Bulletin of Electrical Engineering and Informatics
Vol. 15, No. 6, December 2026, pp. 5092∼5112
ISSN: 2302-9285, DOI: 10.11591/eei.v15i6.12700

Hardware-software co-design for image convolution on the Kria KV260

Kamal Sehairi[1,2], Ibtissam Mama Boucherit[2], Antonio Javier Gallego[3], Fatima Chouireb[2]
[1]Laboratory of Applied Sciences and Didactics, Ecole Normale Supérieure de Laghouat, Laghouat, Algeria
[2]Department of Electronics, Telecommunication, Signals, and Systems Laboratory, University Amar Telidji of Laghouat, Laghouat, Algeria
[3]Pattern Recognition and Artificial Intelligence Group, University of Alicante, Alicante, Spain

Article Info

Article history:

Received Mar 3, 2026
Revised Sep 20, 2026
Accepted Oct 4, 2026

Keywords:

All-programmable SoC
AXI4-Stream
Co-design
Image convolution
Image processing
Kria KV260

ABSTRACT

Image convolution is a computationally intensive operation in embedded computer vision systems. This study aims to develop and evaluate a streaming field-programmable gate array (FPGA)–CPU hardware–software co-design for accelerating two-dimensional image convolution on the AMD Xilinx Kria KV260 platform. The accelerator was implemented using Vitis high-level synthesis (HLS) and Vitis Vision with Advanced eXtensible Interface 4 Stream (AXI4-Stream) interfaces for both image data and runtime-programmable filter coefficients. Two processing configurations were evaluated: XF_NPPC1, processing one pixel per clock cycle, and XF_NPPC8, processing eight pixels per cycle. Performance was compared with the OpenCV filter2D implementation executed on the ARM Cortex-A53 processor using grayscale and RGB images at resolutions up to 4K. Experimental results show that, for a 4K RGB 7×7 convolution, XF_NPPC8 reduces end-to-end latency from 674 ms to 58.6 ms, providing an 11.50× speedup. For a 512×512 grayscale 3×3 workload, it achieves 990.1 frames per second (FPS) and 4.67 GOPS. System-level power consumption is 2.804 W for NPPC1 and 2.851 W for NPPC8, corresponding to energy efficiencies of approximately 0.52 and 1.64 GOPS/W, respectively. These results demonstrate that vector-level parallelism substantially improves throughput and energy efficiency while maintaining moderate system-level power consumption, supporting real-time high-resolution image-processing applications.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Kamal Sehairi
Laboratory of Applied Sciences and Didactics, Ecole Normale Supérieure de Laghouat
Laghouat, Algeria
Email: k.sehairi@lagh-univ.dz

1. INTRODUCTION

With the rapid advancement of deep learning techniques, particularly convolutional neural networks (CNNs) [1], [2] and, more recently, vision transformers [3], [4], image convolution has become a central operation in modern computer vision systems. Convolution layers dominate the computational workload of these models, both in terms of execution time and energy consumption [2], [5]. As a result, achieving fast convolution performance while maintaining low power usage has become a critical challenge, especially for embedded and edge computing applications where resources are constrained [6], [7].

To address this challenge, a wide range of specialized computing technologies have emerged, including neural processing units (NPUs), visual processing units (VPUs), tensor processing units (TPUs), graphics

processing units (GPUs), application-specific integrated circuits (ASICs), deep processing units (DPUs), and field-programmable gate arrays (FPGAs) [2], [6]. Each of these platforms offers different trade-offs in performance, flexibility, and energy efficiency [2], [7]. Among them, FPGAs have attracted significant attention due to their ability to exploit fine-grained parallelism across data, control, and processing flows, while maintaining low and predictable power consumption [5], [8]. Unlike fixed-function accelerators, FPGAs also provide a high degree of reconfigurability, making them well suited for evolving deep learning workloads [7], [8].

In this work, we leverage these advantages by proposing an efficient FPGA-based implementation of image convolution using AMD Xilinx Vitis high-level synthesis (HLS) and the Vitis Vision library. The proposed design implements image convolution as a dedicated hardware intellectual property (IP) core capable of parallel, pipelined processing with deterministic throughput. To maximize data movement efficiency and minimize control overhead, both image data and convolution kernels are transferred using the Advanced eXtensible Interface 4 Stream (AXI4-Stream) protocol, rather than conventional AXI4-Lite-based parameter passing. This streaming approach enables runtime reconfigurability of convolution kernels while sustaining high processing throughput.

The accelerator is integrated within the PYNQ-based system architecture of the Kria KV260 platform, where the processing system runs a full operating system and manages image acquisition, kernel configuration, and data transfer through AXI4-Stream. This hardware–software co-design allows the compute-intensive convolution operation to be offloaded to programmable logic, while higher-level control and application management remain on the embedded processor. The entire design flow, including IP generation, system integration, and bitstream creation, is realized using the Vitis toolchain and Vivado Design Suite.

The main contribution of this work is a streaming FPGA–CPU convolution architecture in which both image data and runtime-programmable filter coefficients are delivered through independent AXI4-Stream paths. Unlike conventional implementations in which convolution coefficients are statically bound at synthesis time or supplied through memory-mapped interfaces, the proposed approach allows the filtering operator to be changed at runtime without hardware re-synthesis, thereby providing an adaptable hardware–software framework for high-throughput embedded image processing.

The remainder of this paper is organized as follows. Section 2 reviews related work on hardware acceleration of image convolution and existing hardware–software co-design approaches. Section 3 describes the proposed methodology, detailing the system architecture, design choices, and implementation steps. Section 4 presents experimental results and performance evaluations, including comparisons with CPU-based implementations. Finally, section 5 concludes the paper by summarizing the main findings and highlighting the potential of the proposed design for efficient convolution acceleration in embedded vision and deep learning systems.

2. RELATED WORKS

In the context of accelerating image processing and deep learning workloads on FPGA platforms, numerous architectural techniques and design approaches have been proposed in the literature. In 2023, Pham-Quoc and Thinh [9] introduced an FPGA-based architecture for efficient computation of convolution operators using a deeply pipelined processing scheme. To improve performance, the authors employed a matrix of multiply–accumulate (MAC) units combined with pipelined registers and multiple firs in first out (FIFO) buffers, enabling parallel computation and continuous data streaming. The proposed design was implemented and evaluated on the Ultra96-V2 development board, which integrates a Zynq UltraScale+ MPSoC featuring a quad-core ARM Cortex processor tightly coupled with FPGA programmable logic. Experimental results demonstrated that the proposed accelerator achieved a speedup of 14.77$\times$ compared to a software implementation. However, this performance gain was obtained at the cost of very high resource utilization, consuming approximately 82.04% of lookup tables (LUTs), 97.45% of block RAMs (BRAMs), 67.78% of digital signal processing blocks (DSPs), and 56.92% of flip-flops (FF).

In another work, Aydin and Bilge [10] presented an FPGA-based implementation of convolution, pooling, and fully connected layers for deep learning acceleration. These layers were designed using nested loop structures, and to optimize the convolution layer, the authors applied loop unrolling techniques to enable concurrent execution of convolution operations. Experimental results indicated that this approach could not be efficiently implemented on the ZedBoard FPGA due to clock frequency limitations and insufficient hardware resources. However, the proposed design was successfully deployed on the Kintex UltraScale+ FPGA KCU116 platform, where it achieved a performance improvement of up to 139× compared to the CPU-based implementation, while maintaining a power consumption below 0.55 W. Zhao *et al.* [11] designed custom IP

cores for accelerating the convolutional computations of the YOLOv4-Tiny network using high-level synthesis. In their approach, the overall YOLOv4-Tiny network structure was implemented on the processing system (PS), while the computationally intensive convolutional multiplication and accumulation operations were offloaded to the programmable logic (PL). Input feature maps, convolution weights, and output feature data were transferred between the PS and PL through the AXI4 interface. The proposed accelerator was implemented on the Zynq-7020 platform, where experimental results showed that the design utilized approximately 99% of available DSP resources, 71% of LUTs, and 69% of BRAMs, achieving a processing time of 383 ms with a power consumption of 2.75 W.

Gao *et al.* [12] investigated the reliability of convolution accelerators implemented on SRAM-based FPGAs under single-event upsets (SEUs). Unlike conventional streaming architectures that focus primarily on maximizing computational throughput, the authors proposed a single computation engine (SCE) architecture that adopts an instruction set architecture (ISA) approach, allowing the same hardware logic to be reused for computing different neural network layers. The proposed SCE-based accelerator consists of four main modules: an instruction scheduler, a computing engine, a data mover, and an on-chip buffer. The design was implemented on the PL of a Zynq-7000 SoC XC7Z030 platform, while the PS, based on an ARM processor, was responsible for providing instruction addresses, weights and biases, and input image data. The computing engine includes a convolution (CONV) module and a miscellaneous (MISC) module that performs pooling and element-wise operations. Communication between the PS and PL is achieved through the AXI interface. Experimental results indicate that the proposed accelerator consumes more than half of the available FPGA resources. Furthermore, the reliability analysis shows that increasing the degree of parallelism in the computation modules (CONV and MISC) improves fault tolerance, as the impact of a single faulty unit becomes less significant. Sestito *et al.* [13] presented the implementation of a parameterizable transposed convolution (TCONV) module on FPGA, generated via HLS. To mitigate the computational redundancy often associated with zero-insertion up-sampling methods, the authors adopted an input-oriented mapping (IOM) algorithmic strategy. The hardware performance was further optimized through the strategic application of HLS pragmas, specifically controlling dataflow, loop unrolling, and pipelining. Experimental results demonstrate that the proposed TCONV module achieves a throughput of 6.25 giga outputs per second (Gout/s) with a power consumption of just 1.53 W. In the context of image decompression, the design attains a throughput exceeding 30,000 frames per second (FPS) while utilizing, on average, only 16% of the available FPGA resources.

In 2024, Moussa *et al.* [14] developed a hardware–software co-design on the Zynq-7000 SoC (Zed-Board) to execute real-time image processing operations, including erosion, dilation, and neighborhood convolution. The system architecture integrates the PS with PL using the AXI4 interconnect, employing direct memory access (DMA) to efficiently stream 512×512 grayscale images from external DDR memory to the custom hardware IP. This IP utilizes internal line buffers and a window generator to process pixels, while an onboard OLED module provides real-time timing verification. Comparison via universal asynchronous receiver-transmitter (UART) confirmed a 99.22% accuracy against a software reference. Experimental results indicate the convolution architecture is highly efficient, achieving a processing time of 2.6470 ms and utilizing only 1154 look-up table random-access memory (LUTRAM), 1701 LUTs, 270 FF, 1 BUFG, and 0.5 BRAM, with zero DSP usage. Liu *et al.* [15] employed a set of loop optimization techniques, including loop tiling, loop interchange, and loop unrolling, to design an efficient FPGA-based convolution accelerator. The convolution operation is formulated as four nested loops, and the accelerator architecture is derived through a quantitative analysis of the trade-offs among computational parallelism, partial-sum storage requirements, on-chip buffer capacity, and off-chip memory traffic. The proposed architecture is implemented on two Xilinx platforms, namely the ZCU102 and ZCU111. Experimental results demonstrate that the implementations of VGG-16, ResNet-50, and ResNet-152 achieve peak throughputs of 721.48 GOPS, 546.98 GOPS, and 664.66 GOPS, respectively. In terms of resource utilization, the ZCU102 implementation consumes 91,245 FFs, 140,832 LUTs, 512 DSPs, and 456 BRAMs, whereas the ZCU111 implementation utilizes 158,760 FFs, 314,218 LUTs, 2,048 DSPs, and 627 BRAMs.

