Contents

1 Introduction 2

2 Methodology 2
2.1 System Architecture . . . 2
2.2 Tiling . . . 3
2.3 Layer implementation . . . 3

3 Experiments 5
3.1 Memory mapping . . . 6
3.2 Precision . . . 7
3.3 Compute size . . . 7
3.4 Tile size . . . 8

4 Discussion 8

5 Deployment 10
5.1 HLS Interface Pragma Directives and RTL/IP Integration Flow . . . 10
5.1.1 Interface Architecture and Design Rationale . . . 10
5.1.2 RTL IP Generation and Packaging . . . 10
5.2 System Integration Architecture . . . 11
5.2.1 Dataflow Architecture and Interface Mapping . . . 11
5.2.2 RTL Export Methodology . . . 12
5.2.3 Vivado Block Design Integration Architecture . . . 12
5.2.4 Interconnect Topology and Bandwidth Optimization . . . 12
5.2.5 Address Space Organization . . . 13
5.2.6 Clock Domain and Reset Architecture . . . 13
5.2.7 Bitstream Generation and Hardware Export . . . 13
5.2.8 Design Correctness and Performance Analysis . . . 14
5.3 Notebook Results . . . 14
5.3.1 Notebook Structure . . . 14
5.3.2 Results . . . 17

6 Conclusion 24

References 24

7 Appendix 25
7.1 Block Diagram . . . 25
7.2 Bitstream Diagram . . . 26

1 Introduction

The single image super resolution (SISR) is a computer vision research field, focusing on how to reconstruct a high resolution image from a low resolution input. Super Resolution Convolution Neural Network (SRCNN) uses deep convolutional neural networks (CNN) to improve the performance of SISR. However, this three-layers convolutional architecture requires intensive computation and memory costs (Gao & Zhou, 2023). This characteristic limits their applications on resource constrained devices. Therefore, this project aims to design and implement a high throughput and resource efficient SRCNN accelerator on FPGA, Kria K26 SOM.

Table 1 summarizes three convolutional layers in SRCNN model. Specifically, the first layer extracts input feature using 9x9 kerne, and then the second layer applies nonlinear mapping through 1×1 convolutions. Lastly, the final layer reconstructs the high-resolution output with a 5×5 kernel.

Layer	Kernel	Feature	Activation	Function
0		1		
1	9x9	64	ReLU	Feature Extraction
2	1x1	32	ReLU	Nonlinear Mapping
3	5x5	1	-	Reconstruction

Table 1: 3 layers comparison

For a 256×256 input, the multiply accumulate (MAC) per layer counts are:

$$\begin{aligned}
\mathrm{MAC}_1 &= 256^2 \times 64 \times 1 \times 9^2 \;=\; 3.40 \times 10^8 \\
\mathrm{MAC}_2 &= 256^2 \times 32 \times 64 \times 1^2 \;=\; 1.34 \times 10^8 \\
\mathrm{MAC}_3 &= 256^2 \times 1 \times 32 \times 5^2 \;=\; 1.05 \times 10^7
\end{aligned}$$

This considerable amount points out the heavy computational workload of the model. Therefore, the naive deployment approaches, such as keep entire feature maps and weights on chip or repeatedly move them, do not work in practice or efficiently. To address this gap, we proposed the tiling dataflow design. This implementation achieves initiation interval (II) of 1 to maximize throughput.

2 Methodology

This section presents the overall implementation strategy, including tiling, three-layer dataflow design, buffer management with line and window structures, and the read-compute-write (RCW) applied in conv2 and conv3.

2.1 System Architecture

The system is designed as a tile-based streaming architecture. Instead of processing the entire image at once, the input feature map is divided into smaller tiles. Within each tile, the three convolution layers operate in a dataflow pipeline, which allows each stage pass its intermediate results to the next one through FIFO buffers without processing the task completely. Figure 1 visualizes the overall structure of this dataflow implementation.

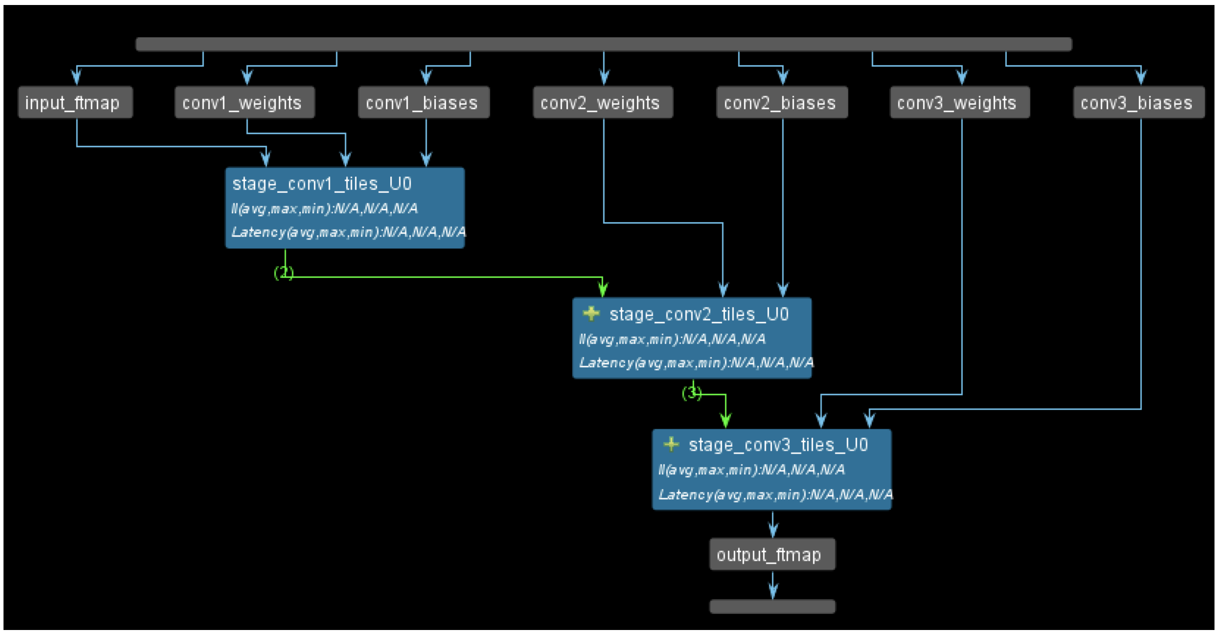
input_ftmap
conv1_weights
conv1_biases
conv2_weights
conv2_biases
conv3_weights
conv3_biases
stage_conv1_tiles_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
(2)
stage_conv2_tiles_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
(3)
stage_conv3_tiles_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
output_ftmap

Figure 1: Stage-level Data Flow

This strategy allow different stages to compute concurrently and thus enhances overall throughput. Specifically, the latency is then dominated by the slowest layer among the three, and the total latency can be estimated as:

$$L_{\text{total}} = L_{\text{fill}} + N_{\text{tiles}} \times \max(L_1, L_2, L_3) \tag{1}$$

, which is significantly lower than the sequential execution latency, $N_{\text{tiles}} \times \sum(L_1, L_2, L_3)$.

2.2 Tiling

To use on-chip resources efficiently, the entire input feature map is divided into smaller tiles of size `TILE_H` × `TILE_W`. The variables `th` and `tw` are used to determine the actual size of each tile at runtime to prevent boundary overflow. Each tile is processed independently, which allows intermediate data to be kept on-chip and reused during computation. This strategy improves data locality and reduces external memory access by ensuring every pixel only needs to be fetched from and written back to DRAM once.

2.3 Layer implementation

conv1 The first convolution layer operates a 9×9 kernel on each input tile. For every tile, the corresponding weights are preloaded into an on-chip cache (*w_cache*) using a row-major access pattern with the `ky` loop outside and `kx` loop inside. This ensures the innermost dimension is read sequentially from memory and the contiguous access pattern allows the AXI controller to form burst transactions, significantly improving memory bandwidth utilization.

On the other side, a line buffer stores the previous F1-1 rows of input pixels so that only one new pixel is fetched from external memory at each clock cycle. The window buffer is then updated by shifting existing values and inserting the new pixel, forming a sliding window for convolution. Partial results from all input channels are accumulated in `acc_tile`, and after adding the bias and applying the ReLU activation, the output values are written to the `S1` FIFO buffer.

conv2 The second convolution layer uses a 1×1 kernel to map 64-channel feature maps to 32 output channels. The main bottleneck is the intensive channel computation. Therefore, we apply the Read-Compute-Write (RCW) structure introduced by Yang, Xie, Yang, Liu, and Guan (2023). In this work, RCW separates the read, compute, and write stages to maximize overlap among sub-computations.

