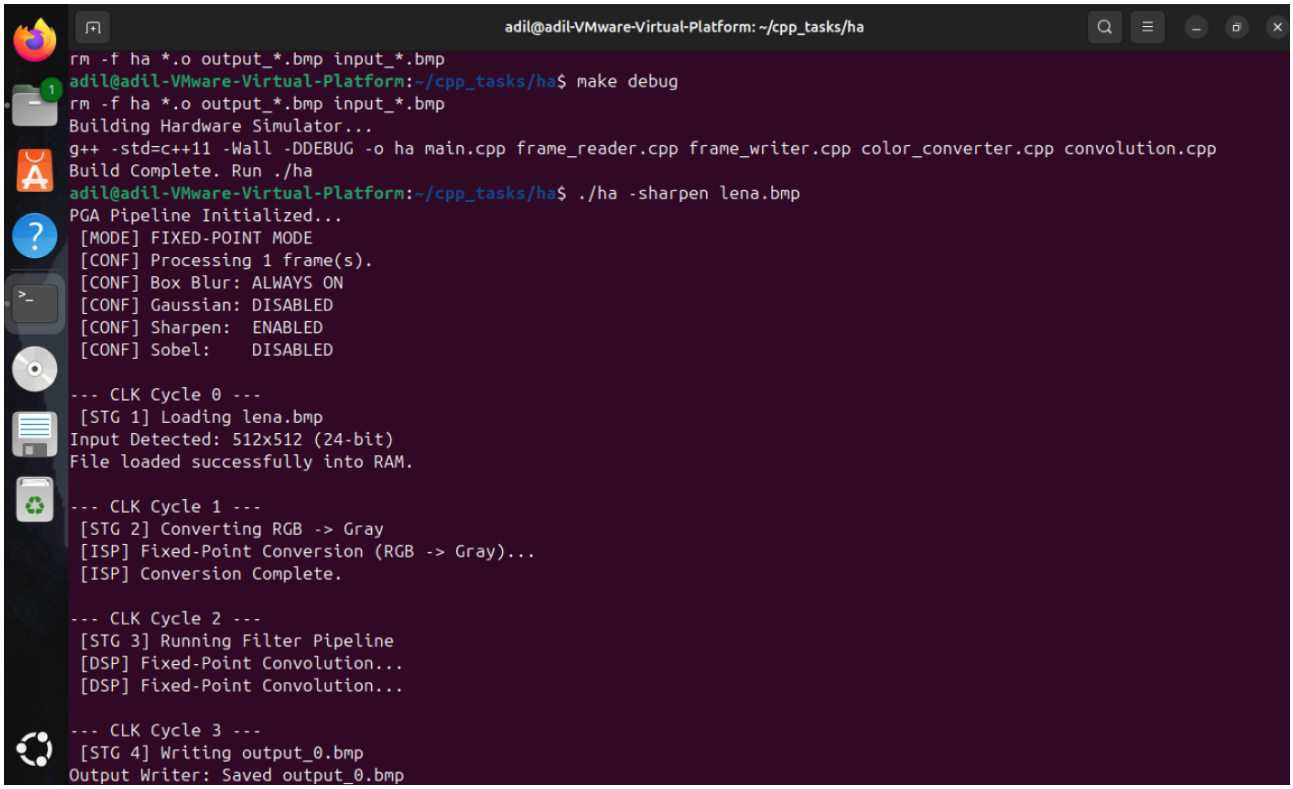

Figure 6: Execution with debug logging enabled.

