GStreamer-integrated HLS-based JPEG Encoder for Edge FPGA SoCs

[1]Yuri Guimaraes Pereira Primo da Silva, [1]Shinya Honda, [12]Sugako Otani,
[1]Masato Edahiro, [3]Abraham Monrroy Cano

[1]Nagoya University, [2]Renesas Electronics Corporation, [3]Map IV, Inc
yuri_gpps1@ertl.jp

Abstract— With the rise in ADAS (Advanced Driver Assistance System) and autonomous driving, there is a demand for lightweight image compression on edge devices. Although many image compression FPGA accelerators have been designed to provide high performance and low power consumption for data centers, they often do not meet the resource constraints of edge devices. This paper presents a hardware-accelerated JPEG encoder implemented using Vitis HLS on an AMD KV260 board for edge processing. We integrate the encoder with GStreamer, a multimedia framework that allows for easy integration with existing Linux tools. The system achieved the capture and encoding of 1080p NV12 images from a MIPI camera at 19.5 fps, demonstrating low variation when compared to CPU-based encoding, even under heavy load, and thus being suited for low-latency applications.

Keywords— Hardware Acceleration, JPEG Encoding, Vitis HLS, GStreamer, Kria KV260

I. Introduction

Edge computing applications, particularly in ADAS (Advanced Driver Assistance Systems) and real-time video processing, demand efficient image compression solutions that can operate under strict resource and latency constraints. CPU-based software implementations experience performance degradation under load and high resource consumption, while lower-level custom RTL (Register Transfer Language) hardware designs, though performant, lack flexibility for diverse deployment scenarios.

High-level Synthesis (HLS) tools offer a promising middle ground, enabling algorithm description at a higher abstraction level while automatically generating optimized hardware implementations. Modern HLS ecosystems like AMD's Vitis HLS [1] provide comprehensive toolchains for FPGA development, including specialized libraries for multimedia processing. However, existing HLS-based image encoders in the Vitis Libraries—such as PIK, Lepton, WebP, and JXL—target high-throughput datacenter applications with high resource requirements, limiting their applicability to resource-constrained edge devices.

Furthermore, practical deployment of FPGA accelerators in edge systems requires seamless integration with existing software frameworks. While solutions like the Vitis Video Analytics SDK (VVAS) provide multimedia support, integration gaps remain between hardware acceleration and standard Linux media pipelines for edge devices. This motivates the need for lightweight, edge-optimized image compression solutions with direct integration into established media frameworks like GStreamer.

To address these challenges, this paper presents a complete JPEG encoding solution optimized for edge computing applications. We selected JPEG due to its widespread adoption and computational simplicity. Our contributions include:

1. A resource-efficient JPEG encoder implemented in Vitis HLS, optimized for edge device deployment with minimal FPGA resource utilization while maintaining real-time performance.
2. A GStreamer plugin enabling integration with Linux multimedia pipelines for practical edge computing deployment.

Section II reviews related work in JPEG encoding and hardware implementation. Sections III, IV and V describe our JPEG encoder implementation and GStreamer integration. Section VI evaluates performance and Section VII concludes.

II. Related Work

Hardware approaches for image compression have explored various optimization strategies. Li et al. [2] implemented an HLS-based JPEG encoder for wearable devices, achieving 2.35 fps at 100MHz for 1280×960 images using dataflow optimizations. Current Vitis Libraries include encoders for PIK, Lepton, WebP, and JXL [3], but these target high-throughput datacenter applications with substantial resource requirements unsuitable for edge devices. As such, the existing HLS solutions either require extensive FPGA resources or achieve limited throughput for modern edge applications.

Additionally, practical FPGA deployment requires seamless software integration. The Xilinx Video Analytics SDK provides GStreamer plugins for FPGA acceleration [4], while Sanderson et al. [5] demonstrate integration in GStreamer pipelines on Zynq platforms. Despite these frameworks, gaps remain between hardware optimization and standard multimedia workflows.

Our work addresses these limitations by providing a resource-efficient HLS JPEG encoder with native GStreamer integration, enabling practical deployment in resource-constrained edge computing environments.