Shao *et al.* [16] proposed an FPGA-based accelerator for EfficientViT that supports various operations, including lightweight convolutions and attention. The proposed design employs a reconfigurable architecture capable of efficiently handling depthwise, pointwise, and generic convolutions, as well as lightweight multi-scale attention. In addition, a time-multiplexed and pipelined dataflow is introduced to enable both inter-layer and intra-layer computation fusion, thereby improving hardware utilization and reducing off-chip memory access. Experiments conducted on the Xilinx ZCU102 board show that the accelerator achieves a throughput of 780.2

GOPS with an energy efficiency of 105.1 GOPS/W. In our previous work (Sehairi and Bouwmans [17]), we proposed a comprehensive hardware–software co-design framework for accelerating high-resolution (4K) image processing applications on the low-cost PYNQ-Z1 platform based on the Zynq-7000 SoC and the Vitis Vision Library. The approach leverages the tight integration between the ARM PS and the FPGA PL by offloading computationally intensive operations, such as edge detection, corner detection, and image filtering, to custom hardware accelerators generated using Vitis HLS, while high-level control and data orchestration are managed by the ARM processor via AXI interfaces. This PS–PL co-design effectively mitigates the latency and bandwidth limitations inherent to CPU-only implementations when handling large image data. Experimental results on real 4K images demonstrated substantial performance gains, with speedups ranging from 30× to 49× for algorithms including Sobel and Harris corner detection, while maintaining moderate resource utilization and power consumption.

In 2025, Jiang *et al.* [18] provided a comprehensive review of FPGA-based CNN accelerators, addressing the computational challenges of deep learning through a robust performance evaluation framework. Their work systematically analyzes key metrics, including latency, throughput, and compute efficiency, to enable fair comparisons across varying hardware architectures. The study further explores critical optimization strategies, such as parallel computing, loop tiling, and model compression techniques like pruning and quantization, alongside hardware–software co-design methodologies. Additionally, it identifies future research opportunities in design space exploration to enhance adaptability and performance in resource-constrained environments. Rao *et al.* [19] proposed a scalable FPGA-based architecture for accelerating real-time image processing through parallel VLSI pipelines, effectively integrating core functions such as convolution, Sobel edge detection, histogram equalization, and adaptive thresholding. Their design optimizes energy efficiency and throughput by employing modular hardware units that utilize shared memory buffers, fixed-point arithmetic, and clock gating techniques. Implemented on a Xilinx Kintex-7 FPGA, the system achieves pixel-level parallelism with a processing speed of up to 180 FPS for 256×256 grayscale images.

Gdaim and Mtibaa [20] proposed a high-performance design methodology for accelerating CNNs on FPGA platforms using OpenCL to bridge the gap between algorithmic complexity and hardware efficiency. Their approach optimizes the computationally intensive convolutional layers by employing burst memory transfers and parallel banking strategies to minimize off-chip memory access latency on Xilinx Zynq-7000 SoC devices. Experimental evaluations on MNIST and CIFAR-10 datasets yielded recognition rates of 97.98% and 86%, respectively, demonstrating a significant improvement over standard implementations. Furthermore, the authors reported that their accelerator achieved a 10-fold speedup compared to single-core CPU execution while maintaining a low power consumption of 2.23 W, effectively balancing accuracy and resource utilization.

3. METHOD

The convolution accelerator was developed through a three-stage hardware–software workflow. First, the accelerator IP was implemented in C++ using Vitis HLS and the Vitis Vision Library, with pipelined and parallelized image processing between the PS and PL. Second, the generated IP was integrated in Vivado with the PS, DDR memory, and AXI DMA engines, followed by synthesis and generation of the hardware overlay. Finally, a Python/PYNQ software layer was developed in Jupyter Notebook to load the overlay, initialize the DMA engines, allocate shared-memory buffers, and execute the complete convolution pipeline, as illustrated in Figure 1.

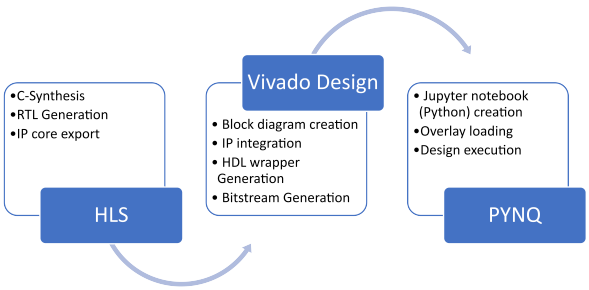


Figure 1. Step-by-step methodology for custom IP integration on Zynq SoCs

3.1. High-level synthesis-based design of the convolution accelerator intellectual property

The HLS-based convolution accelerator was implemented in C++ using the AMD Xilinx Vitis Vision Library as a streaming two-dimensional convolution core for real-time and high-resolution image processing. Synthesis-time parameters define the maximum image resolution, kernel dimensions, input/output formats, and pixel-processing granularity. Two parallelism modes were considered: `XF_NPPC1`, which processes one pixel per clock cycle for resource-constrained operation and `XF_NPPC8`, which processes eight pixels per cycle for higher throughput. Accordingly, the native data-path widths are 24 and 8 bits for NPPC1 RGB and grayscale, respectively, and 192 and 64 bits for NPPC8 RGB and grayscale.

To bridge the AXI4-Stream interface and the internal data structures required by the Vitis Vision kernels, dedicated conversion modules are developed. The input adapter translates the incoming AXI4-Stream pixel stream into an xf::cv::Mat representation by sequentially mapping streamed pixel data to matrix elements. Loop pipelining directives are applied to sustain an initiation interval of one clock cycle, thereby preserving a continuous dataflow and maximizing throughput. Dimension checks are incorporated to ensure that runtime image sizes remain within the bounds defined at synthesis time.

Similarly, an output adapter is designed to convert the processed xf::cv::Mat back into an AXI4-Stream format. Each processed pixel word, containing one pixel in `XF_NPPC1` or eight pixels in `XF_NPPC8`, is forwarded through the AXI4-Stream output interface. This behavior aligns with DMA-based data transfers, where frame-level signaling is required, and avoids unnecessary protocol overhead associated with video-specific interfaces.

The top-level HLS function encapsulates the complete accelerator functionality and defines all external hardware interfaces. AXI4-Stream ports are instantiated for image input, image output, and filter-parameter streaming, enabling independent high-bandwidth data paths driven by the two AXI DMA engines. AXI4-Lite registers provide runtime configurability for parameters such as image dimensions and normalization shift. In contrast to conventional memory-mapped coefficient access, the convolution filter coefficients are supplied through a dedicated AXI4-Stream interface, allowing dynamic kernel updates without re-synthesis or static memory binding. By enabling the DATAFLOW optimization directive, the design permits input adaptation, convolution processing, and output adaptation stages to execute concurrently, thereby overlapping communication and computation and sustaining continuous streaming throughput.

Within this top-level function, the convolution operation is implemented using the 'xf::cv::filter2D' primitive from the Vitis Vision Library. This kernel performs a constant-border $N \times N$ convolution with externally streamed coefficients and a programmable scaling shift. The modular structure of the HLS design facilitates rapid architectural exploration across scalar (`XF_NPPC1`) and vector-parallel (`XF_NPPC8`) configurations, while HLS synthesis automatically infers loop pipelining and spatial parallelism to meet the targeted throughput constraints.

3.1.1. Algorithmic dataflow, buffering, and NPPC8 parallelism

Algorithm 1 summarizes the streaming execution of the proposed convolution accelerator. At the top level, the architecture is decomposed into three processing stages: AXI4-Stream input adaptation, two-dimensional convolution, and AXI4-Stream output adaptation.

These stages are enclosed by the HLS `DATAFLOW` directive and therefore operate concurrently after the initial pipeline-filling period. The input adapter continuously receives packed pixel words from the image AXI4-Stream interface and converts them into the `xf::cv::Mat` representation consumed by the convolution kernel. In parallel, the convolution stage processes previously received data, while the output adapter converts completed results back to AXI4-Stream format. Both stream-adaptation loops are pipelined with an initiation interval of one (`II=1`), allowing one packed input or output word to be accepted or produced per clock cycle when the downstream and upstream interfaces are ready. The output adapter asserts `TLAST` only for the final packed word of the frame, which provides the frame-level termination required by the AXI DMA.

The convolution stage relies on spatial data reuse rather than repeatedly retrieving a complete neighborhood from external DDR memory for every output pixel. For an $N \times N$ filter, previously received image rows are retained in on-chip line buffers, while a local sliding window maintains the pixels required for the current convolution neighborhood. Conceptually, $N - 1$ previous rows are retained; therefore, a 7×7 configuration requires the preceding six rows together with the current incoming row. As the image stream advances, the window is shifted and only newly arriving pixel data are inserted, whereas pixels shared by consecutive convolution windows are reused directly from on-chip storage. Consequently, external memory is used primarily

for sequential frame input and output rather than for repeated neighborhood accesses. After the line buffers and convolution window have been initialized, this organization allows the pipeline to sustain continuous processing. Constant-border handling is performed by the Vitis Vision `filter2D` primitive, while the streamed filter coefficients and the normalization shift determine the convolution operation applied to the complete frame [21].