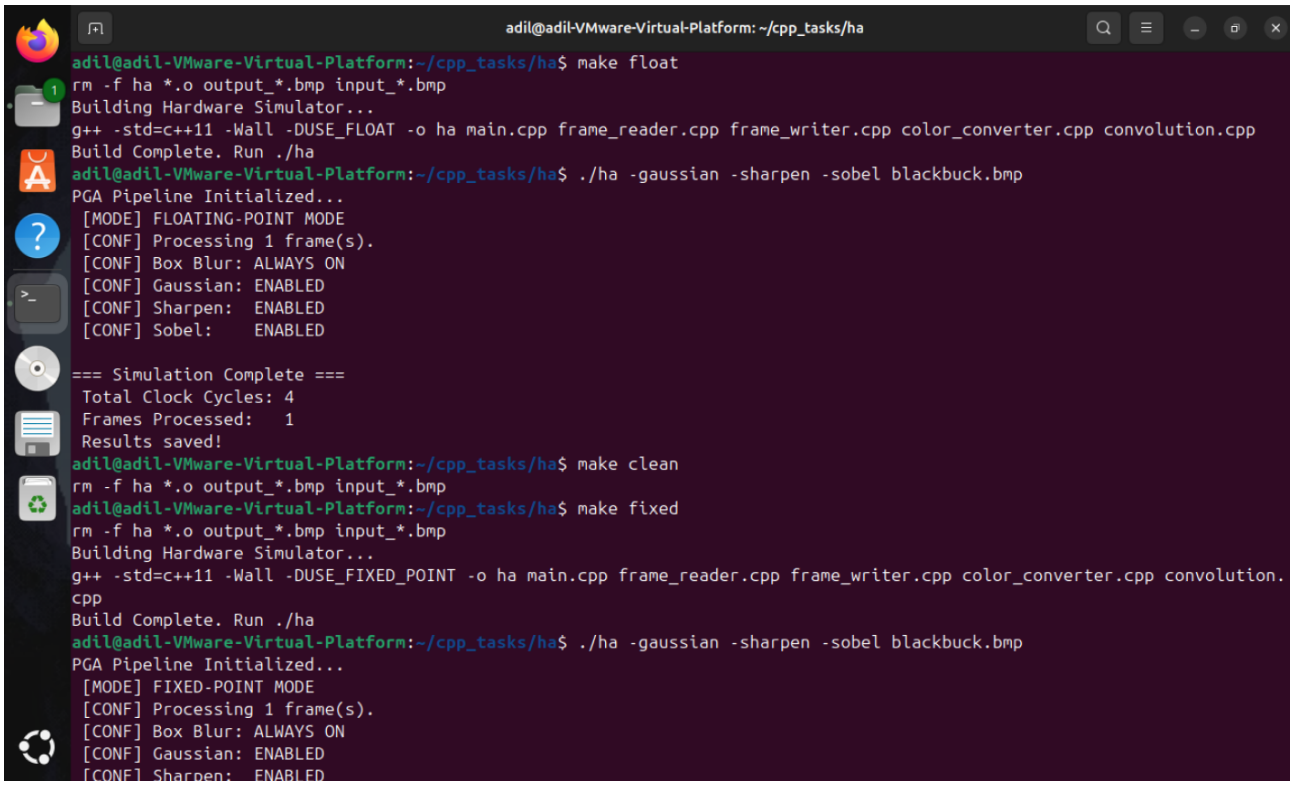

Figure 7: Floating-Point reference execution.

8.4 Results and Comparisons

Two input images, `lena.bmp` and `blackbuck.bmp`, were used for verification of output.

Figure 8: Comparison of input and output of `lena.bmp` for Sharpen filter.

Figure 9: Comparison of input and output of `blackbuck.bmp` for Gaussian and Sobel filters.

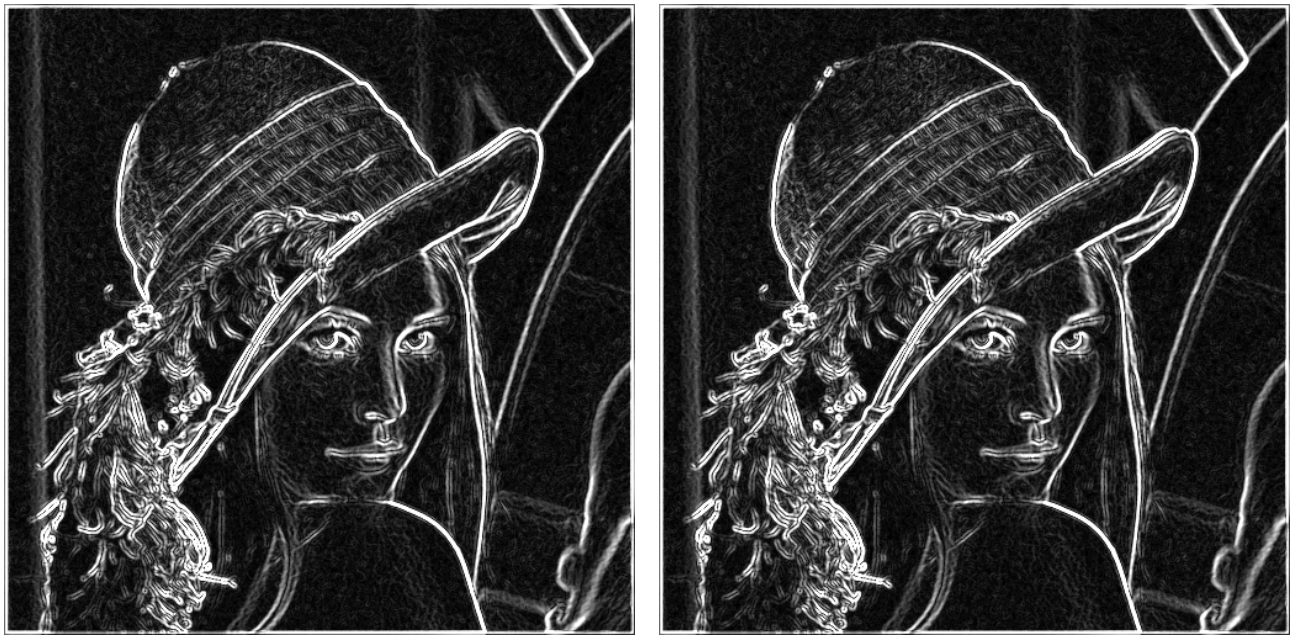

Figure 10: Comparison between Fixed-Point and Floating-Point results.

9 Conclusion

The FPGA Pipeline Simulator provides a robust framework for hardware-software co-design. By enforcing synchronous loops, template-based memory management, and dual-mode arithmetic, the system accurately predicts hardware performance. This documentation serves as the final specification for the hardware implementation phase.