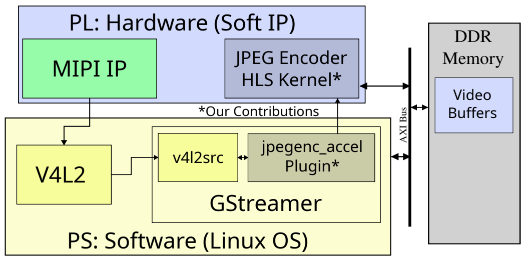

Fig. 1. **Architecture Overview**: Hardware-software architecture for JPEG encoding based on HLS and Linux. V4L2 interfaces with AMD's MIPI CSI-2 RX IP, obtaining DMA buffers passed to the JPEG encoder IP via GStreamer. This work contributes the HLS JPEG encoder kernel implementation and GStreamer integration.

III. Overview of Proposed Architecture

The major considerations for our design take into account the unique constraints of edge computing environments. Edge devices impose strict constraints on PL memory, necessitating a streaming architecture design. As some performance considerations, such as AXI bursts, require a level of input buffering, exploration of design space is crucial to optimize both resource utilization and throughput. At the same time, a rich software ecosystem allows for fast and practical deployment and testing of the system.

These requirements guided our design (Fig. 1), combining HLS hardware implementation with a Linux software stack. HLS enables efficient dataflow design and rapid design space exploration, while GStreamer boasts an extensive ecosystem with flexible I/O device support, which we leverage through a custom plugin interfacing with the HLS JPEG encoder.

IV. JPEG Encoder Kernel Implementation

JPEG encoding consists of color space conversion, DCT (Discrete Cosine Transform), quantization, entropy coding and encoding serializing. Please refer to the JPEG standard [6] for more details, as well as TooJpeg [7] for a short and insightful implementation in C++, of which this work uses as a basis for the HLS implementation.

We will not cover the color space conversion as we assume the input is in the NV12 (YUV420) format, a common format provided by cameras. We refer the reader to [3] for a detailed HLS implementation of various color space conversion functions.

We note that the overall algorithm lends itself to the use of a streaming architecture, as the data is processed in independent units called *MCUs* (Minimum Coded Units). In Vitis HLS, this is achieved by implementing each stage as a separate function and connecting them using `hls::stream` (FIFO streams). The result is a modular architecture with reusable code for processing different color spaces.

Fig. 2 shows an overview of the architecture. The Y and UV components are fed into the top kernel interface for aggregation into a stream of MCU blocks. After each MCU block is processed, it is sent to the next stage of the pipeline. DCT is

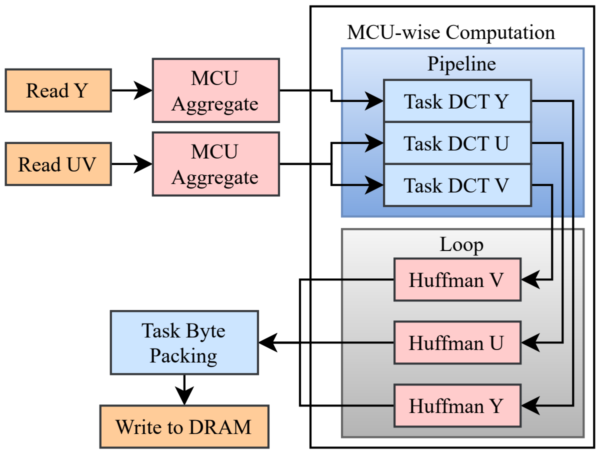

Fig. 2. **JPEG Kernel Dataflow**: An overview of the architecture of the JPEG encoding kernel, highlighting how data flows between different processing blocks.

applied to each block, and the information is compressed by quantization, generating a stream of coefficients that are then encoded by entropy. Finally, the final JPEG byte sequence is generated by packing the encoded bits together and then is written into memory. In particular, we use the HLS Task Library to instantiate the DCT and byte packing blocks.

A. *MCU Aggregation*

The input stream is a continuous stream of pixels received via an AXI4 Memory Mapped interface connecting to DDR memory. To process the data in MCU-sized units, we need to aggregate the input stream into blocks of 8×8 pixels. Using Block RAM (BRAM) allows us to implement internal memory to store these in programmable logic. Due to resource limitations of embedded FPGAs, however, we may not have enough BRAM to store the entire image in memory, or even a single MCU line at a time. At the same time, non-sequential access to memory is expensive in terms of performance, as the compiler is unable to synthesize it into burst transfers. We solve this by first writing in a burst fashion to non-expensive UltraRAM (URAM) blocks, and then rearranging the data into the MCU format using pipelined memory access.