Algorithm 1. Streaming FPGA convolution pipeline for NPPC1 and NPPC8

Require: Image stream S_I, coefficient stream S_K, image dimensions $H \times W$, filter size N, normalization shift s, parallelism $P \in \{1, 8\}$
Ensure: Filtered image stream S_O
$N_w \leftarrow W/P$ ▷ Packed words per image row
Receive the $N \times N$ coefficients from S_K
Initialize line buffers and sliding-window storage

DATAFLOW: execute Stages A--C concurrently

Stage A: AXI4-Stream to xf::cv::Mat
for $r = 0$ to $H - 1$ **do**
  for $w = 0$ to $N_w - 1$ **do**
    $V \leftarrow$ read one packed word from S_I
    Write the P packed pixels of V to the input stream ▷ Pipelined, II=1
  end for
end for

Stage B: Streaming 2-D convolution
for each incoming packed pixel vector V **do**
  Update line buffers with the newly received pixels
  Shift/update the local $N \times N$ convolution neighborhood
  for all $p = 0$ to $P - 1$ **in parallel do**
    for all image channels c **in parallel do**
      $acc_{p,c} \leftarrow 0$
      for $i = 0$ to $N - 1$ **do**
        for $j = 0$ to $N - 1$ **do**
          $acc_{p,c} \leftarrow acc_{p,c} + K[i, j]\, X_{p,c}[i, j]$
        end for
      end for
      $Y_{p,c} \leftarrow \mathrm{scale_and_cast}(acc_{p,c}, s)$
    end for
  end for
  Pack $(Y_0, \ldots, Y_{P-1})$ into one output word
end for

Stage C: xf::cv::Mat to AXI4-Stream
for $r = 0$ to $H - 1$ **do**
  for $w = 0$ to $N_w - 1$ **do**
    Read one packed filtered word
    TLAST $\leftarrow (r = H - 1) \wedge (w = N_w - 1)$
    Write the packed word to S_O ▷ Pipelined, II=1
  end for
end for

The degree of pixel-level parallelism is determined by the `NPPC` parameter. With `XF_NPPC1`, one pixel is represented and processed in each pipeline iteration. With `XF_NPPC8`, eight horizontally adjacent pixels are packed into a single processing word and are processed concurrently. Accordingly, an 8-bit RGB configuration requires a native stream width of 8×3×8=192 bits, whereas an 8-bit grayscale configuration requires 8×8=64 bits. For NPPC8, the number of processing words per image row is therefore reduced from W to $W/8$. Each incoming packed word advances the convolution state by eight pixels, and the convolution engine logically evaluates eight strongly overlapping spatial neighborhoods in parallel. Because adjacent neighborhoods share most of their source pixels, the line-buffer and sliding-window organization allows these samples to be reused instead of maintaining eight independent image buffers.

Arithmetic parallelism is increased correspondingly. For a 7×7 RGB convolution, each output pixel requires 49 coefficient multiplications for each of the three color channels. The NPPC1 architecture therefore

exposes 49×3=147 parallel multiply–accumulate paths, whereas NPPC8 extends this processing to eight pixel lanes, corresponding to 8×49×3=1176 parallel multiplication paths. Similarly, a 3×3 grayscale convolution requires 9 coefficient multiplications per pixel and scales from 9 arithmetic paths in NPPC1 to 8×9=72 in NPPC8. This spatial replication explains the substantial increase in DSP utilization observed for NPPC8 while also enabling up to eight output pixels to progress through the convolution pipeline per clock cycle after the initial buffering and pipeline-filling latency.

Figure 2 summarizes the architecture and dataflow of the proposed AXI4-Stream-based image convolution accelerator implemented on FPGA. The PS configures the accelerator through AXI4-Lite registers by providing the image dimensions and normalization factor, and initiates execution. Input image data are transferred from external DDR memory to the PL via an AXI DMA using the MM2S channel, while convolution kernel coefficients are streamed directly to the accelerator through a dedicated AXI4-Stream interface. Inside the PL, the incoming pixel stream is adapted to an `xf::cv::Mat` representation, buffered using on-chip BRAM line buffers, and processed by a fully pipelined convolution engine based on the Vitis Vision `filter2D` kernel. The design exploits task-level dataflow and loop pipelining to achieve high throughput after pipeline filling. The processed image is streamed back to memory through the S2MM DMA channel, enabling efficient high-throughput image filtering with low latency and reduced CPU involvement.

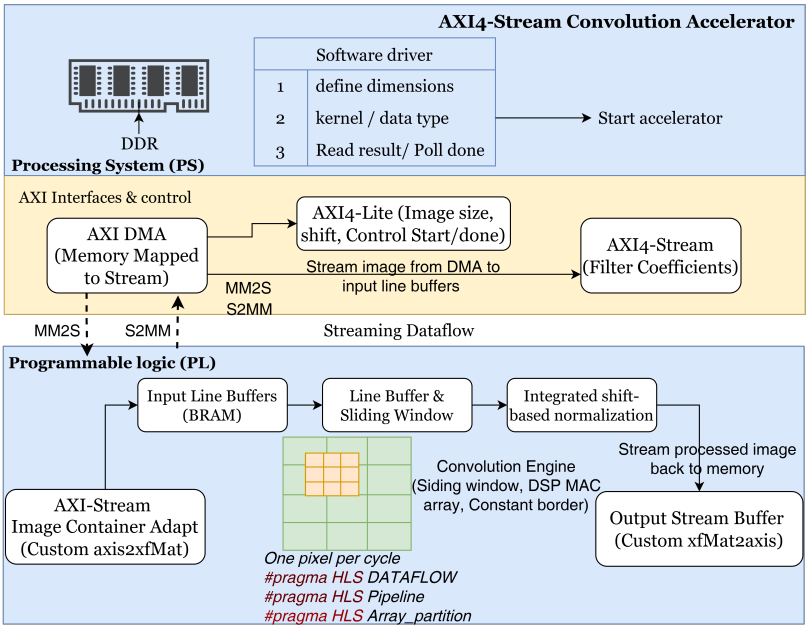


Figure 2. High-level synthesis of the FPGA convolution accelerator core

The proposed accelerator was developed using AMD Xilinx Vitis™ HLS, where the design functionality is described in C/C++ and synthesized into register-transfer level (RTL) hardware. During the C simulation phase, the functional correctness of the design is validated using the Vitis HLS simulation flow. The C synthesis phase translates the high-level description into an optimized RTL implementation, generated in VHDL, while enforcing the specified interface pragmas, loop pipelining, and dataflow directives. Following synthesis, C/RTL co-simulation is performed to verify the functional behavior of the generated RTL against the original C/C++ testbench while providing cycle-level RTL execution information. Finally, the generated RTL is packaged as a reusable IP core and exported for system-level integration in the Vivado® Design Suite.

3.1.2. Functional verification and C/RTL co-simulation

Functional verification was carried out using a self-checking C/C++ testbench before and after HLS synthesis. The verification set included the 512×512 RGB Lena test image used in the filter-validation experiments, evaluated with the Emboss, Sobel, Prewitt, Sharpen, and Strong Sharpen coefficient sets. For each test case, the testbench loaded the input image, configured the corresponding convolution coefficients and normalization parameters, executed the accelerator function, and generated the accelerator output.

During C simulation, the accelerator output was compared pixel-by-pixel against a software reference generated using OpenCV `filter2D`. The software reference used the same convolution coefficients as the accelerator, with `ddepth=-1` and constant zero-padding (`BORDER_CONSTANT`), consistent with the constant-border behavior of the Vitis Vision `filter2D` implementation. Functional correctness was accepted only when all output pixels matched the OpenCV reference, resulting in zero mismatched pixels. Otherwise, the testbench returned a nonzero status and the simulation was considered failed.

Following successful C simulation and C synthesis, C/RTL co-simulation was performed using the same input image, convolution coefficients, image dimensions, normalization parameters, and self-checking C/C++ testbench. During this stage, Vitis HLS executed the generated RTL implementation against the original testbench to verify that the synthesized hardware preserves the functional behavior established during C simulation. Co-simulation was considered successful only when the RTL execution completed with a `PASS` status and satisfied the same zero-mismatch correctness criterion. This verification step therefore confirms the functional equivalence of the synthesized RTL implementation before the accelerator is packaged as an IP core and integrated into the Vivado system design.

For C simulation and C/RTL co-simulation on the Windows x86-64 host, OpenCV 4.4.0 was compiled for x86-64 using the MinGW toolchain and linked to the Vitis HLS C/C++ testbench. This host-side OpenCV installation was used exclusively for HLS verification and is distinct from the OpenCV 4.10.0 installation used by the Python/PYNQ runtime on the Kria platform during system-level execution and experimental evaluation.

3.2. Vivado block design and system integration

In this section, we describe the system-level integration of the proposed convolution accelerator within the Vivado Design Suite, highlighting the interaction between the PS, PL, memory subsystem, and AXI-based interconnects.

3.2.1. Processing system and control interface

At the system level, the design is anchored by the Zynq UltraScale+ MPSoC, which integrates quad-core ARM Cortex-A53 processors and FPGA fabric. The PS provides clocking and reset signals to the PL through `pl_clk0` and `pl_resetn0`, ensuring synchronous operation across all hardware modules. The PS also exposes high-performance AXI master ports (`M_AXI_HPM0_FPD` and `M_AXI_HPM1_FPD`) for control-plane communication and AXI slave high-performance ports (`S_AXI_HPx_FPD`) for data-plane memory access.

Control and configuration of the convolution accelerator are performed through AXI4-Lite interfaces, which provide a low-latency and register-oriented communication channel between the PS and PL. The `ps8_0_axi_periph` AXI interconnect bridges the PS AXI master ports to the AXI4-Lite slave interfaces of both the custom `Filter2d_accel` IP core and the AXI DMA engines. Using this interface, the PS configures key accelerator parameters, including image dimensions, normalization shift values, and execution control signals, and also initiates and monitors DMA transfer operations. By decoupling control-plane communication from high-throughput data movement, this architecture enforces a clear separation between control and data paths, thereby improving system scalability and maintainability in accordance with recommended design methodologies for Vivado IP Integrator [22], [23].

3.2.2. Data movement, stream adaptation and data width conversion

High-throughput data movement between DDR memory and the convolution accelerator is implemented using two AXI DMA engines. The first AXI DMA instance operates in both memory-to-stream (MM2S) and stream-to-memory (S2MM) modes: the MM2S channel transfers input image data from DDR memory to the PL, while the S2MM channel writes the processed output stream back to DDR memory. The second AXI DMA instance is dedicated to streaming convolution kernel coefficients (and optional auxiliary parameters) from memory to the accelerator via an independent MM2S channel.

Both DMA engines are configured and controlled through AXI4-Lite interfaces and use AXI4 memory-mapped master ports connected to the PS high-performance (HP) ports to sustain aligned burst transactions. This dual-DMA organization enables concurrent and independent streaming of image data and filter parameters while preserving high-bandwidth, deterministic memory transfers.

The streamed data is forwarded to the accelerator via AXI4-Stream interfaces, enabling a fully pipelined, continuous dataflow architecture. This streaming paradigm minimizes PS intervention during execution and allows the PL accelerator to operate at steady-state throughput once the pipeline is filled.

To guarantee protocol compatibility and efficient bandwidth utilization between the AXI DMA engines and the HLS-generated convolution IP, two AXI4-Stream data width converter (DWC) IP cores are inserted along the read and write-back paths. These converters reconcile the bus-level data width imposed by the DMA subsystem with the pixel-packing format required by the convolution engine, see Figure 3.