In our implementation, this layer-level dataflow pipeline first reads the feature from `S1` and store it at URAM `in_buf` to each pixel is fetched only once. Moreover, both the input data and weights are packed into blocks of size `Tn` and `Tm`×`Tn`, respectively. This allows fully pipeline across input and output channel groups. In the compute stage, multiply–accumulate (MAC) operations are performed and accumulated in `out_acc`. After the bias and ReLU activation are applied, the results are streamed out through the `S2` FIFO buffer for conv3. Figure 2 shows the data flow of this layer.

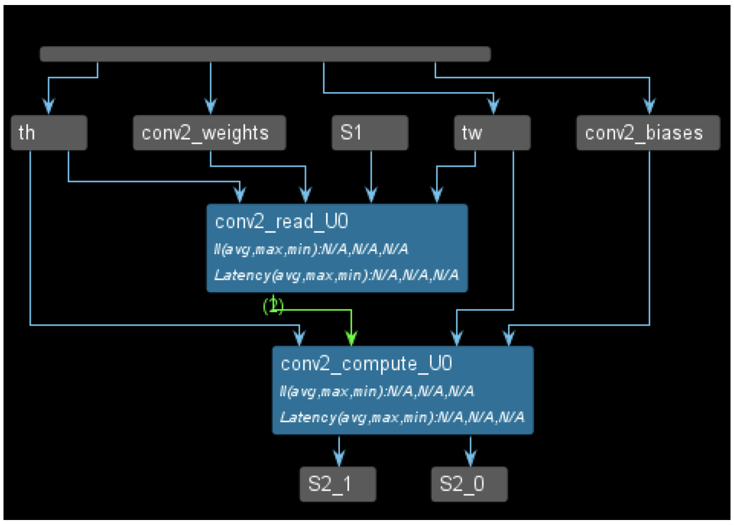
th
conv2_weights
S1
tw
conv2_biases
conv2_read_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
(1)
conv2_compute_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
S2_1
S2_0

Figure 2: Conv2 Data Flow

conv3 The last layer uses a 5×5 kernel to reconstruct the final single channel output from the 32 input feature maps. The main bottleneck in this layer is the large number of MAC across spatial and channel dimensions, which makes on-chip buffering and data reuse challenging.

To overlay the read and computation stages, we apply a RCW structure. In the read stage, only the weights are packed in small blocks *WPackK* of size `Tm`×`Tn`. This strategy supplies only necessary data for computation to prevent on-chip memory overuse. Meanwhile, the input feature from `S2` are buffered by multiple line buffers and window registers. A line buffer is assigned per input channel and each window maintains a 5×5 size by updating a new pixel in every clock cycle.

During computation, each output channel tile performs fully unrolled 5×5 MAC and accumulates results in `out_acc`. After all input channels have been processed, the accumulated results are added with the corresponding biases and written to the output feature map. Figure 3 shows the dataflow within this layer.

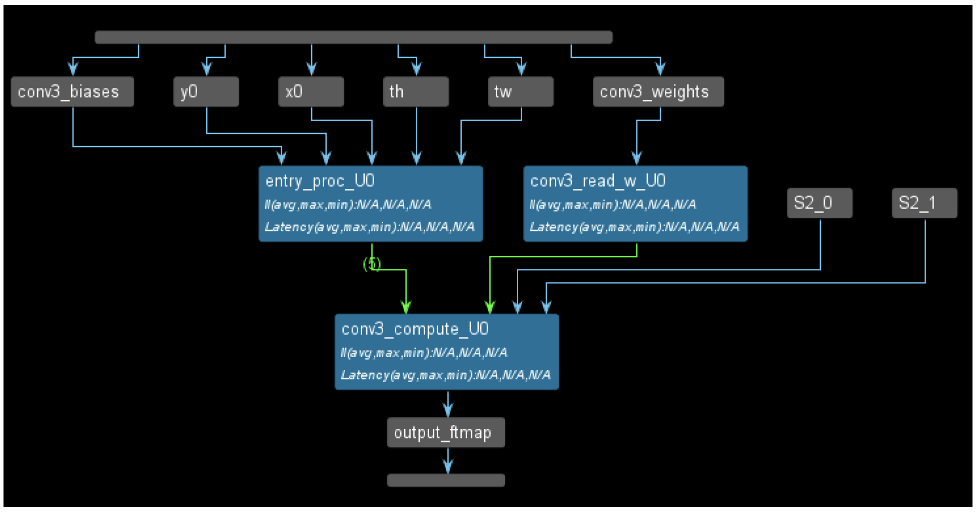
conv3_biases
y0
x0
th
tw
conv3_weights
entry_proc_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
conv3_read_w_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
S2_0
S2_1
(5)
conv3_compute_U0
II(avg,max,min):N/A,N/A,N/A
Latency(avg,max,min):N/A,N/A,N/A
output_ftmap

Figure 3: Conv3 Data Flow

3 Experiments

Table 2 summarizes the resource capacity of the Xilinx Kria K26 SOM (part: xck26-sfvc784-2LV-c). All experiments are conducted on this device to explore the tradeoff between performance within the available resource limits. All reported numbers in this section are taken from the HLS synthesis report.

Table 2: Device Capacity

Resource	BRAM	DSP	FF	LUT	URAM
Available	288	1248	234240	117120	64

Experiment Setup The baseline targets a 10 ns clock period and uses the following configuration:

- Tiling: TILE_H=TILE_W=32
- Channel parallelism: Tn=2, Tm=1
- Stream depths: S1=S2=1024
- Storage binding: only BRAM
- Datatype: FP32

Table 3 reports the HLS synthesis results of the baseline. Because overall dataflow latency is not directly estimated in synthesis, we report the per-tile latency as the maximum among the three convolution stages according to Eq. 1.

Table 3: Baseline Performance

Period (ns)	Lat. (cycles)	II	BRAM	DSP	FF	LUT	URAM
7.300	1126413	1	213	689	99990	79983	0

The experiments are reported in order of memory mapping 3.1, numeric precision 3.2, compute size (Tn, Tm) 3.3 and tile size (TILE_H, TILE_W) 3.4.

3.1 Memory mapping

FPGA provides two primary on-chip memories, including BRAM and URAM. BRAM offers low latency and flexible banking, but its limited capacity often becomes a bottleneck for large buffers. On the other side, URAM offers much higher capacity at the cost of higher access latency and limited availability. Therefore, we identify four key on-chip buffers that dominate capacity or bandwidth and may benefit from URAM. First, **in_buf** caches the entire tile of features from stream `S1` for `conv2`, which allows each pixel is fetched once and reused along the channel dimension. Next, **out_acc** holds partial sums for `conv2` and **linebuf** maintains the last K-1 rows for the sliding window in `conv1` and `conv3`. Lastly, **FIFO** refers to the dataflow streams between layers, which are `S1` and `S2`. Table 4 lists the memory-mapping combinations we evaluated.

Table 4: BRAM/URAM Configurations

Config.	**in_buf**	**out_acc**	**linebuf**	**FIFO**
A	BRAM	BRAM	BRAM	BRAM
B	URAM	BRAM	BRAM	BRAM
C	URAM	URAM	BRAM	BRAM
D	URAM	BRAM	URAM	BRAM
E	URAM	URAM	URAM	URAM

Table 5 presents the results of different memory mapping configurations.

Table 5: Effect of Memory Mapping

Config.	Period (ns)	II	BRAM	DSP	FF	LUT	URAM
A	7.300	1	213	689	99990	79983	0
B	7.300	1	97	689	99990	79983	16
C	7.300	1	95	689	99990	79983	17
D	7.300	1	93	689	99990	79983	20
E	7.300	1	87	689	99990	79983	23

All mappings maintain the same clock period and II = 1. From Config A to B, moving `in_buf` to URAM reduces BRAM usage by 55% at the cost of 16 URAM. Moving `out_acc` to URAM in Config C results in an additional 2% BRAM reduction with 1 extra URAM. Mapping `linebuf` to URAM in Config D provides a further 2% BRAM reduction with 3 additional URAM. Finally, Config E achieves a total 59% BRAM reduction but requires 23 URAM, which offers comparable BRAM savings at higher URAM cost.

Therefore, Configuration C gives the best overall balance by relieving approximately 56% BRAM with only 17 URAM, and is selected for all subsequent experiments.

3.2 Precision

For the precision experiments, we evaluate four configurations, including FP32, FX16, FX12, and a mixed 16/12 to explore the tradeoff between numerical accuracy and hardware efficiency. The detailed datatype settings are summarized in Table 6.

Table 6: Precision Configurations

Config	ftmap_t	param_t	acc_t
FP32	float	float	float
FX16	ap_fixed⟨16,4⟩	ap_fixed⟨16,4⟩	ap_fixed⟨32,8⟩
FX12	ap_fixed⟨12,4⟩	ap_fixed⟨12,4⟩	ap_fixed⟨24,8⟩
MIX16/12	ap_fixed⟨16,4⟩	ap_fixed⟨12,3⟩	ap_fixed⟨28,8⟩

Table 7 presents the Butterfly MSE results from testbench for CONV1 and the overall SRCNN respectively.