In this solution, we showcase the effectiveness of HLS in adapting to changes in hardware resource requirements. While a Verilog solution may require extensive rewrite of the modules used, HLS achieves the same result by adding pragma directives, allowing one to specify the implementation type of the memory, as shown below:

```
uint8_t uram_y[WIDTH * 16];
uint8_t uram_uv[WIDTH * 8];
#pragma HLS BIND_STORAGE variable=uram_y type=
    ram_2p impl=uram
#pragma HLS BIND_STORAGE variable=uram_uv type=
    ram_2p impl=uram
```

B. *Discrete Cosine Transform and Quantization*

The 2D DCT is implemented using a separable approach, in which the 1D DCT is applied to each row and then to each column. The module takes as input an 8×8 block of pixels from a single channel (Y, Cb, or Cr) and outputs an 8×8 block of DCT coefficients.

By leveraging HLS's `array_partition` directive, we divide the memory accesses into chunks that can be processed concurrently. This enables 1D DCT to be performed concurrently for each row or column, optimizing memory access patterns and achieving a pipelined architecture. The 1D DCT itself is implemented using a traditional "DCT-II" convolutional approach [8].

We combine quantization with the DCT step, multiplying the DCT coefficients element-wise by the quantization matrix. By applying a scaling factor to the matrix, we avoid the usual division operation, such that the final operation is reduced to a multiplication and shift operation (Eqs. 1 and 2),

$$Q'_{i,j} = \frac{2^S}{Q_{i,j}} \tag{1}$$

$$\mathrm{DCT}_{i,j} = \mathtt{SRR}(\lfloor \mathrm{DCT}_{i,j} Q'_{i,j} \rfloor, S) \tag{2}$$

where $Q_{i,j}$ is the original quantization matrix, S is the scaling factor, and $\mathtt{SRR}(x, S)$ is the rounded arithmetic shift right operation, implemented in C++ as

```
(x + (1 « (S - 1))) » S
```

C. *Entropy Coding and Serialization*

The final stage performs Huffman encoding and serialization of quantized DCT coefficients, as shown in Fig. 3. DC coefficients use differential encoding with the previous block's DC value, while AC coefficients are run-length encoded in zig-zag order. The Huffman encoder outputs variable-length codes (`huff_code`), their bit lengths (`huff_size`), and count (`huff_num`) via `hls::stream` interfaces.

The serialization module packs variable-length Huffman codes into byte-aligned output using a stateful bit accumulator that tracks current byte and bit positions. When processing completes, an end-of-image marker (`0xFFD9`) is appended to generate a valid JPEG stream.

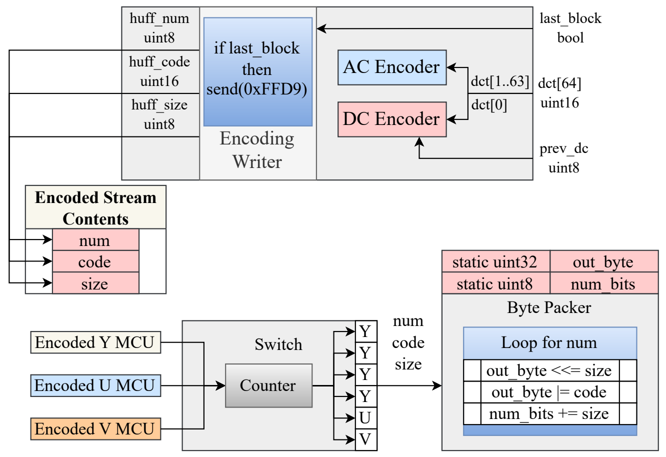

Fig. 3. **Huffman Coding and Serialization**: The Huffman encoder receives DCT coefficients and performs differential and RLE-based encoding, while the byte packer accumulates completed MCUs and serializes the resulting codes according to the JPEG stream format.

V. Software Architecture

The software architecture for the JPEG encoder bridges the gap between hardware and software components, allowing for the encoder to be used in a variety of applications. We utilize AMD's Xilinx Runtime Library to interface with the hardware accelerator, as well as GStreamer for interaction with other media components. GStreamer, a media framework, is a useful candidate for a common media interface. Along with Video4Linux2, it provides increased flexibility and compatibility with different media formats. Fig. 4 presents an overview of the GStreamer plugin architecture for interacting with the HLS-defined JPEG encoder.