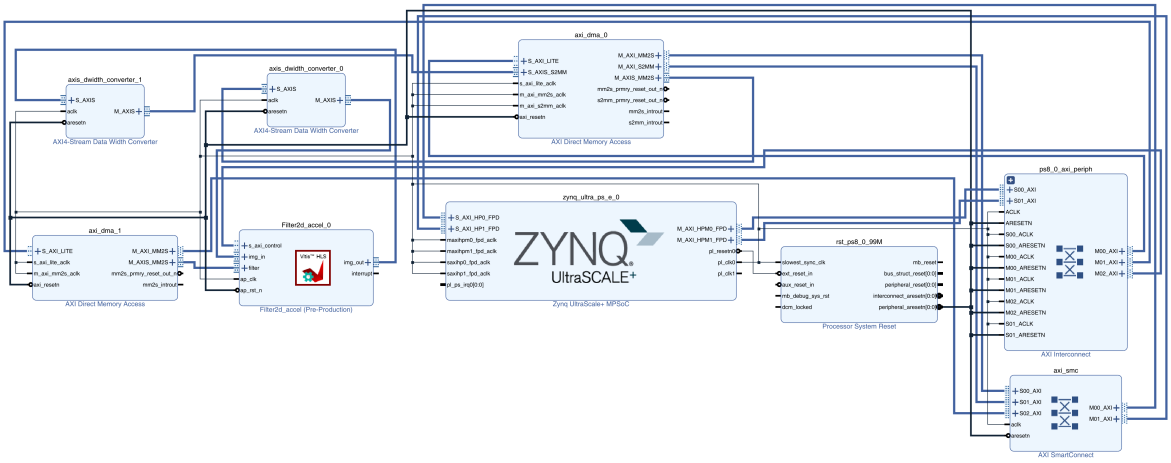


Figure 3. Vivado block design of the proposed AXI4-Stream–based image convolution accelerator on the Kria KV260 platform

In the input data path, the AXI4-Stream DWC adapts the incoming stream to the processing granularity defined by the selected parallelization mode. In the scalar `XF_NPPC1` configuration, the DWC receives a 32-bit (4-byte) stream for RGB images and a 16-bit (2-byte) stream for grayscale images at its slave interface. These widths are dictated by DMA alignment and AXI burst granularity rather than by intrinsic pixel size. The converter then reshapes the stream to the native accelerator representation, producing a 24-bit (3-byte) stream for RGB and an 8-bit (1-byte) stream for grayscale at the master interface. A second DWC in the write-back path performs the inverse adaptation before data is returned to the PS memory subsystem.

Transitioning to the vector-parallel `XF_NPPC8` architecture requires a substantially wider interface to maintain proportional throughput. In this mode, the read-path DWC accepts a 256-bit (32-byte) AXI4-Stream for RGB processing and converts it to the native 192-bit (24-byte) representation expected by the accelerator. The remaining bytes in each 256-bit word correspond to padding introduced to satisfy AXI alignment constraints and to maximize burst efficiency during DMA transfers. By localizing this width adaptation at the stream boundary, the architecture preserves computational efficiency within the accelerator core, avoiding superfluous arithmetic or control logic.

For the `XF_NPPC8` grayscale configuration, two architectural options are explored. Since the native 64-bit (8-byte) pixel vector width is already a power-of-two and naturally aligned with standard bus widths, the DWCs are not strictly required for alignment purposes and may be omitted to reduce resource utilization and routing complexity. Alternatively, to fully exploit the available memory bandwidth, the DWCs can be retained and configured to interface with a 256-bit (32-byte) DMA stream, implementing a 4:1 width reduction toward the 64-bit accelerator input and a reciprocal 1:4 expansion in the write-back path. The experimental results reported for the `XF_NPPC8` grayscale case correspond to the former, resource-optimized configuration.

Collectively, these data width conversion stages enable sustained high-throughput operation for both the scalar `XF_NPPC1` and vector-parallel `XF_NPPC8` modes. This alignment between PS-side memory bandwidth and PL-side computational parallelism ensures efficient streaming utilization and balanced system-level performance.

3.2.3. Convolution accelerator intellectual property

The core computation is performed by the `Filter2d_accel` IP, generated using Vitis HLS and the Vitis Vision library, as described in subsection 3.1. The accelerator exposes AXI4-Stream slave and master interfaces for input and output image streams, respectively, and an AXI4-Stream interface for filter coefficient

input. Internally, the IP implements a fully pipelined convolution dataflow, consisting of AXI4-Stream-to-`xf::cv::Mat` conversion, convolution processing using the `filter2D` primitive, and conversion back to AXI4-Stream output.

3.2.4. Reset and synchronization logic

During operation, the PS configures and coordinates the accelerator and DMA engines through AXI4-Lite, while image data and filter coefficients are transferred through the corresponding streaming paths. The detailed software-controlled execution and synchronization procedure is described in subsection 3.3.

3.3. Python-based control and execution flow on the Kria platform

To enable rapid prototyping, control, and performance evaluation of the proposed convolution accelerator, a Python-based runtime environment was developed using a Jupyter Notebook executing on the Kria PS. This software layer leverages the PYNQ framework to interface with the hardware accelerator instantiated in the PL.

The execution flow begins by loading the synthesized hardware design, exported from the Vivado Design Suite as a bitstream and associated hardware handoff file (`.bit`, `.hwh`), through the PYNQ `Overlay` class. This operation configures the PL fabric and automatically maps the AXI4-Lite control interfaces and AXI4 memory-mapped DMA registers into the PS address space. Consequently, the custom `Filter2d_accel` IP core and both AXI DMA engines (image data and filter-parameter DMA) become programmatically accessible from Python. Once instantiated, these objects enable direct control of register configuration, buffer allocation, and coordinated MM2S/S2MM transaction management within the streaming PS–PL pipeline.

3.3.1. Memory allocation, buffer preparation, and AXI4-lite parameter configuration

Input and output image buffers are allocated in physically contiguous DDR memory using the PYNQ `allocate` API, which provides DMA-compatible memory regions accessible by both the PS and PL. The buffers are instantiated with carefully selected data types and alignment constraints to satisfy AXI burst requirements and ensure seamless interoperability with the AXI DMA engines.

For the primary AXI DMA instance (MM2S/S2MM), the input buffer is populated with image data formatted in RGB or grayscale representation, consistent with the pixel-packing scheme defined by the HLS-based convolution accelerator IP. The corresponding output buffer is pre-allocated to store the processed image data returned by the accelerator via the streaming write-back path.

A second AXI DMA engine is dedicated to the transfer of convolution kernel coefficients and associated parameters. This separation of image and filter data paths enables flexible runtime reconfiguration of the convolution operator while preserving high-throughput streaming and deterministic memory transactions.

Before computation begins, the convolution accelerator is configured through its AXI4-Lite control interface. Using memory-mapped register access, the PS programs essential parameters such as image width, image height, and the normalization shift factor applied after convolution. These parameters directly correspond to the control ports defined in the HLS design and allow the same hardware IP to process images of varying resolutions without re-synthesis.

3.3.2. DMA-based streaming execution, synchronization and completion handling

Once the configuration phase is completed, both AXI DMA engines are programmed to initiate their respective transfers. For the primary DMA instance, the memory-to-stream (MM2S) channel conveys the input image from DDR memory to the convolution accelerator via the AXI4-Stream interface, while the stream-to-memory (S2MM) channel captures the processed output stream and writes it back to the designated output buffer in DDR memory. In parallel, the second AXI DMA engine performs the MM2S transfer of convolution kernel coefficients (and associated parameters) from memory to the accelerator through a dedicated AXI4-Stream path.

On the software side, the PS monitors the status registers of both DMA engines to guarantee correct synchronization between parameter loading, image transfer, and hardware execution. The convolution task is considered complete only after the filter-parameter MM2S transfer and both image-related MM2S/S2MM transactions have successfully finished. This polling- or interrupt-based synchronization mechanism ensures memory consistency and data coherence before any subsequent post-processing, validation, or visualization stage is initiated.

3.3.3. AXI4-stream throughput and latency validation

The AXI4-Stream integration was validated at both the functional and system levels. At the accelerator level, the input and output stream adaptation loops operate with an initiation interval of one (`II=1`), allowing one packed NPPC1 or NPPC8 word to be accepted or produced per clock cycle after pipeline filling. Correct frame termination was verified through the assertion of `TLAST` on the final packed output word.

At the system level, each accelerator invocation was considered complete only after the filter-parameter MM2S transfer and the image MM2S/S2MM transactions had successfully finished. End-to-end latency was measured from accelerator and DMA initiation until completion of all required DMA transactions, thereby including the communication overhead introduced by the AXI4-Stream data path. Effective frame throughput was computed as $1/t_{\mathrm{E2E}}$, while effective pixel throughput can be expressed as HW/t_{E2E}, where H and W denote the image dimensions. Measurements were repeated over seven independent runs with ten executions per run, using pre-allocated DMA-compatible buffers to ensure reproducibility.

3.3.4. Result retrieval and validation

Finally, the output buffer stored in DDR memory is accessed from the Python environment for visualization, storage, or quantitative evaluation of the results. The processed image can be displayed directly within the Jupyter Notebook using the Python Imaging Library (Pillow), or compared against a software reference implementation, such as the `filter2D` function from OpenCV executed on the ARM Cortex-A53 processor, to verify functional correctness and to quantify performance improvements, as discussed in subsequent sections. Figure 4 illustrates the corresponding notebook execution flowchart.

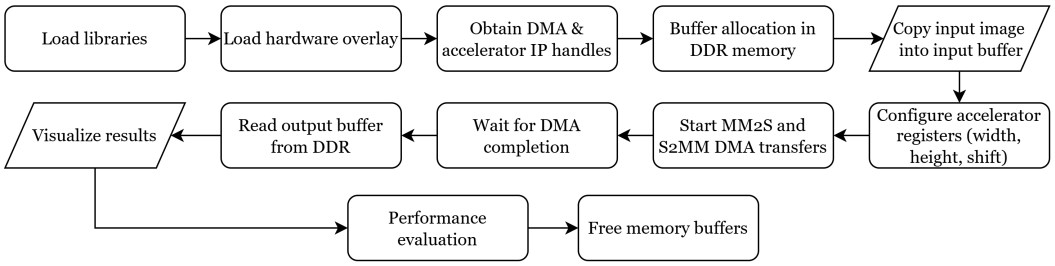


Figure 4. Python-based execution flow of the AXI4-Stream convolution accelerator on the Kria platform using the PYNQ framework

4. RESULTS AND DISCUSSION

Our accelerator was implemented using the Vivado Design Suite and Vitis HLS 2023.2.2 and evaluated on the AMD Xilinx Kria KV260 Vision AI Starter Kit at a clock frequency of 100 MHz. The KV260 is based on the Zynq™ UltraScale+™ MPSoC, integrating a PS comprising a quad-core ARM Cortex-A53, a dual-core Cortex-R5F real-time processing unit (RPU), and an ARM Mali™-400 MP2 GPU. The PL provides approximately 256 K logic cells, 144×36 Kb Block RAMs (BRAM), about 1.2 K DSP slices, and 64×288 Kb dual-port 72-bit UltraRAM blocks, alongside 4 GB of DDR4 memory [24], [25]. We validated the accelerator using multiple convolution kernels across several image resolutions on this platform. Notably, a single synthesized hardware design supports seven custom filters via an AXI4-Stream-based streaming architecture: Emboss, Sobel (vertical and horizontal), Prewitt (vertical and horizontal), Sharpen, and Strong Sharpen.