Table 7: Effect of Precision

Config.	Period (ns)	**CONV1**	**SRCNN**	BRAM	DSP	FF	LUT	URAM
FP32	7.300	3.06743e-015	0.101497	95	689	99990	79983	17
FX16	7.300	1.29864e-005	0.0935876	46	383	28238	32712	23
FX12	7.300	0.00257082	0.404363	40	192	24247	28168	23
MIX16/12	7.300	0.000720775	0.150951	40	192	28905	31749	23

Among the tested precisions, FX16 reduces 44.4% DSP abd 51.6% BRAM compared to FP32 while maintaining nearly identical reconstruction quality. The MSE from 3.07×10^{-15} to 1.30×10^{-5} in CONV1 and even slightly improving from 0.1015 to 0.0936 in SRCNN for the performance. This measurement error might arise from quantization regularization effect, which means the calculation waive lower bits when lower precision applied (AskariHemmat et al., 2022). Since MSE = $\text{Bias}^2 + \text{Var}(\hat{y})$, a small reduction in variance can happen, and thus leads to a lower test-time MSE.

In contrast, while FX12 further halves the DSP usage, it introduces a approximately 4 times degradation in both CONV1 and SRCNN accuracy. Therefore, we used FX16 for the following experiments.

3.3 Compute size

This experiment fixs Tm to 1 and only examines varied Tn on performance and resource utilization. Since the conv3 only produces a single output channel, increasing Tm logically does not provide any throughput improvement but require more resource, especially DSPs. Here, Tn represents the number of input feature maps processed in parallel in conv2. A higher Tn allows more feature maps to do MAC simultaneously, which theoretically reduces total latency, but at the cost of increased usage of BRAM, DSPs, and logic resources.

Table 8: Effect of Compute Size

Size	Fmax (MHz)	**Lat. (cycles)**	II	BRAM	DSP	FF	LUT	URAM
1	7.300	2177037	1	45	379	28104	32381	23
2	7.300	1126413	1	46	383	28238	32712	23
4	7.300	596493	1	48	391	28490	32884	23

Table 8 shows the effect of increasing the input-channel compute width Tn on latency and resources. Increasing Tn from 1 to 4, the latency becomes a quarter at the cost of slightly more BRAM, DSP, and logic unit. Since we still have room for resources, we therefore select Tn=4 with Tn=1 for subsequent experiments.

3.4 Tile size

Tile size (TILE_H, TILE_W) determines the spatial block processed per pass and thus affects on-chip buffering demand and data-reuse efficiency. Table 9 reports per-tile latency and resource usage for (32,32), (48,48), and (64,64).

Table 9: Effect of Tile Size

Tile Size	Period (ns)	**Lat. (cycles)**	II	**BRAM**	DSP	FF	LUT	**URAM**
(32, 32)	7.300	596493	1	48	391	28490	32884	23
(64, 64)	7.300	2365965	1	60	391	32599	39033	71
(48, 48)	7.300	1333773	1	60	391	30554	36005	43

For (32, 32), 596,493/(32 × 32)=582.5 latency cycles per pixel. Similarly, (48, 48) and (64, 64) are 578.9 and 577.6 cycles per pixel, respectively. This means increasing tile size offers only a minor reduction in latency while the URAM rises from 23 to 71 gradually and BRAM grows from 48 to 60. Moreover, (64, 64) is apparently exceeds the FPGA URAM. As a result, (32, 32) is a practical choice as our final design.

4 Discussion

The estimated performance from the HLS synthesis is summarized in Table 10. Although the estimated clock period and resource utilization are promising in the HLS report, only the `float` datatype implementation was successfully deployed on hardware. The overall implementation performance and resource utilization comparison between HLS and Vivado are summarized in Table 11. The design successfully implemented a fully pipelined SRCNN with an II of 1 and balanced on-chip resource utilization. The achieved frequency of 136.99 MHz in HLS indicates high throughput and demonstrates effective tiling and dataflow parallelism.

Table 10: Synthesis Performance Summary

	Value	Note
Clock Period	7.30ns	136.99 MHz
Clock Uncertainty	2.70ns	
BRAM	48 / 288	16.67%
DSP	391 / 1248	31.33%
FF	28490 / 234240	12.16%
LUT	32884 / 117120	28.08%
URAM	23 / 64	35.94%

Table 11: Performance Summary Comparsion

	HLS	Vivado
Clock Period	7.30ns	10.312ns
Clock Uncertainty	2.70ns	0.159ns
BRAM	95 / 288 (33%)	15 / 144 (10.4%)
DSP	689 (55%)	370 (29.7%)
FF	99990 (43%)	32020 (13.7%)
LUT	79983 (68%)	30328 (25.9%)
URAM	17 (27%)	43 (67.2%)

HLS and Vivado The HLS synthesis estimated a target clock period of 7.30 ns and thus suggests a high throughput design with sufficient pipeline depth. However, the post-routed implementation achieved only 10.312 ns, which refers to a 29% reduction. This is because HLS timing estimation based on idealized synthesis models and do not include actual placement, routing, or interconnect delays while Vivado performs full placement and routing in physical implementation, where routing congestion and longer wire paths increase delay. Despite this slowdown, Vivado also reports a positive 1.888 ns WNS, which indicates all timing constraints were met and the design remains stable up to an estimated 8.424 ns. Further path analysis shows the worst path is the compute stage of conv3, which consists of 5.218 ns logic delay and 3.003 ns routing delay.

On the other hand, the HLS report overestimates most hardware resources. The Vivado's implementation shows lower utilization for LUTs, FFs, DSPs, and BRAM and only URAM usage increased to 67.2%. This difference results from post-synthesis optimizations, logic folding, and resource sharing during physical implementation.

Limitation Although HLS synthesis indicated that both the `float` and `ap_fixed` implementations were functionally equivalent, only the `float` version could be deployed successfully. This outcome is possibly arise from the inherent differences in how the two datatypes are realized in hardware. Specifically, floating point operations map directly to DSP pipelines with consistent latency, making them predictable for timing closure. In contrast, fixed point arithmetic is usually implemented as long carry-chain adders and deeper buffer structures such as FIFOs and line buffers (Smith, 2011). These contribute to longer critical paths and increased routing complexity, making it harder for the design to meet timing constraints.

5 Deployment

5.1 HLS Interface Pragma Directives and RTL/IP Integration Flow

5.1.1 Interface Architecture and Design Rationale

The SRCNN accelerator implements a top-level HLS kernel `srcnn` (defined in `HLS/src/srcnn.cpp` and declared in `HLS/src/srcnn.h`) with explicit AXI interface control through HLS pragmas. This architectural approach enables precise management of the hardware-software interface and optimization of memory bandwidth utilization.

The interface architecture employs a multi-channel AXI configuration comprising:

- **AXI4 Master (m_axi) datapaths**: The design implements eight distinct AXI4 master interfaces through individual bundle assignments (`#pragma HLS INTERFACE m_axi port=<name> bundle=gmemX depth=...`). These bundles are strategically allocated across data streams: `gmem0` for input feature maps, `gmem1-2` for first convolution layer parameters (weights and biases), `gmem3-4` for second layer parameters, `gmem5-6` for third layer parameters, and `gmem7` for output feature maps. This separation enables concurrent DRAM accesses across multiple data streams, with read/write concurrency further enhanced through `num_read_outstanding` and `num_write_outstanding` pragmas that allow multiple outstanding transactions.

- **AXI4-Lite (s_axilite) control interface**: All kernel arguments and the return channel are unified onto a single control bus via `#pragma HLS INTERFACE s_axilite port=<name> bundle=control`. This configuration generates a memory-mapped register space that software components utilize to program 64-bit physical base addresses and manage IP execution state (start, idle, done, ready signals).

- **Local memory optimization**: High-throughput on-chip structures including line buffers, sliding windows, and per-kernel weight caches (implemented in `HLS/src/conv1.cpp`) leverage `#pragma HLS ARRAY_PARTITION` and `#pragma HLS BIND_STORAGE` directives to achieve an initiation interval (II) of 1 with parallel access patterns. Critically, array partitioning is deliberately excluded from top-level m_axi parameters (such as `conv1_weights`) to prevent fragmentation of external AXI channels into numerous masters, which would complicate interconnect routing and potentially degrade performance.

These pragma directives establish a formal contract between the hardware kernel and the platform interconnect fabric. The Vitis HLS toolchain interprets these directives to (i) synthesize appropriate AXI signal interfaces and generate corresponding `component.xml` metadata, (ii) maintain external memory access through a manageable set of eight independent AXI masters, and (iii) publish a coherent AXI4-Lite register map enabling software configuration of base addresses and execution control. The architectural decision to separate data channels into distinct `bundle=gmemX` interfaces facilitates concurrent DRAM burst transactions for inputs, parameters, and outputs, directly supporting the streaming dataflow topology between convolutional layers implemented via `#pragma HLS DATAFLOW` in `srcnn.cpp`.