Most notably, GStreamer provides a plugin API for interfacing with arbitrary image processing functions. We provide `jpegenc_accel`, a plugin akin to GStreamer's `jpegenc` plugin. The plugin is based on the Vitis Video Analytics SDK (VVAS) utils API, which provides a set of utility functions for handling DMA transactions specifically for XRT kernels via `vvas_xrt_import_bo`. Using this approach, we can efficiently manage DMA transactions and ensure optimal performance, enabling a path for direct camera encoding through V4L2 DMA buffers.

VI. Experiments and Results

We synthesized and implemented the design of the JPEG encoder for the AMD Kria KV260 board[9], which is equipped with a Xilinx Zynq UltraScale+ MPSoC. Table I contains the specifications of the board.

For the rest of the section, we provide an evaluation based on the **resource utilization** of the encoding IP compared to the total available for the Kria KV260, as well as a **comparative evaluation** between our encoder and the libjpegturbo-based `jpegenc` encoder available on GStreamer.

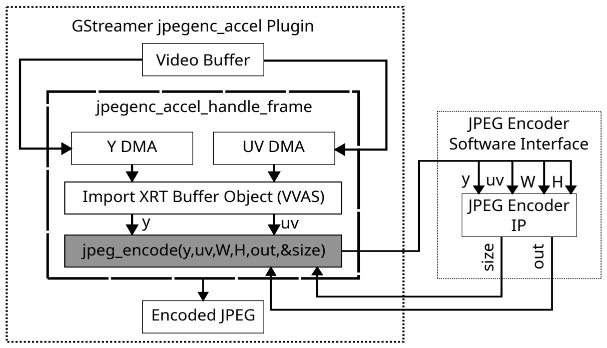
GStreamer jpegenc_accel Plugin
Video Buffer
jpegenc_accel_handle_frame
Y DMA
UV DMA
Import XRT Buffer Object (VVAS)
y
uv
jpeg_encode(y,uv,W,H,out,&size)
Encoded JPEG
JPEG Encoder Software Interface
y
uv
W
H
JPEG Encoder IP
size
out

Fig. 4. **Overview of the GStreamer Plugin**: After obtaining the video buffer from a source element, we extract the DMA file descriptors corresponding to the Y and UV planes, using them as input for the JPEG encoder. The encoder itself is executed via the Xilinx Runtime Library.

TABLE I
Specifications of the AMD Kria KV260 board

Parameter	**Value**
CPU	Cortex-A53 up to 1.5GHz
Memory	4GB 64-bit DDR4

A. *Resource Utilization*

We used Vitis 2023.2 to synthesize and implement the JPEG encoder for the AMD Kria KV260 board at 250 MHz. The scaling factor was set to 10, and the quantization matrix was set to the standard JPEG quantization matrix. We compared resource utilization and megapixel throughput with the WebP encoder from Vitis Libraries [3], implementing it at 100 MHz since the default 200 MHz configuration exceeded available resources. We ran the JPEG encoder on the Kria KV260 board using a GStreamer pipeline that reads from raw 1920×1080 NV12 video frames. For the WebP encoder, we used the `cwebp` test program available in the Vitis Libraries.

Table II shows the JPEG encoder utilizes 21% of available URAM and less than 14% of other resources on the Kria KV260 board. Compared to the WebP encoder, the JPEG encoder operates at higher frequency (250 MHz vs 100 MHz) with lower resource utilization across all categories. The WebP encoder contains 5856 lines of HLS code compared to 1253 lines for our JPEG encoder. This reduced complexity enables lower resource utilization suitable for edge device deployment.

B. *Comparative Evaluation*

We made a baseline comparison of GStreamer pipelines using either the libturbojpeg-based `jpegenc` plugin or our own implementation. Three different pipelines were used for the evaluation:

- `cam`: Provides images from a live camera feed. It uses the `v4l2src` source element.
- `file`: Provides images from a dataset captured from the camera. It uses the `multifilesrc` source element.

The camera used was an onsemi AR1335 producing 1920×1080 NV12 images at a maximum frame rate of 30 fps.

B.1. Overall Performance

We evaluated performance in terms of throughput (fps), PSNR (Peak Signal-to-Noise Ratio), compression ratio, and CPU usage. Throughput was measured using the `gst-perf` utility [10], while CPU usage was monitored via `htop`. For PSNR calculation, we converted both original and encoded images to luminance using ITU-R BT.601 [11] weights (Y = 0.299R + 0.587G + 0.114B) and calculated:

$$PSNR = 20 \log_{10}(255) - 10 \log_{10}(MSE)$$

where MSE is the mean squared error between luminance values. Compression ratio represents the percentage of original file size after encoding. The results are shown in Table III.