The experimental configuration used to evaluate the proposed FPGA–CPU convolution accelerator is summarized in Table 1. The same Kria KV260 platform was used for both the PS-only software baseline and the heterogeneous PS–PL implementation. The software baseline was executed on the ARM Cortex-A53 processor using the OpenCV `filter2D` function, whereas the accelerated implementation offloaded the convolution operation to the PL and used AXI DMA for image and coefficient transfers. Input and output buffers were allocated in physically contiguous DDR memory through the PYNQ `allocate` API before timing measurements. Reported latency corresponds to repeated execution using pre-allocated buffers and includes the DMA transfers required to feed and drain the hardware accelerator.

Table 1. Hardware, software, and runtime configuration used for experimental evaluation

Category	Parameter	Configuration
Hardware	Evaluation platform	AMD Xilinx Kria KV260 Vision AI Starter Kit
	PS	Zynq UltraScale+MPSoC, quad-core ARM Cortex-A53
	DDR memory	4 GB DDR4
	Available PL resources	117,120 LUTs; 234,240 FFs; 144 BRAM36; 1,248 DSP slices
	Accelerator clock frequency	100 MHz
	Data-transfer engines	Two AXI DMA instances: image MM2S/S2MM and filter-parameter MM2S
Software	FPGA design tools	Vivado Design Suite 2023.2.2 and Vitis HLS 2023.2.2
	Vision library	AMD Xilinx Vitis Vision Library 2023.2
	Runtime framework	PYNQ, Jupyter Notebook
	Programming language	C++ and Python 3.10.12
	PYNQ image	3.0.1
	Host OpenCV for HLS verification	OpenCV 4.4.0, x86-64 MinGW
	Embedded OpenCV runtime	OpenCV 4.10.0 on Kria/PYNQ
	Supporting libraries	NumPy and Pillow
	Hardware loading	PYNQ `Overlay` using `.bit` and `.hwh` files
Runtime	Accelerator configurations	`XF_NPPC1` and `XF_NPPC8`
	Pixel parallelism	1 pixel per clock cycle (NPPC1); 8 pixels per clock cycle (NPPC8)
	Input formats	8-bit grayscale (1 channel) and 24-bit RGB (3×8-bit channels)
	Evaluated resolutions	512×512, HD (1280×720), FHD (1920×1080), and 4K (3840×2160)
	Kernel sizes	3×3, 5×5, and 7×7
	Evaluated filters	Emboss, Sobel-H/V, Prewitt-H/V, Sharpen, and Strong Sharpen
	Image-buffer datatype	Unsigned 8-bit integer (`uint8`)
	Filter-coefficient datatype	Signed 16-bit integer (`int16`)
	Runtime configuration	Image rows, columns, and normalization shift through AXI4-Lite
	Data movement	AXI4-Stream with DMA-based MM2S/S2MM transfers
	CPU baseline	OpenCV `filter2D`, `ddepth=-1`, constant zero-padding (`BORDER_CONSTANT`), executed on Cortex-A53
Measurement	Timing repetitions	7 independent runs ×10 executions per run (70 measurements per configuration)
	Reported statistic	Mean execution time and standard deviation
	FPGA timing scope	End-to-end accelerator invocation, including image MM2S/S2MM transfers, filter-parameter MM2S transfer, accelerator start, and DMA completion synchronization
	CPU timing scope	OpenCV `filter2D` execution, excluding image loading and file I/O
	Excluded FPGA overhead	Overlay loading and DMA-buffer allocation performed before timing

Figure 5 illustrates the evaluated convolution operations on the RGB Lena image. Figure 5(a) shows the original image, while Figure 5(b) presents the Emboss result. Figures 5(c) and (d) show the Sobel vertical and horizontal responses, respectively, and Figures 5(e) and (f) show the corresponding Prewitt responses. The Sobel outputs exhibit stronger prominent boundaries because of the higher central weighting of the Sobel masks. Figures 5(g) and (h) show the Sharpen and Strong Sharpen results, with the latter producing greater edge contrast and more visible fine texture. Overall, the results demonstrate runtime switching among edge detection, embossing, and sharpening operations without modifying the accelerator architecture.

Table 2 summarizes the Vitis HLS resource utilization of the proposed 2D convolution accelerators on the Kria KV260, reported relative to the available fabric (117,120 LUTs, 234,240 FFs, 144 BRAMs, and 1,248 DSPs). Across all configurations, the implementations remain logic-efficient: LUT usage spans from 2.79% for the grayscale 3×3 kernel to 7.09% for the RGB 7×7 kernel, while FF utilization ranges from 1.15% to 5.77%. In contrast, increasing kernel size and color depth leads to a more pronounced growth in on-chip memory and arithmetic resources, consistent with the additional buffering requirements and the higher multiply–accumulate workload. For instance, BRAM consumption increases from 3 (2.08%) in the grayscale 3×3 case to 42 (29.16%) for RGB 7×7, and DSP usage rises from 9 (0.72%) to 147 (11.78%).

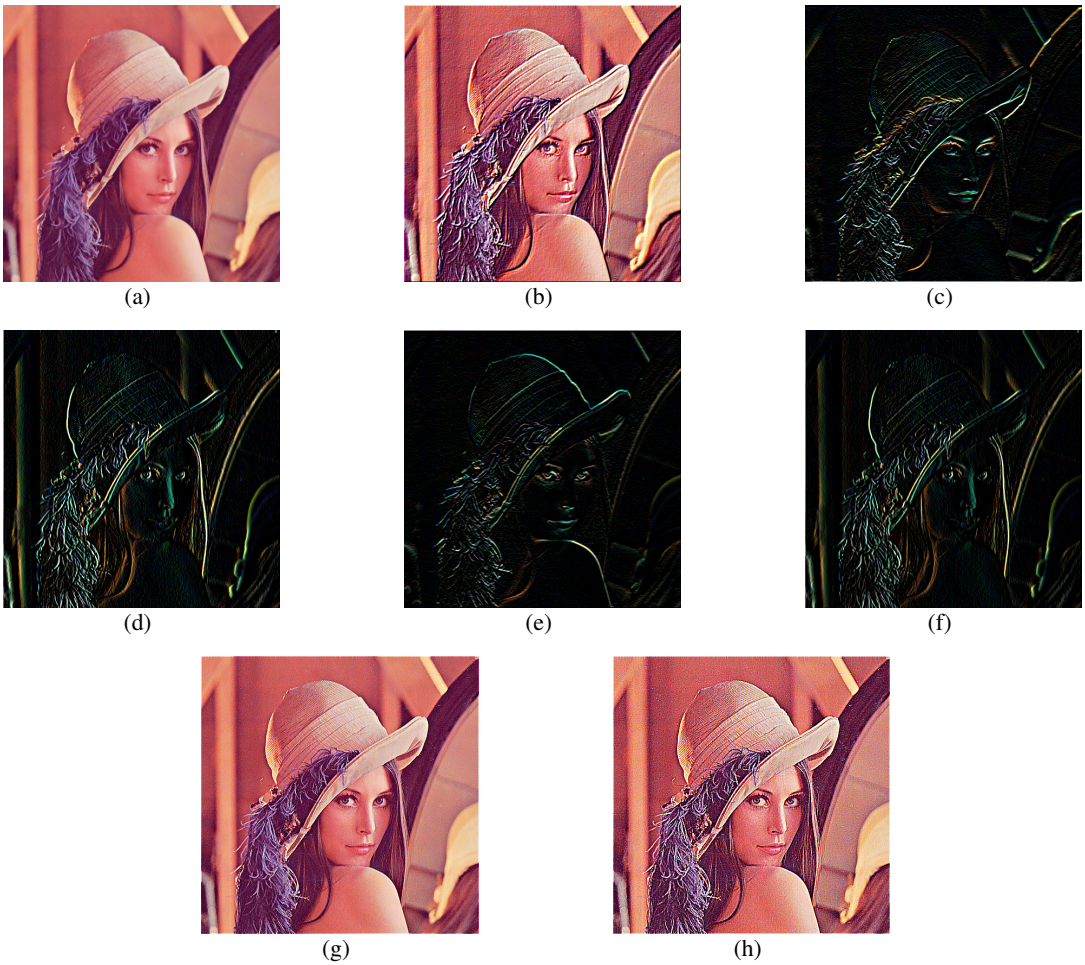


Figure 5. PS–PL implementation results of the evaluated 2D convolution operations applied to the RGB Lena image; (a) original, (b) emboss, (c) sobel vertical, (d) sobel horizontal, (e) prewitt vertical, (f) prewitt horizontal, (g) sharpen, and (h) strong sharpen

Table 2. Vitis HLS resource-utilization estimates for the NPPC1 2D convolution accelerator on the AMD Xilinx Kria KV260 Vision AI Starter Kit

Configuration	Resources	LUT	FF	BRAM36	DSP
—	Available	117120	234240	144	1248
Grayscale 3×3	Used	3274	2702	3	9
	Utilization (%)	2.79	1.15	2.08	0.72
Grayscale 5×5	Used	4232	4992	5	25
	Utilization (%)	3.61	2.13	3.47	2.00
RGB 3×3	Used	3862	3284	18	27
	Utilization (%)	3.29	1.40	12.50	2.16
RGB 5×5	Used	5406	7354	30	75
	Utilization (%)	4.61	3.14	20.83	6.01
RGB 7×7	Used	8308	13521	42	147
	Utilization (%)	7.09	5.77	29.16	11.78

Overall, the results demonstrate substantial remaining capacity on the KV260, while highlighting BRAM and DSP scaling as the primary limiting factors when targeting larger RGB kernels. When comparing the Vitis resource utilization of our Full HD grayscale 3×3 convolution accelerator against the AMD Vitis Libraries (2025.2) implementation reported on the AMD Technical Information Portal [21], we note that both designs

exhibit identical BRAM and DSP requirements (3 BRAM, 9 DSP), indicating comparable buffering and multiply–accumulate structures for a 3×3 kernel. In contrast, our design requires 3274 LUTs and 2702 FFs, versus 1161 LUTs and 1701 FFs for the AMD reference, corresponding to 2.82× and 1.59× increases, respectively. We attribute this overhead primarily to differences in coefficient delivery: our coefficients are provided via an AXI4-Stream interface `axis`, which introduces additional stream control and synchronization logic, whereas the AMD design fetches coefficients through an AXI4 memory-mapped interface (`m_axi`), consistent with a memory-to-stream pipeline organization commonly used in Vitis convolution reference designs.

Table 3 summarizes the post-implementation resource utilization of the PS–PL accelerated 2D convolution pipeline on the AMD Xilinx Kria KV260 Vision AI Starter Kit. The results highlight a distinct shift in resource pressure as the design scales from scalar to vector-parallel processing.