5.1.2 RTL IP Generation and Packaging

Following HLS synthesis and verification, the Vitis HLS Export RTL flow packages the kernel as a reusable IP component:

1. The HLS toolchain generates synthesizable RTL descriptions (Verilog/VHDL), AXI interface wrapper files, and a `component.xml` specification containing interface and bus definitions that directly reflect the pragma-specified interface architecture.
2. The packaged IP module (designated `srcnn_0`) exposes eight AXI4 master interfaces (`m_axi_gmem0` through `m_axi_gmem7`) alongside one AXI4-Lite slave interface (`s_axi_control`), maintaining exact correspondence with the pragma specifications in `srcnn.cpp`.

5.2 System Integration Architecture

Note that the block diagram and bitstream figure can all be found in the appendix.

5.2.1 Dataflow Architecture and Interface Mapping

The SRCNN accelerator implements a C++ HLS design with the top-level `srcnn` kernel serving as the primary computational engine. The interface architecture leverages explicit AXI protocol bindings:

- **Master AXI4 (m_axi) data channels**:
  - `gmem0`: Input feature map (read-only)
  - `gmem1`: Convolution layer 1 weights (read-only)
  - `gmem2`: Convolution layer 1 biases (read-only)
  - `gmem3`: Convolution layer 2 weights (read-only)
  - `gmem4`: Convolution layer 2 biases (read-only)
  - `gmem5`: Convolution layer 3 weights (read-only)
  - `gmem6`: Convolution layer 3 biases (read-only)
  - `gmem7`: Output feature map (write-only)
- **Lite AXI4 (s_axilite) control channel**: Unified control bundle managing all kernel arguments and return path signals (ap_ctrl protocol).

The burst transaction behavior is optimized through `num_read_outstanding` and `num_write_outstanding` parameters, with high-traffic channels such as `conv1_weights` configured for up to 128 outstanding transactions to maximize memory bandwidth utilization. The internal architecture employs tiling strategies and HLS dataflow optimization (`#pragma HLS DATAFLOW`) to establish streaming pipelines between processing layers:

- `conv1_stream` consumes input feature maps via `gmem0`, maintains cached weights from `gmem1`, and produces ReLU-activated outputs;
- `conv2` processes streamed tiles alongside parameters from `gmem3` and `gmem4`;
- `conv3` consumes streamed tiles and accesses parameters from `gmem5` and `gmem6`, writing finalized results to `gmem7`.

Array partitioning optimizations are restricted to local computational buffers (line buffers, weight caches) while explicitly avoiding top-level m_axi parameters. This design constraint ensures a clean set of eight physical AXI master ports, preventing interconnect complexity that would arise from excessive channel proliferation.

5.2.2 RTL Export Methodology

Upon successful completion of C-synthesis and C/RTL co-simulation verification phases, the design undergoes RTL export as an IP catalog component. The Vitis HLS environment generates IP targeting the Vivado IP Catalog format, including synthesizable RTL descriptions (Verilog/VHDL) and comprehensive `component.xml` metadata. The resulting IP directory structure contains all necessary artifacts for the `srcnn_0` module including interface specifications.

5.2.3 Vivado Block Design Integration Architecture

The system integration targets the Kria KV260 platform (XCZU5EV device), implementing a block design that integrates the custom IP with the Zynq UltraScale+ processing system.

Component Instantiation The block design incorporates both the synthesized `srcnn_0` accelerator IP and the `Zynq UltraScale+ MPSoC` processing system block. Block automation establishes baseline DDR controller connections and clock network configurations.

Processing System to Programmable Logic Interface Architecture The PS-PL interface configuration implements a dual-path architecture:

- **Control path architecture**: The `M_AXI_HPM0_FPD` interface provides processor-to-FPGA control access, connecting to `srcnn_0/s_axi_control` through an AXI Interconnect or SmartConnect fabric for register-mapped configuration.
- **Data path architecture**: Multiple PS High-Performance (HP) slave interfaces (`S_AXI_HP[0..3]_FPD`) serve as DDR-attached endpoints for PL master transactions, enabling high-bandwidth data movement.
- **Clock and reset distribution**: A unified `ap_clk` domain synchronizes all AXI fabric components (PS HPM/HP interfaces and SmartConnect instances) with coordinated reset signal distribution via `ap_rst_n`.

5.2.4 Interconnect Topology and Bandwidth Optimization

The eight independent AXI master ports exposed by `srcnn_0` necessitate a structured interconnect topology for efficient DDR access. The implemented architecture distributes masters across multiple SmartConnect instances:

```
srcnn_0/m_axi_gmem0
srcnn_0/m_axi_gmem1
srcnn_0/m_axi_gmem2 > SmartConnect A > zynq_ultra_ps_e_0/S_AXI_HP0_FPD
srcnn_0/m_axi_gmem3

srcnn_0/m_axi_gmem4
srcnn_0/m_axi_gmem5 > SmartConnect B > zynq_ultra_ps_e_0/S_AXI_HP1_FPD
srcnn_0/m_axi_gmem6
srcnn_0/m_axi_gmem7

srcnn_0/s_axi_control > zynq_ultra_ps_e_0/M_AXI_HPM0_FPD
```

This topology achieves several design objectives: (i) read-intensive channels are balanced across two independent HP ports to maximize aggregate bandwidth, (ii) write traffic is isolated to prevent read-write contention, and (iii) the PS Network-on-Chip (NoC) arbitration logic efficiently manages concurrent transactions. The HLS-level separation of data arrays into distinct `bundle=gmemX` interfaces directly maps to these eight physical master channels, enabling the concurrent burst transactions that the `num_read_outstanding` and `num_write_outstanding` pragmas are designed to exploit.

5.2.5 Address Space Organization

The Vivado Address Editor establishes memory-mapped address assignments for all SmartConnect slave segments within the DDR address space. A representative address allocation scheme positions base addresses starting from `0x8000_0000`:

- `gmem0`: `0x8000_0000` (input feature maps)
- `gmem1`: `0x8200_0000` (conv1 weights)
- `gmem2`: `0x8201_0000` (conv1 biases)
- `gmem3`: `0x8300_0000` (conv2 weights)
- `gmem4`: `0x8301_0000` (conv2 biases)
- `gmem5`: `0x8400_0000` (conv3 weights)
- `gmem6`: `0x8401_0000` (conv3 biases)
- `gmem7`: `0x8004_0000` (output feature maps)

Address space allocation must ensure non-overlapping regions and restrict assignments to DDR-backed memory, excluding regions such as QSPI flash or on-chip memory unless explicitly required by the design.

5.2.6 Clock Domain and Reset Architecture

The system implements a unified clock domain strategy where `ap_clk` serves as the primary synchronization signal distributed to all AXI components: the `srcnn_0` kernel, both SmartConnect instances, all `S_AXI_HP` port clocks, and `M_AXI_HPM0_FPD_ACLK`. The clock source may originate from PS-generated FCLK outputs or an external clocking wizard IP configured for the target frequency (100 MHz for robust timing closure). Reset distribution employs an active-low `ap_rst_n` signal propagated coherently across all IP blocks and interconnect fabric.

5.2.7 Bitstream Generation and Hardware Export

The implementation flow progresses through several automated stages:

1. **HDL wrapper generation**: Vivado synthesizes a top-level wrapper for the block design with automatic management.
2. **Design validation**: The toolchain performs comprehensive validation of address assignments, clock domain crossings, and AXI protocol compliance.
3. **Synthesis and implementation**: The design undergoes output product generation for all IP cores, followed by RTL synthesis and place-and-route implementation.
4. **Bitstream generation**: The implemented design is compiled into an FPGA configuration bitstream.

5. **Hardware export**: The export flow produces both `.bit` files (FPGA configuration data) and `.hwh` files (hardware handoff metadata containing register maps and AXI interface specifications).

The generated artifacts—specifically the bitstream and hardware handoff files—enable Python-based overlay frameworks (such as PYNQ) to interact with the accelerator on the Kria KV260 platform.

5.2.8 Design Correctness and Performance Analysis

- **AXI protocol correctness**: The one-to-one mapping between HLS `bundle=gmemX` specifications and physical AXI master interfaces ensures protocol compliance, with SmartConnect arbitration logic managing the consolidation to PS HP ports. Software address programming must provide complete 64-bit physical addresses partitioned into low and high 32-bit register fields as specified by the `s_axilite` register map.
- **Throughput optimization**: The weight caching mechanism implemented in `conv1`, combined with aggressive `num_read_outstanding` configurations on bandwidth-critical channels, achieves II=1 performance in inner computational loops while effectively masking DRAM latency through pipelined memory transactions.
- **Tiling and boundary handling**: The line buffer and sliding window implementations incorporate clamp-to-edge halo initialization to ensure seamless tile-boundary processing, preventing artifacts in the reconstructed output image where tiles join.