TABLE II
Resource Utilization Comparison of JPEG and WebP Encoders on KV260

	JPEG (our work)	WebP [3]	Total Available
Frequency	250 MHz	100 MHz	
Throughput	4.49 MPps	1.88 MPps	
LUT	**11,513**	62,296	92,832
LUTAsMem	**2,355**	6,610	55,336
REG	**18,150**	71,227	191,197
BRAM	**6**	81	106
URAM	**12**	46	56
DSP	**85**	834	1248

Results show performance differences between FPGA and CPU implementations across test scenarios. For camera input, the FPGA achieves 19.5 fps compared to 2.6 fps for CPU-based encoding—a 6.5× throughput difference. With file input, both implementations perform similarly (21.7 vs 22.6 fps), while the FPGA maintains lower CPU usage (14% vs 107%). Image quality is comparable with PSNR values of 27.45 dB (FPGA) and 27.72 dB (CPU), while compression efficiency is similar (3.219% vs 3.202%). The FPGA reduces CPU usage from 93% to 5% during camera processing, enabling workload offloading for edge applications.

B.2. CPU Stress Testing on Single Core

We also investigated the throughput performance of the FPGA and CPU implementations under resource-constrained environments. The experiment consisted of pinning the GStreamer pipeline to a single core via the Linux `taskset` utility and gradually increasing the CPU load stress via `stress-ng`. To minimize interferences from the scheduler, we used the `nice` utility to set the priority of the `stress-ng` process to -10.

Fig. 5 shows the throughput slowdown per CPU stress level for the FPGA and CPU implementations under a single-core environment. From the figure, we can see that the FPGA implementation is more robust to CPU stress than the CPU

TABLE III
Performance Evaluation of the JPEG Encoder

		CPU	FPGA (Our)
`cam`	Throughput (fps)	2.6	**19.5**
	CPU Usage (%)	93	**5**
`file`	Throughput (fps)	**22.6**	21.7
	CPU Usage (%)	107	**14**
	Compression Ratio	3.202%	**3.219**%
	PSNR (dB)	**27.72**	27.45

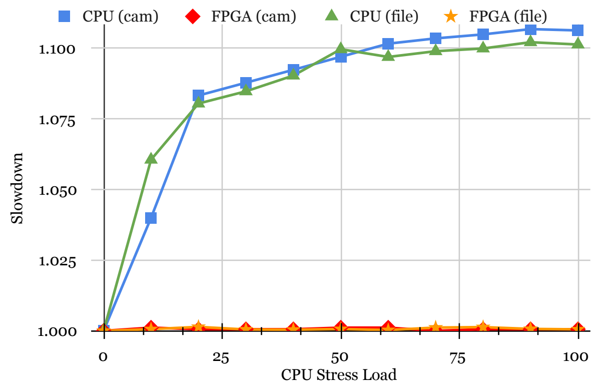

Fig. 5. **Slowdown During CPU Stress**: The figure shows the relative slowdown of the FPGA and CPU implementations under a single-core environment and at various CPU stress levels.

implementation. The FPGA implementation shows a consistent throughput performance even when the CPU is under heavy load, while the CPU implementation shows a significant decrease in throughput as the CPU stress level increases, peaking at around 10% slowdown for 50% CPU stress load.

B.3. Memory Stress Testing

While the CPU stress testing did not affect the FPGA implementation greatly, memory stress testing should show some variations since the images must pass through the AXI bus, which can be overloaded by stress. For the experiment, we established two configurations for each pipeline:

1. **Single-Core**: The FPGA implementation with a single core (CPU3 in this case).
2. **Multi-Core**: The FPGA implementation with multiple cores (CPU0 through CPU3, denoted as CPU*).

We used `stress-ng` with the options `-vm-bytes 100M -vm-method modulo-x` to simulate memory stress. Fig. 6 illustrates the configuration of the stress testing.