Table 3. Vivado post-implementation resource utilization for the complete PS–PL accelerated 2D convolution on the AMD Xilinx Kria KV260 Vision AI Starter Kit across different NPPC configurations

NPPC	Configuration	Resources	LUT	LUTRAM	FF	BRAM36	DSP	BUFG
—	—	Available	117120	57600	234240	144	1248	352
NPPC1	Grayscale 3×3	Used	9585	1553	13602	7.5	9	1
		Util. (%)	8.18	2.70	5.81	5.21	0.72	0.28
	Grayscale 5×5	Used	10008	1652	14675	7.0	25	1
		Util. (%)	8.55	2.87	6.26	4.86	2.00	0.28
	RGB 3×3	Used	10262	1582	14722	15.0	27	1
		Util. (%)	8.76	2.75	6.29	10.42	2.16	0.28
	RGB 5×5	Used	11228	1769	16981	21.0	75	1
		Util. (%)	9.59	3.07	7.25	14.58	6.01	0.28
	RGB 7×7	Used	12872	2090	20280	27.0	147	1
		Util. (%)	10.99	3.63	8.66	18.75	11.78	0.28
NPPC8	Grayscale 3×3	Used	11432	1770	15398	10.0	72	1
		Util. (%)	9.76	3.07	6.57	6.94	5.77	0.28
	RGB 7×7	Used	34864	3192	47190	39	1176	14
		Util. (%)	29.77	5.54	20.15	27.08	94.23	3.98

Note: BRAM is reported in Vivado BRAM36 units. Xilinx devices have BRAM18 and BRAM36 primitives; fractional values occur because a BRAM18 contributes 0.5 to the total BRAM36 count.

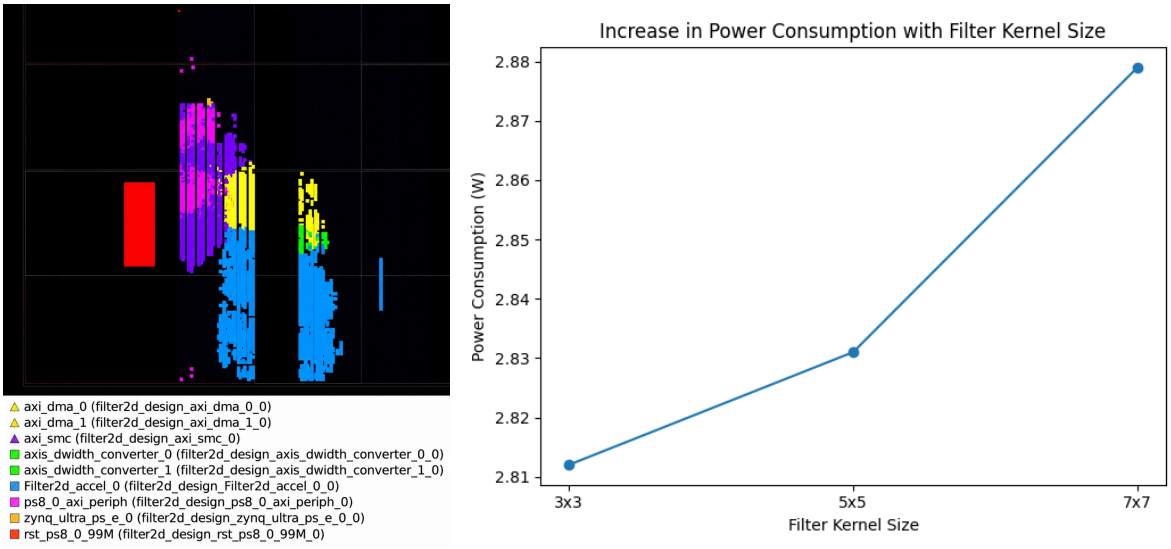
For the NPPC1 (one pixel per clock) configurations, logic utilization remains modest across all evaluated kernel sizes and color depths. LUT usage ranges from 8.18% to 10.99%, while FF utilization stays below 8.66%. In this design space, scaling trends are primarily driven by kernel size and color depth rather than throughput logic. Specifically, BRAM usage increases from 5.21% (7.5 units) for grayscale 3×3 to 18.75% (27.0 units) for RGB 7×7, reflecting the increased depth of line buffers and expanded window storage. Similarly, DSP utilization rises from 0.72% (9 units) to 11.78% (147 units) to accommodate the higher multiply–accumulate workload inherent to multi-channel kernels.

The transition to NPPC8 (eight pixels per clock) significantly alters the resource profile, particularly for high-complexity configurations. While the grayscale 3×3 setup remains efficient (9.76% LUTs), the RGB 7×7 configuration exerts substantial pressure on the device's arithmetic resources. In this case, BRAM utilization remains relatively manageable at 27.08% (39 units); however, DSP usage surges to 94.23% (1176 units). These metrics indicate that arithmetic density, rather than memory bandwidth or global clocking (where BUFG usage remains low at 3.98%), constitutes the primary limiting factor for high-throughput scaling within the Kria KV260 architecture. The DSP utilization further confirms the spatial replication introduced by NPPC8: the grayscale 3×3 configuration scales from 9 to 72 DSPs (8×), while RGB 7×7 scales from 147 to 1176 DSPs (8×), consistent with eight concurrently processed pixel lanes.

The `XF_NPPC1` and `XF_NPPC8` configurations were therefore selected as representative endpoints of the design-space trade-off: `XF_NPPC1` prioritizes low resource utilization, whereas `XF_NPPC8` prioritizes throughput through eight-lane pixel-level parallelism at the cost of substantially higher arithmetic resource consumption.

Figure 6 summarizes the implementation resource footprint and estimated power consumption of the proposed NPPC1 2D convolution accelerator on the Kria KV260. Figure 6(a) shows the post-implementation footprint of the NPPC1 7×7 RGB accelerator, confirming that the design remains within the available KV260 resources. Figure 6(b) shows that its estimated power for 4K RGB processing increases moderately from 2.812 W for 3×3 to 2.831 W for 5×5 and 2.879 W for 7×7, an overall increase of approximately 2.38%. The NPPC8 7×7

RGB configuration reaches 3.743 W because of its substantially higher DSP utilization and switching activity. These results indicate that further NPPC8 scaling is constrained primarily by arithmetic resources, particularly DSP slices, rather than BRAM capacity.



(a) (b)

Figure 6. NPPC1 accelerator on the Kria KV260; (a) resource footprint and placement for 7×7 RGB convolution; (b) estimated power consumption versus kernel size for 4K RGB images

Figure 7 compares CPU and XF NPPC1 end-to-end latency from 512×512 to 4K. Figures 7(a)–(g) correspond respectively to Emboss, Sobel vertical, Sobel horizontal, Prewitt vertical, Prewitt horizontal, Sharpen, and Strong Sharpen. For all filters, latency increases with resolution, while the PS–PL implementation provides an increasing performance advantage at higher resolutions.

Each data point in Figures 7(a)–(g) is obtained by averaging seven independent measurement batches, with ten consecutive executions performed within each batch. This protocol reduces timer-quantization effects and transient operating-system noise while enabling a reliable estimate of timing variability through the standard deviation. Across all experiments, the standard deviation does not exceed 340 μs, indicating stable and repeatable execution.

The common scaling behavior observed across Figures 7(a)–(g) is consistent with the streaming FPGA datapath: the processing cost grows approximately with the number of input pixels while the compute pipeline remains continuously utilized. Because the measured end-to-end FPGA latency also includes AXI DMA transfers between PS memory and the PL stream, the reported speedup reflects both computational acceleration and the communication overhead required to feed and drain the accelerator. Consequently, the relative benefit becomes more pronounced at higher resolutions, where fixed transfer and synchronization overheads are amortized over a larger number of pixels.

The XF NPPC8 configuration substantially improves performance through eight-pixel-per-cycle processing. For a 512×512 grayscale 3×3 Sobel vertical workload, NPPC8 achieves 1.01 ms $\pm$45.7 μs compared with 2.6 ms $\pm$36.5 μs on the CPU, corresponding to a 2.57× speedup. For the more demanding 4K RGB 7×7 workload, NPPC8 requires 58.6 ms $\pm$124 μs compared with 674 ms $\pm$340 μs on the CPU, yielding an 11.50× speedup. Under the same workload, NPPC1 requires 127 ms $\pm$44.3 μs; therefore, NPPC8 provides a further 2.17× speedup and reduces latency by approximately 53.9%. This improvement results from the wider streaming datapath and eight-lane arithmetic parallelism, while the low measurement variability indicates stable PS–PL execution.

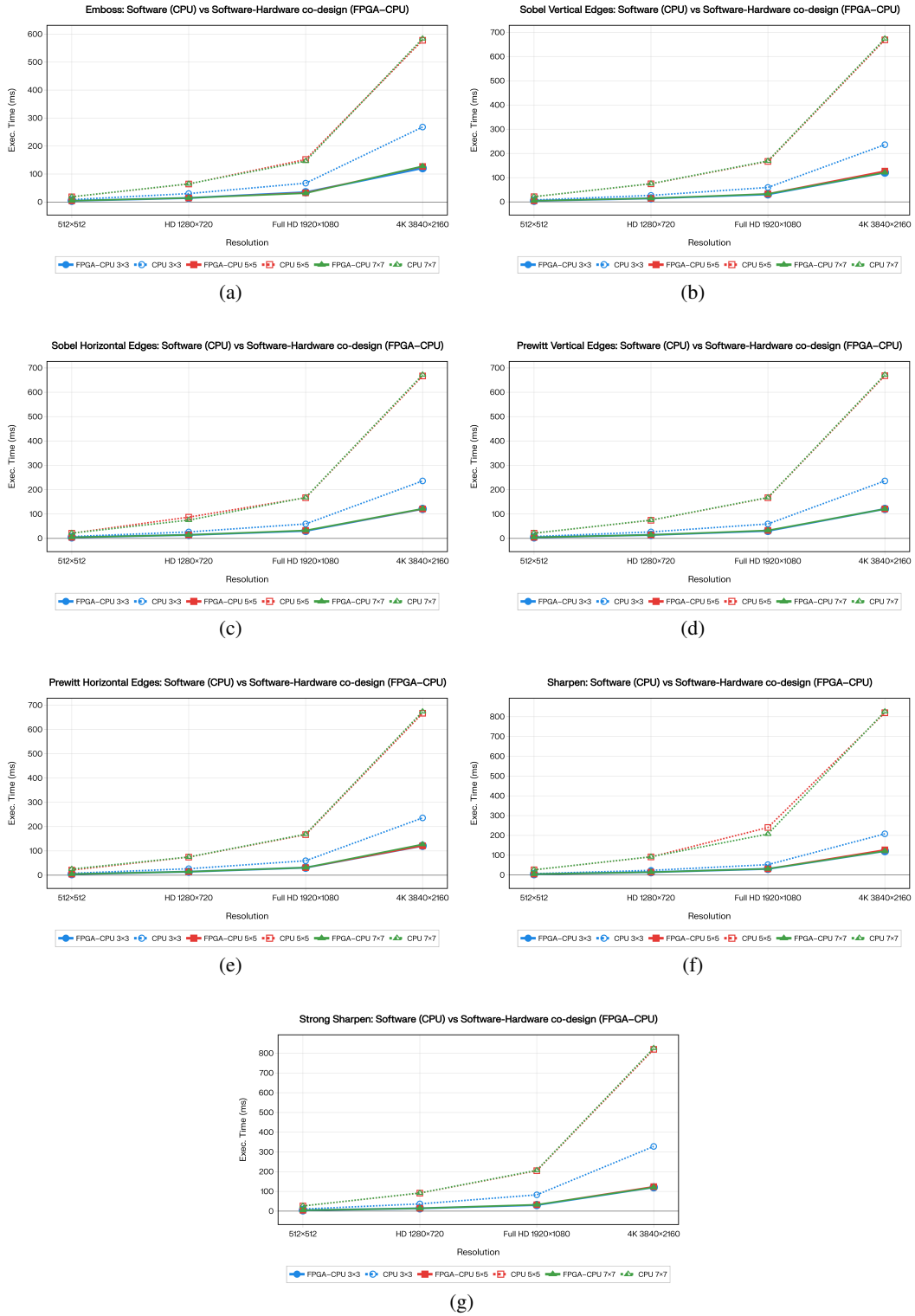
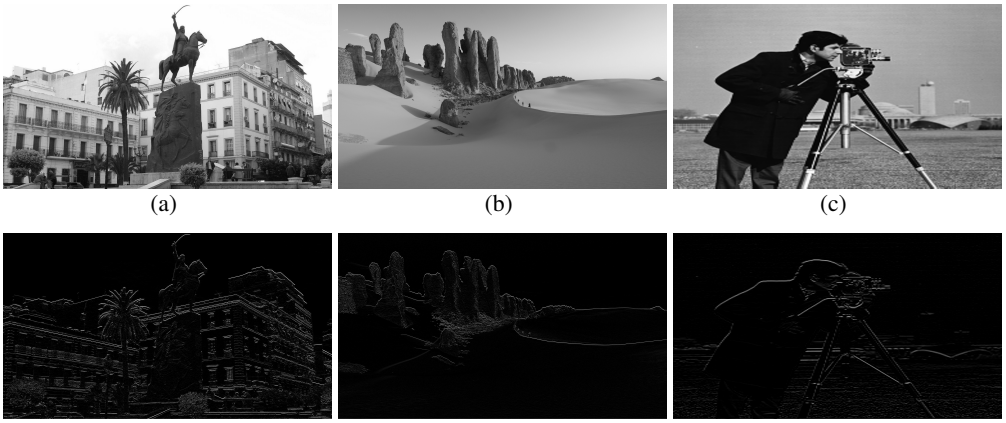