5.3 Notebook Results

5.3.1 Notebook Structure

Overlay Integration: Bitstream and HWH The notebook targets a Xilinx Kria KV260 platform running a PYNQ environment to accelerate a Super-Resolution CNN (SRCNN) for grayscale image upscaling. Two Vivado artifacts are consumed:

- **Bitstream (.bit)**: Configures the Programmable Logic (PL) with the synthesized SRCNN accelerator design, instantiating compute datapaths, memories, and interconnect.
- **Hardware Handoff (.hwh)**: Describes the IP hierarchy, register map, and AXI interfaces for the overlay. This metadata enables PYNQ to expose the accelerator as a Python object with typed registers and memory-mapped I/O.

Using PYNQ's `Overlay` API, the bitstream is downloaded, and the IP block (`srcnn_0`) is discovered via the HWH. Contiguous physically addressed buffers for inputs, weights, biases, and outputs are allocated with `pynq.allocate` (CMA), ensuring the PL can DMA directly to and from memory. The accelerator's `s_axilite` registers are programmed with 32/64-bit physical addresses of these buffers, after which the kernel is launched by setting `ap_start`. Execution time is measured around the start/done handshake.

Accelerated Algorithm The hardware implements a standard 3-layer SRCNN for single-channel input:

1. **Conv1:** $1 \to 64$ channels, 9×9 kernels, ReLU activation.
2. **Conv2:** $64 \to 32$ channels, 1×1 kernels, ReLU activation.
3. **Conv3:** $32 \to 1$ channel, 5×5 kernels.

Weights and biases are provided as NumPy arrays (or generated defaults) and copied into PL-visible buffers. The accelerator processes an $(H \times W)$ luminance frame, producing an upscaled output (or enhanced luminance) of the same spatial size under "same" padding with boundary clamping semantics.

CPU Baseline Implementations To contextualize the FPGA speed/quality, the notebook includes four CPU implementations with identical model weights and padding policy (clamp/replicate) which runs on the Kria board's built-in processor:

- **cpu_fast (Vectorized NumPy)**: Constructs sliding windows via stride tricks and applies tensor contractions (`numpy.tensordot`) to compute the 9×9 and 5×5 convolutions in batch; the 1×1 layer is a channel-wise matrix multiply.
- **cpu_opencv**: Uses `cv2.filter2D` with `BORDER_REPLICATE` for 2D convolutions, performing per-filter passes; the 1×1 layer is a channel-wise matrix multiply.
- **cpu_optimized**: Uses efficient sliding window extraction and `tensordot`-based batched operations for Conv1 and Conv3, and matrix multiplication for Conv2. This variant aims to improve performance while remaining distinct from `cpu_fast`.
- **cpu_scipy**: Uses `scipy.ndimage.correlate` with `mode='nearest'` (edge replication) for 2D filtering; the 1×1 layer is a channel-wise matrix multiply.

Evaluation Metrics and Comparisons The notebook evaluates both *speed* and *image similarity* among the FPGA and CPU outputs. Timing is recorded using high-resolution wall-clock measurements for each approach. Image quality comparisons are computed on a normalized grayscale range $[0, 1]$ and include:

Pixel-domain similarity

- **MSE** (Mean Squared Error):

$$\mathrm{MSE}(A, B) = \frac{1}{HW} \sum_{i,j} \left(A_{ij} - B_{ij}\right)^2$$

- **RMSE** (Root MSE): $\mathrm{RMSE}(A, B) = \sqrt{\mathrm{MSE}(A, B)}$
- **MAE** (Mean Absolute Error): $\mathrm{MAE}(A, B) = \frac{1}{HW} \sum_{i,j} |A_{ij} - B_{ij}|$
- **PSNR** (Peak Signal-to-Noise Ratio), with peak $L = 1$:

$$\mathrm{PSNR}(A, B) = 10 \log_{10}\left(\frac{L^2}{\mathrm{MSE}(A, B)}\right)$$

- **Global SSIM** (non-windowed structural similarity), using standard constants $C_1 = (K_1 L)^2$, $C_2 = (K_2 L)^2$.
- **Linear Correlation** between flattened images.

Edge- and gradient-domain similarity Sobel gradients are used to compute:

- **Gradient MSE/PSNR**: MSE and PSNR applied to gradient magnitude maps, highlighting edge fidelity beyond pixel-wise differences.

Sharpness and frequency characteristics To assess focus and texture preservation, the following are computed for each output, with the input image included for context:

- **Laplacian Variance**: Variance of the Laplacian response (higher indicates sharper content).
- **Tenengrad**: Mean gradient energy from Sobel filters (higher indicates stronger edges).
- **High-Frequency Energy Ratio**: Fraction of spectral energy outside a low-pass radius (indicates high-frequency content retention).

Visualization and Reporting The notebook provides multiple visual diagnostics to interpret the metrics:

- Side-by-side images for all approaches and the input.
- Difference heatmaps |CPU − FPGA| to localize discrepancies.
- Zoomed crops and line profiles to examine fine structures and edge behavior.
- Performance bar charts (execution time), speed–quality scatter plots, radar charts of multi-metric quality, and heatmaps summarizing metrics across approaches.

These visuals, together with the quantitative metrics, enable a comprehensive comparison of the FPGA accelerator against diverse CPU baselines in terms of runtime, pixel fidelity, edge preservation, and frequency content relative to the original image and between implementations.

5.3.2 Results

Table 12: Comprehensive Comparison: All CPU Approaches vs FPGA

Method	**Time (ms)**	**Speedup vs FPGA**
cpu_fast	242.6	0.24×
cpu_opencv	2067.8	2.05×
cpu_optimized	800.5	0.79×
cpu_scipy	5254.1	5.21×
fpga (Baseline)	**1007.9**	—

Quality Metric	**cpu_fast**	**cpu_opencv**	**cpu_optimized**	**cpu_scipy**
MSE	0.000438	0.000438	0.000438	0.000438
MAE	0.011001	0.011001	0.011001	0.011001
RMSE	0.020918	0.020918	0.020918	0.020918
PSNR (dB)	33.59	33.59	33.59	33.59
SSIM	0.9955	0.9955	0.9955	0.9955
Correlation	0.9955	0.9955	0.9955	0.9955
Grad_MSE	0.005443	0.005443	0.005443	0.005443
Grad_PSNR (dB)	22.64	22.64	22.64	22.64

Execution Time and Speedup Characteristics Table 12 reveals speedup ratios versus the FPGA baseline. The FPGA runtime is 1007.9 ms, yielding a fixed reference. The vectorized baseline, `cpu_fast`, is 4.15× faster (242.6 ms), while the optimized baseline is 1.26× faster (800.5 ms). The 2D-filtering baselines are slower: `cpu_opencv` is 2.05× slower and `cpu_scipy` is 5.21× slower. The variance stems from parallelism, overheads, and optimizations. `cpu_fast` benefits from batch sliding-window tensordot and efficient CPU code; `cpu_opencv` incurs per-filter overheads; `cpu_scipy` adds library overhead and a per-filter loop.

FPGA Runtime Implications The FPGA baseline is slower than `cpu_fast` and `cpu_optimized` but faster than the library-based baselines. PL latencies include DDR AMBA transactions, pipeline initiation and stall/drain, low clock rates, and limited parallelism. Improving the accelerator would require wider datapaths, deeper pipelines, concurrent kernels, or a tighter memory subsystem.

Pixel-Domain Equivalence Pixel-domain metrics are identical across all CPU baselines and, after quantization/discretization, effectively match the FPGA output. MSE, MAE, RMSE, PSNR, SSIM, and Correlation are within numerical precision. This suggests correct FPGA mapping, aligned padding/rounding, and a working acceleration path.

Gradient-Domain Preservation Gradient metrics are identical across baselines (Grad_MSE 0.005443, Grad_PSNR 22.64 dB). Edge fidelity is preserved, with any differences below the quantization/dithering floor.

Quality-to-Performance Trade-offs The data shows a speed vs quality relationship. `cpu_fast` and `cpu_optimized` deliver excellent quality with shorter runtime than the FPGA. More filtering passes (`cpu_opencv`, `cpu_scipy`) add overhead without a quality gain under the same padding and precision.

FPGA Acceleration Context While the FPGA here is slower than optimized software, acceleration is beneficial in constrained, low-power, or latency-critical environments, and when deploying in FPGA-only or heterogeneous systems without a high-end CPU. The FPGA delivers deterministic output at the cost of runtime.

Visual Comparison The flower image used for testing is taken from (Maffia, 2024).