Fig. 7 shows the throughput slowdown during memory stress testing, each subfigure representing one of the pipelines. The V4L2 source, shown at the top, presents the best performance, with the DMA-enabled FPGA encoder reaching at worst 10.5%

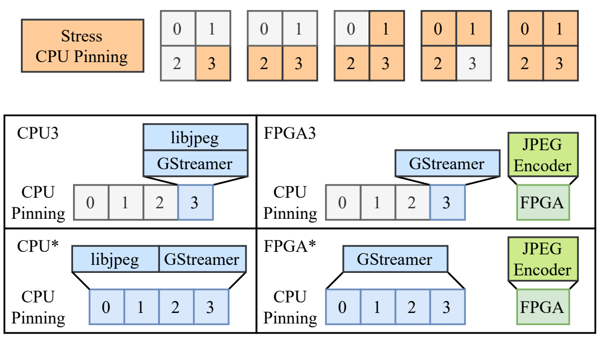

Fig. 6. **Memory Stress Configuration**: Four configurations for single-core or multi-core affinity of the pipeline (CPU3, CPU*, FPGA3, FPGA*), as well as five CPU affinity configurations for the memory stress.

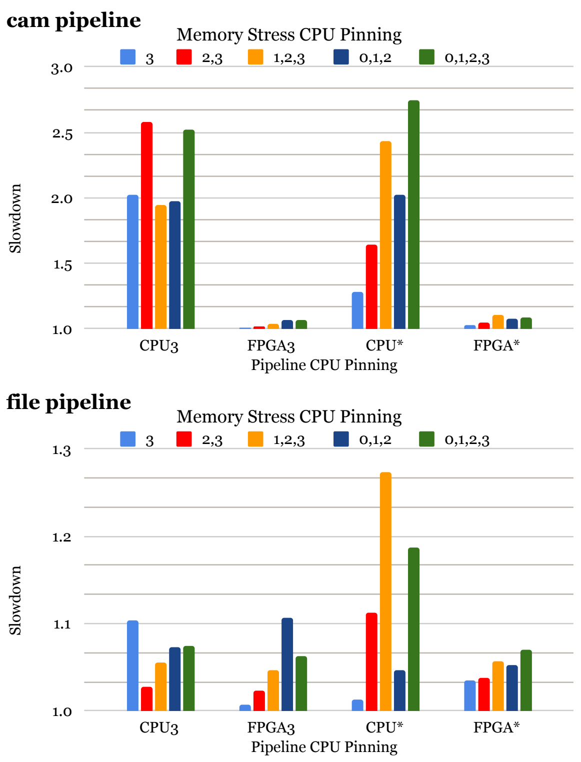

Fig. 7. **Slowdown During Memory Stress**: The figure shows the performance slowdown of FPGA and CPU implementations at various memory stress afinity configurations across each pipeline. $y = 1.0$ represents the baseline performance of each configuration (CPU3, CPU*, FPGA3, FPGA*) with no memory stress applied.

slowdown, while the CPU implementation reaches at worst as much as 175% slowdown. The file source from system memory, shown at the bottom, presents a more stable performance, with the FPGA encoder reaching at worst 10.6% slowdown, while the CPU implementation reaches at worst as much as 27.3% slowdown.

C. *Implications of Stress on FPGA and CPU Performance*

The CPU and FPGA implementations exhibit different behavior under both compute (CPU) and memory stress, from which we draw the following conclusions:

(1) Compute stress is remedied by FPGA offloading
Fig. 5 shows that the GStreamer pipeline management, which includes data acquisition from source elements and scheduling into downstream elements, minimally affects the total throughput, even when under compute stress. As such, offloading of the heavy operations is enough to overcome throughput degradation. CPU encoding, which is preemptible, has a reduced throughput under compute stress. On multicore systems, this might be mitigated by isolating a CPU core so as to only dedicate it to the encoding task. However, such a solution loses viability on more constrained systems, or systems with heavy critical tasks.

(2) Non-cacheable memory hugely impacts CPU pipelines
DMA-based sources (the `cam` pipeline) more deeply affect the responsiveness of the CPU encoding implementation. We attribute this mainly to the non-cacheable nature of DMA addressing on the CPU. A possible countermeasure would be to

copy the contents of the DMA buffer onto cacheable memory before processing it.

(3) Multicore systems are more predictable with FPGA offloading
When using input cacheable from user space (`file` pipeline), the behavior of both CPU and FPGA solutions are similar when pinned to a single core. However, when unpinned, the CPU-based encoding pipeline shows higher instability. We attribute this to two reasons: Firstly, when unpinned, the encoding portion of the pipeline can execute on a separate thread at a higher core usage, thus incurring more memory accesses per second, which makes it more susceptible to memory congestion. Secondly, the asymmetric nature of the CPU encoder workload across multiple cores causes the effects of memory stress to be less predictable.