Figure 7. End-to-end execution-time comparison between the PS-only software implementation (CPU) and the `XF_NPPC1` PS–PL hardware–software co-design (FPGA–CPU) across the evaluated image resolutions and convolution filters; (a) emboss, (b) sobel vertical, (c) sobel horizontal, (d) prewitt vertical, (e) prewitt horizontal, (f) sharpen, and (g) strong sharpen

4.1. Application-level evaluation: 4K Sobel edge detection

To complement the kernel-level analysis, the accelerator was evaluated in an application-level Sobel vertical edge-detection task using three grayscale 4K (3840×2160) images with different scene characteristics: Emir Abdelkader (urban architecture), Sahara (natural landscape), and the Cameraman image (person/object), which was resized to 4K before processing. The same 3×3 Sobel vertical kernel was applied using the ARM Cortex-A53 OpenCV implementation and the `XF_NPPC1` and `XF_NPPC8` FPGA configurations. For each image and implementation, seven independent timing runs with 10 executions per run were performed. FPGA end-to-end latency includes image MM2S/S2MM transfers, filter-parameter MM2S transfer, accelerator invocation, and DMA completion synchronization, while image loading, grayscale conversion, overlay loading, and DMA-buffer allocation were excluded from the measured interval.

Figure 8 shows the three original 4K grayscale scenes in Figures 8(a)–(c) and the corresponding `XF_NPPC8` Sobel vertical outputs in Figures 8(d)–(f). The accelerator preserves the principal structural boundaries in the urban, natural, and person/object scenes, while homogeneous regions produce weaker responses. These results confirm that the same runtime-programmable accelerator can process substantially different image contents without hardware modification or re-synthesis.



(a) (b) (c)

(d) (e) (f)

Figure 8. Application-level evaluation of 4K grayscale Sobel vertical edge detection using the `XF_NPPC8` FPGA accelerator; (a) Emir Abdelkader, (b) Sahara, (c) Cameraman, (d) Emir Abdelkader Sobel-V, (e) Sahara Sobel-V, and (f) Cameraman Sobel-V

Table 4 shows consistent execution times across the three 4K scenes. The Cortex-A53 requires an average of 79.5 ms, while NPPC1 requires 85.0 ms (0.94×), indicating that the scalar accelerator does not amortize DMA and synchronization overhead for this lightweight 3×3 grayscale workload. In contrast, NPPC8 reduces average latency to 12.33 ms, corresponding to a 6.45× CPU speedup and 81.08 FPS. Its latency varies only from 12.3 to 12.4 ms across the three scenes, confirming that execution time is primarily determined by image dimensions and accelerator configuration rather than image content. The results support the use of the architecture for real-time high-resolution preprocessing in smart-camera and edge-AI systems; industrial inspection and medical-image preprocessing represent potential, but not experimentally validated, extensions.

Table 4. Application-level evaluation of 3×3 Sobel vertical edge detection on three 4K grayscale images

Image	CPU latency (ms)	NPPC1 latency (ms)	NPPC8 latency (ms)	NPPC1 speedup	NPPC8 speedup	NPPC8 FPS
Emir Abdelkader 4K	79.5±0.0232	85.0±0.0296	12.3±0.0631	0.94×	6.46×	81.30
Sahara 4K	79.6±0.0324	85.0±0.0174	12.4±0.0797	0.94×	6.42×	80.65
Cameraman (4K-resized)	79.4±0.0206	85.0±0.0136	12.3±0.0382	0.93×	6.46×	81.30
Average	79.5	85.0	12.33	0.94×	6.45×	81.08

Latency values are reported as mean±standard deviation over seven runs, with 10 executions per run. The average row reports the arithmetic mean of the per-image mean latencies.

Table 5 compares the proposed implementations with related FPGA-based convolution systems. The KV260 designs use more LUTs and FFs than [14], [17] because they include a complete PS–PL acceleration pipeline, while NPPC8 additionally widens the datapath for eight-pixel processing. DSP usage increases from 9 in NPPC1 to 72 in NPPC8 for the matched 512×512 grayscale 3×3 workload. NPPC8 reduces processing time from 3.25 ms for NPPC1 to 1.01 ms, a 3.22× improvement, and reaches approximately 990.1 FPS. The processing times reported by the related works are provided only as indicative comparisons because their resolutions, FPGA platforms, architectures, operating frequencies, and experimental conditions differ.

System-level power increases only from 2.804 W for NPPC1 to 2.851 W for NPPC8, corresponding to approximately 1.7% (0.047 W), despite the increase in parallelism. Computational throughput consequently rises from approximately 1.45 to 4.67 GOPS, while energy efficiency improves from 0.52 to 1.64 GOPS/W. The NPPC8 configuration therefore provides a superior throughput-per-watt trade-off while maintaining moderate system-level power consumption.

Table 5. Comparison with related FPGA-based convolution implementations

Parameter	This work (NPPC1)	This work (NPPC8)	[14]	[19]	[17]
FPGA platform	Kria KV260	Kria KV260	ZedBoard	Kintex-7	PYNQ-Z1
Image size	512×512	512×512	512×512	256×256	4K
Color space	Grayscale	Grayscale	Grayscale	Grayscale	Grayscale
Processing type	3×3 Sobel	3×3 Sobel	Convolution	Convolution	3×3 Sobel
Kernel size	3×3	3×3	3×3	–	3×3
Clock (MHz)	100	100	100	150	100
LUTRAM	1553	1770	1154	–	332
LUT	9585	11432	1701	–	4714
FF	13602	15398	270	–	6181
DSP48E	9	72	0	–	0
BRAM36	7.5	10.0	0.5	–	14
BUFG	1	1	1	–	1
Processing time (ms)	3.25	1.01	2.65	5.55	148
Power (W)	2.804	2.851	–	0.075	1.448

Note: "–" indicates not reported in the corresponding reference.

4.2. Limitations and future work

The present evaluation is limited to convolution kernels up to 7×7. Larger kernels were not experimentally evaluated. This limitation is particularly relevant to the `XF_NPPC8` architecture because the RGB 7×7 implementation already utilizes 1176 of the 1248 available DSP slices (94.23%) on the Kria KV260. Consequently, extending the same eight-pixel-parallel architecture to larger dense kernels would require additional architectural strategies, such as reduced pixel parallelism, time-multiplexed arithmetic, separable or decomposed convolution, or a device with greater DSP capacity. The energy-efficiency analysis is also limited by the absence of a directly comparable GPU implementation of the same convolution workload. The reported energy-efficiency values therefore characterize the evaluated FPGA configurations, while a fully controlled CPU–GPU–FPGA comparison using equivalent processing pipelines remains an important direction for further evaluation.

A further limitation is that the proposed architecture was experimentally validated only on the AMD Xilinx Kria KV260. Although Table 5 provides contextual comparisons with implementations reported on other FPGA platforms, these literature-based results do not constitute direct cross-platform validation. Consequently, portability, resource scaling, and performance consistency across different FPGA families remain to be experimentally established. This represents an important consideration for broader adoption of the proposed framework.

Future work will investigate the integration of the proposed streaming convolution accelerator as a reusable processing stage within complete CNN acceleration pipelines. Extension toward vision-transformer and hybrid CNN–transformer workloads will focus on convolutional preprocessing, patch-embedding or convolutional-stem operations, while attention-specific operators will require complementary hardware acceleration. Additional work will also investigate architectural techniques for supporting convolution kernels larger than 7×7 while maintaining a suitable balance between throughput and FPGA resource utilization.

5. CONCLUSION

This work introduced a scalable, high-performance hardware–software co-design for two-dimensional image convolution on the AMD Xilinx Kria KV260 Vision AI Starter Kit. The proposed architecture leverages AXI4-Stream interfaces and a dataflow-oriented streaming model to support both scalar (`XF_NPPC1`) and vector-parallel (`XF_NPPC8`) processing modes. By integrating on-chip line buffers and deep pipelining, the accelerator achieves steady-state throughputs of one and eight pixels per cycle, respectively, while maintaining a clear separation between control (AXI4-Lite) and data (AXI4-Stream+DMA) planes.

Experimental evaluation demonstrates the significant performance gains of the proposed parallel architecture. For a 4K RGB 7×7 Sobel vertical-edge workload, the `XF_NPPC1` configuration reduces end-to-end latency from 674 ms on the CPU to 127 ms, corresponding to a 5.31× speedup. Under the same workload, the `XF_NPPC8` configuration further reduces latency to 58.6 ms, providing an 11.50× acceleration over the software implementation and a 2.17× speedup over `XF_NPPC1`. For a 512×512 grayscale 3×3 Sobel workload, the NPPC8 implementation achieves approximately 990.1 FPS. The low timing variability observed throughout the experiments further demonstrates the stable and predictable execution behavior of the proposed PS–PL streaming pipeline.