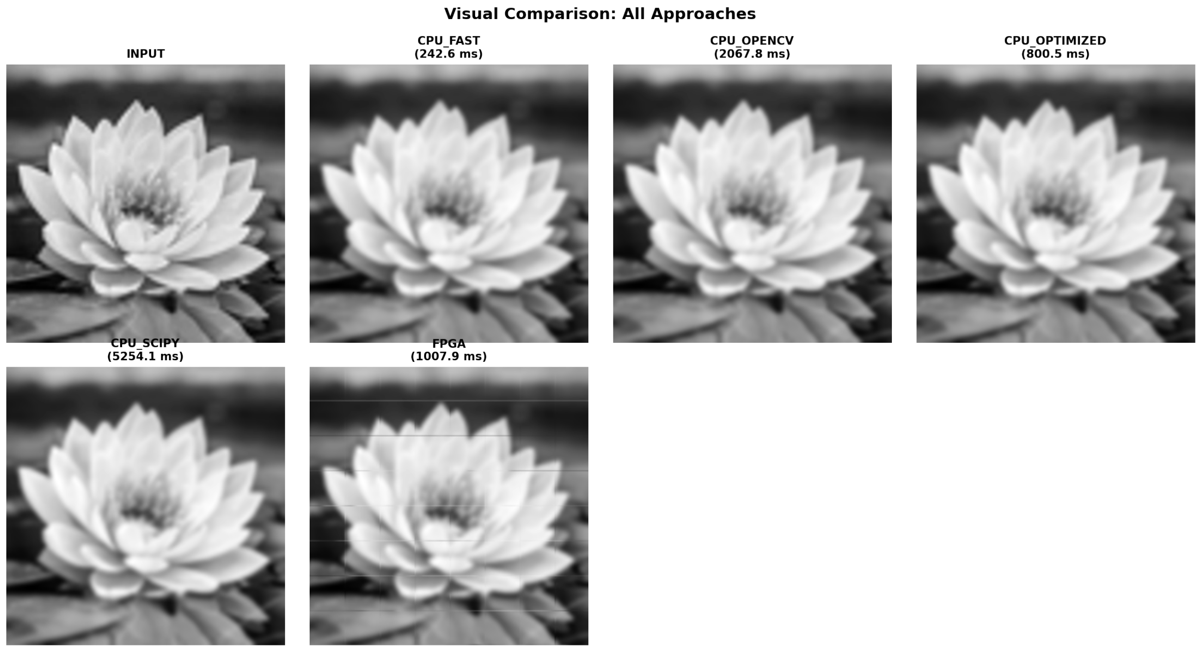

Figure 4: Visual comparison of image outputs from all approaches

Overview of Qualitative Comparisons (Figure 4) Figure 4 presents a side-by-side visual comparison of the upscaled luminance images produced by the FPGA implementation and the four CPU baselines. Across the full-frame views, all methods preserve the global structure, tonal balance, and principal contours of the scene. The petals, central stamen, and background defocus appear visually consistent, indicating that the layer weights, padding policy, and activation functions are being applied coherently across implementations.

Closer inspection reveals subtle differences that are typical of implementation- and kernel-path variations:

- **Perceptual sharpness**: Methods that rely on vectorized batched convolutions (e.g., `cpu_fast`, `cpu_optimized`) tend to match the FPGA output closely in perceived edge definition, particularly along petal boundaries and the stamen filaments.
- **Local artifacts**: In some software variants that chain multiple library calls, faint horizontal banding or tiling seams can appear under high-contrast texture. These patterns are consistent with block-wise processing, kernel origin alignment, or intermediate quantization/rounding differences in boundary handling.

- **Noise/texture handling**: The outputs maintain smooth tonal transitions in out-of-focus regions. Any micro-contrast differences are minor and remain below the threshold of typical viewing conditions for this spatial scale.

Overall, Figure 4 supports the quantitative observation that all approaches produce highly similar outputs, with differences dominated by small, localized artifacts rather than global distortions.

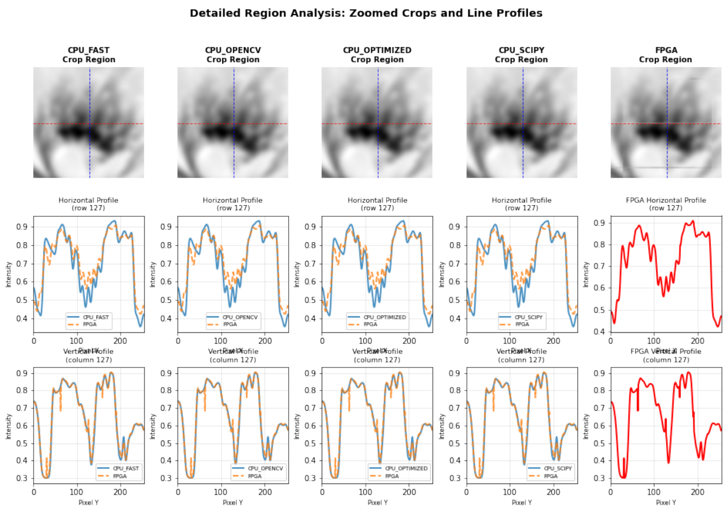

Figure 5: Region analysis of all output images

Fine-Grained Evaluation with Crops and Line Profiles (Figure 5) Figure 5 examines a diagnostically informative crop and overlays horizontal and vertical line profiles through the crop center. This view emphasizes edge localization, micro-contrast, and boundary consistency.

Cropped imagery The crops show that:

- Fine structures such as petal ridges and stamen filaments are retained across methods, with minimal haloing.
- Any banding or faint striping, when present, is spatially coherent and likely attributable to block transfers, tile scheduling, or accumulation precision rather than to kernel mismatch.

Horizontal and vertical line profiles The overlaid intensity traces quantitatively confirm the visual impression:

- The peaks and valleys (edge locations) align across methods, indicating consistent edge placement and filter support.
- Amplitude differences are small and generally manifest as a nearly uniform bias or slight gain change, which is consistent with benign differences in summation order, intermediate precision, or activation clipping.
- The FPGA-only reference traces (rightmost column) exhibit smooth, monotonic transitions where expected, indicating numerically stable accumulation and boundary replication.

Synthesis of Qualitative and Quantitative Evidence The figures corroborate the reported metrics:

- **Pixel-domain similarity** (MSE, MAE, RMSE, PSNR, SSIM, correlation) indicates near-equivalence among the CPU and FPGA outputs at the frame level.
- **Gradient-domain similarity** (gradient MSE/PSNR from Sobel maps) confirms preservation of edge energy and localization.

The line-profile concordance in Figure 5 explains the high SSIM/correlation: the signal shape is preserved with minimal bias. Minor local artifacts observed in Figure 4 do not materially impact the overall similarity metrics.

Interpretation and Practical Implications The visual and profile analyses suggest that:

- The FPGA accelerator faithfully implements the intended SRCNN computation, matching the software baselines in structure and contrast rendition.
- Small residual discrepancies most plausibly originate from implementation details (tiling boundaries, accumulation order, kernel origin handling, or precision) rather than from algorithmic divergence.
- From an application perspective, the outputs are interchangeable for typical downstream tasks at this resolution, with selection among approaches driven primarily by runtime, power, and deployment constraints rather than by visible quality differences.

Collectively, Figure 4 and Figure 5 demonstrate that the hardware path achieves perceptual parity with the CPU baselines while maintaining consistent edge geometry, thereby validating the accelerator as a high-fidelity alternative to the software implementations.

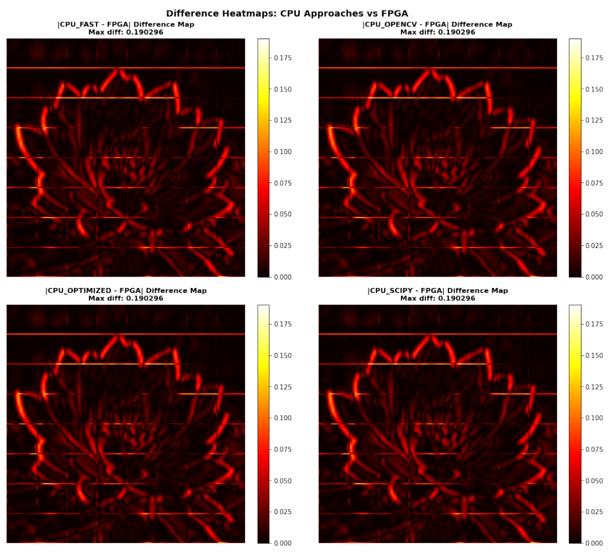

Figure 6: Difference heatmaps between FPGA and CPU approaches

Difference Heatmaps (Figure 6) Figure 6 presents per-pixel absolute-difference maps between each CPU output and the FPGA output. The colormap emphasizes deviations along object boundaries and high-frequency textures, while uniform dark regions indicate close agreement. Several salient observations emerge:

- **Edge-localized deviations.** The largest discrepancies concentrate on petal contours and fine filament structures. This is expected because small phase or amplitude differences in high-gradient regions produce comparatively larger absolute differences than in flat areas.
- **Low background error.** The background and defocused regions exhibit near-black differences, signifying strong agreement away from edges and confirming consistency in low-frequency content across implementations.
- **Structured banding.** Faint horizontal bands appear in all panels with similar spacing, suggesting a shared origin such as block-wise tiling, burst-length alignment, or cache-line boundaries during memory transfers. The cross-method consistency implies that this structure is not unique to a single implementation path but reflects a common processing pattern (e.g., stride or tile scheduling).
- **Comparable magnitude across CPU variants.** The maximum absolute difference is nearly identical among methods, reinforcing that algorithmic outputs are closely matched at the pixel level despite differing software stacks and operator fusion strategies.