On the other hand, the FPGA pipeline demonstrates predictable behavior when unpinned. Under the same reasoning as above, the CPU workload involved in the FPGA pipeline is light when compared to encoding, and as such, demonstrates symmetrical behavior across all threads.

VII. Summary and Conclusion

In this paper, we presented a Vitis HLS-based JPEG encoder specifically designed for edge computing applications on FPGAs. Our work addresses the growing demand for real-time, resource-efficient image compression in embedded systems, where traditional CPU-based solutions suffer from performance degradation under load and RTL designs lack flexibility.

Our main contributions include: (1) a JPEG encoder implemented with Vitis HLS optimized for edge device resource constraints, and (2) a GStreamer plugin for integration with Linux multimedia pipelines. The encoder implements the full JPEG compression pipeline including MCU aggregation, DCT with quantization, and entropy coding with serialization.

Experimental results on the AMD Kria KV260 board show the FPGA implementation achieved 19.5 fps for camera input and 21.7 fps for file input, while reducing CPU usage to 5% compared to 93% for the CPU-based libjpeg implementation. The encoder maintains image quality with a PSNR of 27.45 dB and compression ratio of 3.2%. Resource utilization analysis shows the design uses 21% of available URAM and less than 14% of other FPGA resources. The remaining space thus allows for other circuits to be implemented, including multiple encoders for parallel stream processing of multiple cameras.

Stress testing results indicate stable FPGA performance under CPU stress conditions, while CPU-based encoding experienced performance degradation. Memory stress testing showed 10.5% slowdown in the worst case for the FPGA implementation, compared to up to 175% slowdown for CPU implementations.

Integration with GStreamer enables ease of deployment in various scenarios, including ADAS and multimedia processing applications requiring consistent performance.

Current limitations of the architecture include fixed quantization tables and single-stream processing. The encoder is optimized for 1080p resolution; higher resolutions may require architectural modifications. Future work could explore adaptive quantization techniques to improve compression efficiency, implementation of additional image formats, and optimization for higher resolution inputs. Additionally, investigating multi-channel processing capabilities could further enhance throughput for applications requiring parallel image streams.

Some of the evaluations in this work are inconclusive. An in-depth comparison with other hardware encoders was not performed, but could provide insights into the relative strengths and weaknesses of our approach. The GStreamer pipelines had a high scheduling priority during our evaluation, but in a real-world scenario critical processes may compete for resources. In future work, we plan to explore the impact of these factors on the encoder's performance and consider implementing adaptive strategies to mitigate their effects.

References

[1] AMD. "Vitis HLS". `https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html`. Accessed 2025-05-30.

[2] Yuecheng Li et al. "A FPGA implementation of JPEG baseline encoder for wearable devices". In: *2015 41st Annual Northeast Biomedical Engineering Conference (NEBEC)*. IEEE, Apr. 2015, pp. 1–2.

[3] AMD. "Vitis Libraries". `https://github.com/Xilinx/Vitis_Libraries`. Accessed 2025-05-30.

[4] AMD. "Vitis Video Analytics SDK". `https://www.amd.com/en/developer/resources/vitis-video-analytics-sdk.html`. Accessed 2025-05-30.

[5] Jonathan Sanderson et al. "Integrating Gstreamer with Xilinx's ZCU 104 Edge Platform for Real-Time Intelligent Image Enhancement". In: *2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, Aug. 2023, pp. 639–643.

[6] Gregory K. Wallace. "The JPEG still picture compression standard". In: *Commun. ACM* 34.4 (Apr. 1991), pp. 30–44.

[7] Stephan Brumme. "A JPEG encoder in a single C++ file". `https://github.com/stbrumme/toojpeg`. Accessed 2025-05-30.

[8] N. Ahmed, T. Natarajan, and K.R. Rao. "Discrete Cosine Transform". In: *IEEE Transactions on Computers* C-23.1 (1974), pp. 90–93.

[9] AMD. "Kria KV260 Vision AI Starter Kit". `https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html`. Accessed 2025-08-20.

[10] RidgeRun. "gst-perf". `https://github.com/RidgeRun/gst-perf`. Accessed 2025-05-30.

[11] ITU-R. "Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios". `https://www.itu.int/rec/R-REC-BT.601`. ITU-R Recommendation BT.601.