The analysis of resource utilization and power efficiency reveals that the transition to the NPPC8 configuration significantly increases arithmetic parallelism, reaching 94.23% DSP occupancy for the RGB 7×7 configuration. However, memory utilization remains moderate, with BRAM usage reaching only 27.08%, indicating that the architecture efficiently manages line buffering even at high throughput. Despite the increased computational density, the NPPC8 RGB 7×7 configuration exhibits a post-implementation power estimate of 3.743 W. These findings indicate that the primary scaling bottleneck of the proposed architecture is the availability of arithmetic resources, particularly DSP slices, rather than BRAM utilization. Overall, the proposed architecture provides a robust, energy-efficient foundation for high-throughput embedded AI pipelines requiring predictable, high-performance convolution acceleration.

ACKNOWLEDGMENTS

We would like to express our sincere appreciation to Mario Ruiz (AMD University Program) and Jinger Zeng (Hackster.io) for their expert guidance and technical support. Their contributions via the PYNQ forums and the Hackster.io platform were instrumental to the success of this work. This research was not supported by any research grant or contract. The authors acknowledge in-kind hardware support in the form of a Kria KV260 Vision AI Starter Kit provided by Hackster.io.

FUNDING INFORMATION

This research received no external financial funding. The Kria KV260 Vision AI Starter Kit used in this study was provided by Hackster.io as in-kind hardware support.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Kamal Sehairi	✓	✓	✓	✓					✓				✓	
Ibtissam Mama Boucherit					✓		✓	✓	✓		✓			
Antonio Javier Gallego					✓	✓				✓	✓	✓		
Fatima Chouireb	✓					✓				✓		✓		

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal Analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject Administration
Fu : **Fu**nding Acquisition

CONFLICT OF INTEREST STATEMENT

The authors acknowledge receipt of hardware support (Kria KV260 Vision AI Starter Kit) from Hackster.io. Hackster.io had no involvement in the study design, data collection, analysis, interpretation, manuscript preparation, or the decision to publish the findings. The authors declare no other conflicts of interest.

DATA AVAILABILITY

The implementation artifacts used in this study are publicly available on Figshare under doi: 10.6084/m9.figshare.33454960. The dataset includes the Vitis HLS source files, Vivado design files, and the generated `.bit` and `.hwh` hardware overlay files for each evaluated implementation, together with the input images used in the experiments. The accompanying PYNQ Jupyter Notebook (`.ipynb`) used for DMA control, accelerator execution, and performance evaluation is also made available through the associated project repository.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," in *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017, doi: 10.1109/JPROC.2017.2761740.

[3] A. Dosovitskiy *et al.*, "An image is worth 16×16 words: Transformers for image recognition at scale," in *Proceedings of the International Conference on Learning Representations*, 2021.

[4] Z. Liu *et al.*, "Swin Transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp. 9992–10002, doi: 10.1109/ICCV48922.2021.00986.

[5] H. Hong *et al.*, "Survey of convolutional neural network accelerators on field-programmable gate array platforms: Architectures and optimization techniques," *Journal of Real-Time Image Processing*, vol. 21, no. 3, 2024, doi: 10.1007/s11554-024-01442-8.

[6] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019, doi: 10.1109/ACCESS.2018.2890150.

[7] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A survey of FPGA-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019, doi: 10.1145/3289185.

[8] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, no. 4, pp. 1109–1139, 2020, doi: 10.1007/s00521-018-3761-1.

[9] C. Pham-Quoc and T. N. Thinh, "Efficient FPGA-based convolutional neural network implementation for edge computing," *Journal of Advances in Information Technology*, vol. 14, no. 3, pp. 479–487, 2023, doi: 10.12720/jait.14.3.479-487.

[10] S. G. Aydin and H. S. Bilge, "FPGA implementation of image registration using accelerated CNN," *Sensors*, vol. 23, no. 14, 2023, doi: 10.3390/s23146590.

[11] S. Zhao, S. Gao, R. Wang, Y. Wang, F. Zhou, and N. Guo, "Acceleration and implementation of convolutional neural networks based on FPGA," *Digital Signal Processing*, vol. 141, 2023, doi: 10.1016/j.dsp.2023.104188.

[12] Z. Gao *et al.*, "Systematic reliability evaluation of FPGA implemented CNN accelerators," *IEEE Transactions on Device and Materials Reliability*, vol. 23, no. 1, pp. 116–126, 2023, doi: 10.1109/TDMR.2023.3235767.

[13] C. Sestito, S. Perri, and R. Stewart, "FPGA design of transposed convolutions for deep learning using high-level synthesis," *Journal of Signal Processing Systems*, vol. 95, no. 10, pp. 1245–1263, 2023, doi: 10.1007/s11265-023-01883-7.

[14] A. M. Moussa, R. Groves, and N. Rafla, "Hardware implementation of image processing morphological and convolution operations as SoC on FPGAs," in *DMIP '24: Proceedings of the 2024 7th International Conference on Digital Medicine and Image Processing*, 2025, pp. 72–77, doi: 10.1145/3705927.3705940.

[15] Y. Liu, Y. Ma, B. Zhang, L. Liu, J. Wang, and S. Tang, "Improving the computational efficiency and flexibility of FPGA-based CNN accelerator through loop optimization," *Microelectronics Journal*, vol. 147, 2024, doi: 10.1016/j.mejo.2024.106197.

[16] H. Shao, H. Shi, W. Mao, and Z. Wang, "An FPGA-based reconfigurable accelerator for convolution-transformer hybrid EfficientViT," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2024, pp. 1–5, doi: 10.1109/ISCAS58744.2024.10557992.

[17] K. Sehairi and T. Bouwmans, "Accelerating image processing functions on ZYNQ-FPGA using the Vitis Vision Library," in *2024 3rd International Conference on Advanced Electrical Engineering (ICAEE)*, 2024, pp. 1–6, doi: 10.1109/ICAEE61760.2024.10783257.

[18] J. Jiang, Y. Zhou, Y. Gong, H. Yuan, and S. Liu, "FPGA-based acceleration for convolutional neural networks: A comprehensive review," *arXiv Preprint*, 2025, doi: 10.48550/arXiv.2505.13461.

[19] K. N. Rao, D. R. V. A. S. Kumar, and B. Shankar, "FPGA-based accelerated image processing using parallel VLSI pipelines," *International Journal of Environmental Sciences*, vol. 11, no. 18s, pp. 1932–1937, 2025, doi: 10.64252/szb8ph26.

[20] S. Gdaim and A. Mtibaa, "Accelerating convolutional neural networks on FPGA platforms: A high-performance design methodology using OpenCL," *Journal of Real-Time Image Processing*, vol. 22, no. 2, 2025, doi: 10.1007/s11554-025-01647-5.

[21] AMD, "Vitis Vision Library Functions (API Reference), Version 2025.2," *Vitis Libraries*, Feb. 2026. [Online]. Available: https://docs.amd.com/r/en-US/Vitis_Libraries/vision/api-reference.html_1_32. (Accessed: Aug. 23, 2026).

[22] AMD Xilinx, "Vivado Design Suite User Guide: System-Level Design Entry (UG895)," ver. 2024.1, May 2024. [Online]. Available: https://docs.amd.com/r/en-US/ug895-vivado-system-level-design-entry/Working-with-the-Vivado-IDE. (Accessed: Aug. 23, 2026).

[23] AMD Xilinx, "Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994)," Nov. 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug994-vivado-ip-subsystems/Getting-Started-with-Vivado-IP-Integrator. (Accessed: Aug. 23, 2026).

[24] Advanced Micro Devices, Inc., "Kria K26 SOM Data Sheet (DS987)," ver. 1.5, Jan. 2024. [Online]. Available: https://docs.amd.com/r/en-US/ds987-k26-som/Overview. (Accessed: Aug. 23, 2026).
[25] Advanced Micro Devices, Inc., "Kria KV260 Vision AI Starter Kit," [Online]. Available: https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html#specifications. (Accessed: Jan. 12, 2026).

BIOGRAPHIES OF AUTHORS



Dr. Kamal Sehairi is an associate professor at École Normale Supérieure de Laghouat (ENSL), Algeria. He obtained his engineering degree in electronics in 2009 from the University of Laghouat Amar Telidji, followed by a magister degree in advanced signal processing in 2012 from École Militaire Polytechnique. He earned his Ph.D. in Electrical Engineering in 2017 from the University of Laghouat Amar Telidji. He is affiliated with the Laboratory of Telecommunications, Signals, and Systems at the University of Laghouat and the Laboratory of Applied Sciences and Didactics at ENSL. He has authored numerous peer-reviewed publications in international journals and conferences, with research contributions in background modeling, object detection, action recognition and FPGA/GPU-based real-time implementations. He is the developer of the UATL Threshold Library and the MATLAB Background Subtraction Library, holds one patent, and serves as a reviewer for several international journals. He can be contacted at email: k.sehairi@lagh-univ.dz.



Ibtissam Mama Boucherit received the B.Sc. degree in Automatics and the M.Sc. degree in Automatics and Industrial Computing from Amar Telidji University of Laghouat, Algeria. She is currently pursuing the Ph.D. degree in Automatics and Robotics at the same university. Her research interests include computer vision, intelligent visual systems, deep learning, and object detection techniques. Her current research focuses on the design and implementation of an intelligent visual system for fall detection, with emphasis on performance evaluation and cross-dataset analysis of deep learning-based object detection models. She can be contacted at email: ib.boucherit@lagh-univ.dz.



Antonio Javier Gallego is an associate professor in the Department of Software and Computing Systems at the University of Alicante (UA), Spain, and a member of the Pattern Recognition and Artificial Intelligence Research Group (gRFIA). He received his B.Sc. and M.Sc. degrees in Computer Science from the UA in 2004 and his Ph.D. in Advanced Computer Science from the same university in 2012. He has participated in 21 funded research projects (including one European), serving as principal investigator in several national and regional grants. He held an HPC-Europa3 (H2020) research fellowship and carried out a 12-month research stay at the University of Edinburgh (Informatics Forum). He has authored 115 publications, including 41 JCR journal articles, and his work has received more than 2,000 citations (h-index 24). He serves on the Board of Directors of AERFAI (Spanish Association for Pattern Recognition and Image Analysis) and is a member of the ValgrAI Joint Research Unit (UMI). His research interests include deep learning, pattern recognition, computer vision, and remote sensing. He can be contacted at email: jgallego@dlsi.ua.es.



Fatima Chouireb received her diploma in electrical engineering in 1992 and her magister and Ph.D. degrees from Saad Dahlab University of Blida, Algeria, in 1996 and 2007, respectively. In 1997, she joined the Department of Electrical Engineering, University of Laghouat, Algeria, as an assistant lecturer. Since 2015, she has been a full professor in the same department. She is also a team leader of the Signal, Image, and Speech Research Group of LTSS Laboratory, Laghouat University, Algeria. Her current research interests include signal, image, and speech processing, computer vision, localization/mapping/SLAM, and mobile robotics problems. She can be contacted at email: f.chouireb@lagh-univ.dz.