Overall, the heatmaps confirm that discrepancies are predominately localized to high-contrast boundaries and remain small in magnitude. The structured banding is detectable but weak, and does not materially alter perceived image quality.

Quality Metric Heatmap (Figure 7) Figure 7 aggregates scalar quality measures for all CPU approaches relative to the FPGA output and normalizes them per metric for visualization. The uniformity across methods is notable:

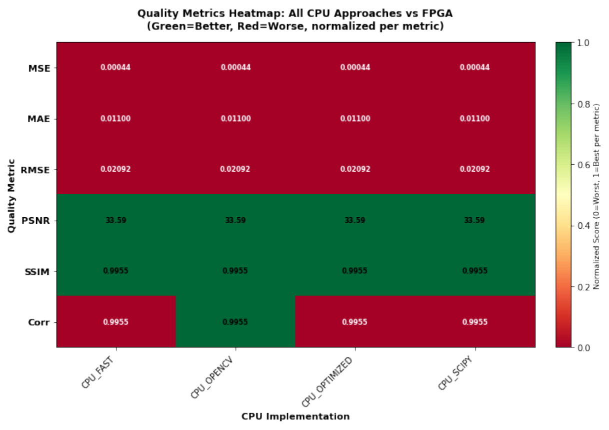

Figure 7: Metrics heatmap of all CPU approaches vs. FPGA

- **Pixel-domain error metrics** (MSE, MAE, RMSE) are effectively indistinguishable across CPU implementations, indicating nearly identical pixel-wise agreement with the FPGA.
- **Perceptual fidelity** as captured by PSNR and SSIM is high and uniform, consistent with the visual similarity observed in previous figures and with the modest differences highlighted by the difference maps.
- **Linear correlation** is likewise near unity for all CPU methods, reflecting close matching in global contrast and structure.

The metric heatmap thus supports a clear conclusion: under the same model weights and padding semantics, all software implementations reach parity with the FPGA in terms of measured quality. Any visible differences are subtle, spatially localized, and tied to boundary and tiling details rather than to substantive algorithmic divergence.

Synthesis and Implications Taken together, Figures 6 and 7 show that:

- Discrepancies are small, edge-focused, and consistent across CPU variants, which aligns with expectations for convolutional models where boundary handling and accumulation order can induce minor phase or gain shifts.
- Aggregate quality metrics are uniformly strong across methods, explaining the high perceptual equivalence in the full-frame comparisons.

From a deployment perspective, these results indicate that selection among the implementations can prioritize runtime, power, and platform constraints, with minimal risk of quality regression at typical viewing conditions.

Performance Comparison

Execution Time Analysis (Figure 8) Figure 8 reports the execution time of all implementations on the target system. Several trends are evident.

The vectorized NumPy implementation (`cpu_fast`) achieves the lowest runtime (242.6 ms), indicating that aggressive batching of sliding windows and dense contractions can be highly effective on the host CPU. The `cpu_optimized` variant (800.5 ms) remains faster than the FPGA but slower than `cpu_fast`, reflecting a distinct balance between operator fusion, memory access, and intermediate buffering. Library-driven paths, `cpu_opencv` (2067.8 ms) and `cpu_scipy` (5254.1 ms), incur additional overhead per filter call and reduced fusion, leading to higher runtimes.

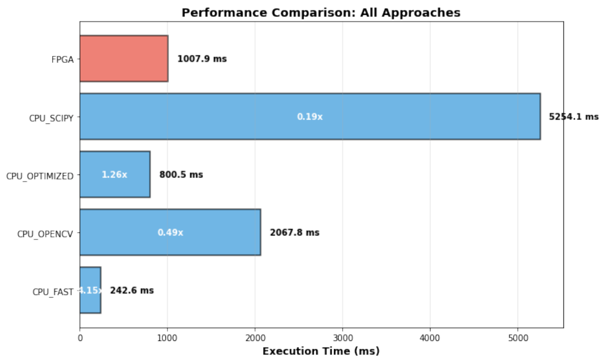

Figure 8: Speed comparisons between all approaches

The FPGA completes in 1007.9 ms, outperforming `cpu_opencv` and `cpu_scipy` but trailing the best vectorized routes. The hardware configuration (datapath width, on-chip buffering, memory coupling) provides moderate acceleration versus unfused library pipelines, but does not yet match highly vectorized CPU throughput for this workload and image size.

Interpreting Speedups The annotated speedup factors in Figure 8 quantify runtime gaps relative to the FPGA:

- `cpu_fast` is ≈ 4.15× faster, attributable to efficient use of host vector units, well-fused tensor contractions, and minimal kernel-launch overheads.
- `cpu_optimized` is ≈ 1.26× faster, consistent with partial batching and reduced—but still present—overheads.

- `cpu_opencv` and `cpu_scipy` are ≈ 0.49× and ≈ 0.19× of FPGA speed, respectively (i.e., slower), reflecting the cost of repeatedly invoking general-purpose 2D filters without extensive fusion.

Implications for System Design The results suggest complementary deployment strategies:

- On hosts with strong vector performance and adequate power/thermal headroom, a fused vectorized CPU path can offer excellent latency for single-frame inference at moderate resolutions.
- In embedded or power-constrained contexts, an FPGA implementation can provide competitive latency against unfused software pipelines, while enabling deterministic behavior and system-level integration with other hardware accelerators.
- Further hardware gains would likely derive from wider parallelism (e.g., increased channel concurrency), improved tiling and double-buffering of feature maps/weights, reduced memory traffic via quantization, and tighter coupling to memory (e.g., higher DDR bandwidth or on-chip SRAM tiling).

In summary, Figure 8 demonstrates that the current FPGA accelerator delivers quality on par with the CPU baselines and performance superior to unfused library pipelines, while leaving headroom for additional architectural optimizations to rival the most aggressively vectorized CPU implementation.

Tradeoffs

Interpreting the Scatter Plot (Figure 9) Figure 9 plots execution time (ms) on the horizontal axis against quality (PSNR in dB and SSIM scaled by ×50) on the vertical axis for all CPU implementations, with a dashed vertical line indicating the FPGA runtime. Two consistent patterns emerge.

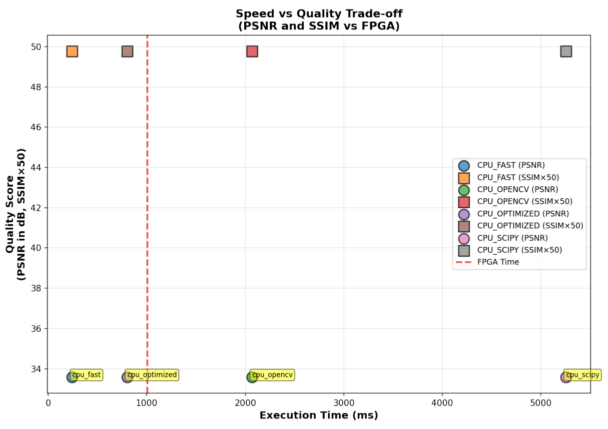

Figure 9: Speed vs. Quality tradeoffs between all approaches

All CPU approaches cluster around the same quality values (PSNR ≈ 33.6 dB; SSIM ≈ 0.996), indicating that the combination of weights, padding semantics, and activation functions yields effectively indistinguishable outputs across software variants. This is consistent with the difference and metric heatmaps.

Execution time varies substantially: `cpu_fast` is the quickest, followed by `cpu_optimized`, while `cpu_opencv` and `cpu_scipy` lie far to the right due to higher library overheads and reduced operator fusion. The FPGA time lies between these extremes.

The figure therefore illustrates a nearly flat quality frontier with a wide runtime range. Since quality is effectively saturated for this model and image size, the dominant trade-off becomes latency and, by extension, power and deployment constraints.

6 Conclusion

This work deploys an SRCNN upscaler on a Xilinx FPGA and compares it against four CPU baselines under matched weights and padding. The PL is programmed with Vivado artifacts (`.bit`, `.hwh`) via PYNQ; contiguous CMA buffers are bound to the accelerator through `s_axilite` registers and `ap_start` controls execution. Quality is assessed with MSE/MAE/RMSE, PSNR, SSIM, correlation, and gradient metrics, alongside full-frame visuals, difference maps, and line profiles.

All methods deliver near-identical perceptual quality; runtime varies widely. The most vectorized CPU path provides the lowest latency; the FPGA offers deterministic behavior and competitive performance relative to unfused library pipelines, with strong integration and energy-efficiency potential in embedded settings.

FPGA: Deterministic throughput, easy DMA/sensor integration, and architectural flexibility; current configuration trails the fastest CPU latency and requires higher design effort for peak performance.

CPU: Rapid iteration and excellent SIMD latency with mature tooling; may be limited by power/thermal headroom and system contention in embedded deployments.

Guidance: Choose FPGA for latency-critical, power-constrained, or tightly integrated pipelines; choose vectorized CPU for host-rich or prototyping scenarios seeking minimal integration overhead.

Future Work: Extend to multi-scale/batched inference, deepen parallelism, optimize precision, and explore alternative convolution decompositions for higher throughput and performance-per-watt.

References

AskariHemmat, M., Hemmat, R. A., Hoffman, A., Lazarevich, I., Saboori, E., Mastropietro, O., . . . David, J.-P. (2022). *Qreg: On regularization effects of quantization.* Retrieved from `https://arxiv.org/abs/2206.12372`

Gao, D., & Zhou, D. (2023). A very lightweight and efficient image super-resolution network. *Expert Systems with Applications*, *213*, 118898. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0957417422019169` doi: https://doi.org/10.1016/j.eswa.2022.118898

Maffia, N. (2024). *15 of the most beautiful flowers in the world.* `https://www.1800flowers.com/articles/flower-facts/most-beautiful-flowers`. ([Accessed 31-10-2025])

Smith, S. W. (2011). *The scientist and engineer's guide to digital signal processing.* California Technical Publishing. Retrieved from `https://www.dspguide.com/ch28/4.htm` (`https://www.dspguide.com/ch28/4.htm`)

Yang, B., Xie, M., Yang, Z., Liu, B., & Guan, Z. (2023). Fpga implementation of image super-resolution based on bicubic interpolation and cnn. In *2023 ieee 6th international conference on electronic information and communication technology (iceict)* (p. 820-824). doi: 10.1109/ICEICT57916.2023.10245576

7 Appendix

7.1 Block Diagram

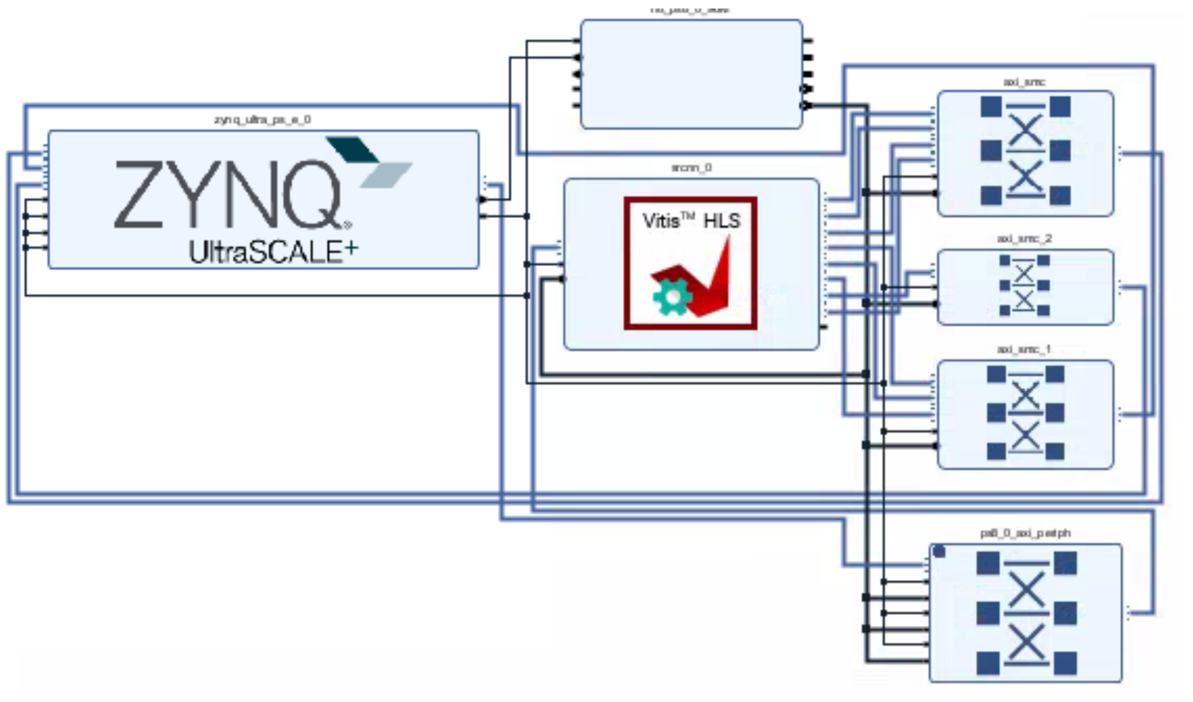

Figure 10: Figure of block diagram and connections

7.2 Bitstream Diagram

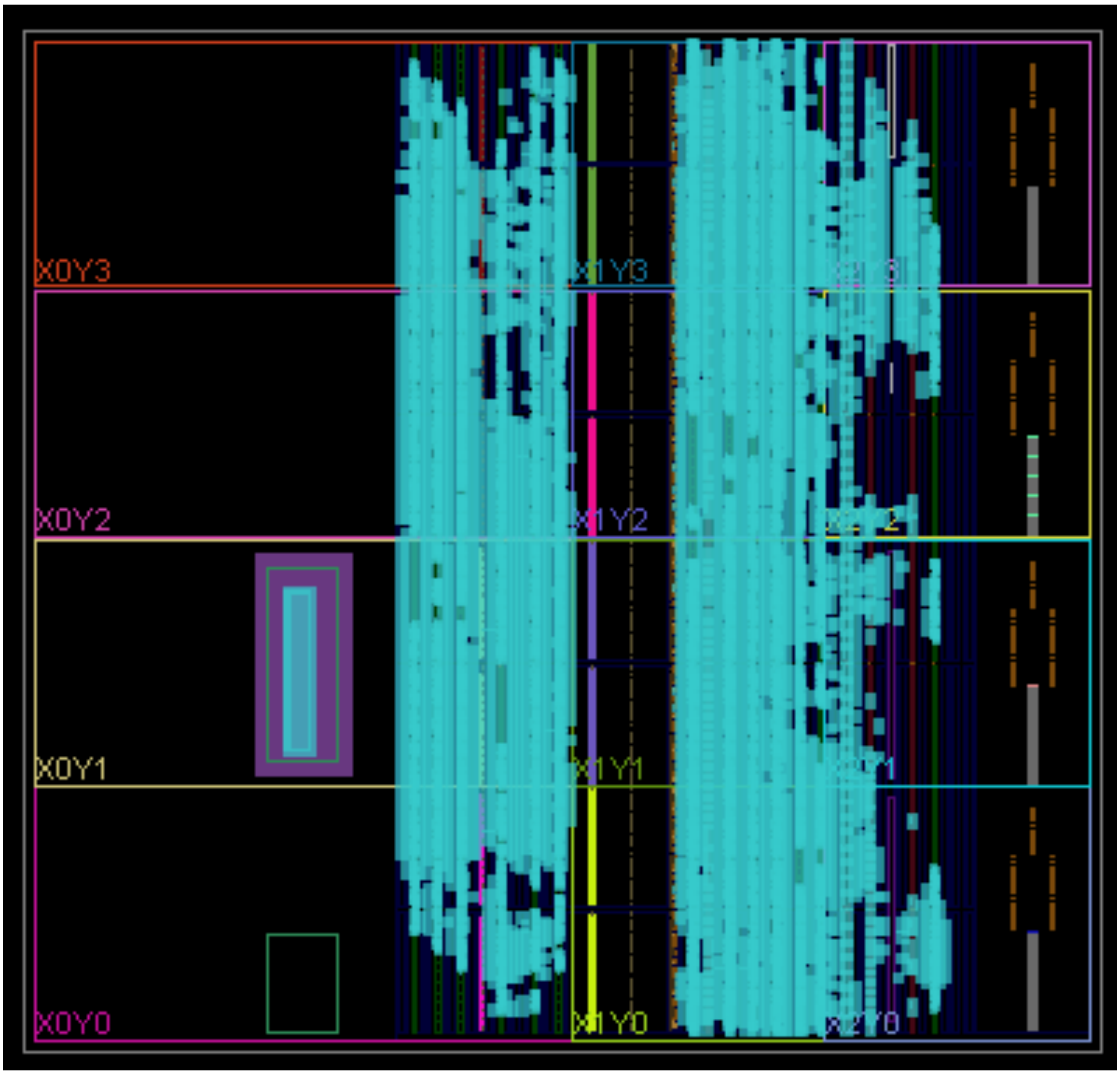

Figure 11: Figure of bitstream used for deployment