IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 4 April 2024, accepted 28 April 2024, date of publication 6 May 2024, date of current version 13 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3397536

RESEARCH ARTICLE

Energy-Efficient Implementation of YOLOv8, Instance Segmentation, and Pose Detection on RISC-V SoC

HANSEN WANG, DONGJU LI, (Member, IEEE), AND TSUYOSHI ISSHIKI, (Member, IEEE)
Department Information and Communications Engineering, Tokyo Institute of Technology, Tokyo 152-8550, Japan

Corresponding author: Hansen Wang (wang.h.aj@m.titech.ac.jp)

This work was supported by the New Energy and Industrial Technology Development Organization (NEDO) under Grant JPNP23015.

ABSTRACT This article introduces a programmable Artificial Intelligence Visual System on Chip (AIV-SoC), optimized for YOLOv8 end-to-end execution. It operates on input image resolution of 352 × 352 to perform object detection, instance segmentation, and pose detection, achieving notable Frames per Second (FPS) of 67.1, 55.2, and 64.9, respectively. The AIV-SoC is founded on the RISC-V architecture and meticulously crafted utilizing the LLVM-C2RTL toolkit. Neural network layer scheme optimization, piecewise linear function (PLF) approximation, multi-cycle path, 8-bit fix-point data per-group quantization and 4-pattern transposed convolution are introduced to enhance its versatility and power efficiency. Notably, the implemented model exhibits superior power efficiency in comparison to other GPU and FPGA platforms. This characteristic envisions practical applications across industries. The presented AIV-SoC platform, with its efficiency, opens up possibilities for deployment in resource-constrained environments such as healthcare, agriculture, and surveillance. This potential redefines the landscape of object detection, instance segmentation, and pose detection in real-world scenarios, providing insight into a future where these technologies play a pivotal role in reshaping how we interact with machines and interpret visual information.

INDEX TERMS Artificial intelligence visual system, application-specific integrated circuit, hardware/software co-design, instance segmentation, object detection, pose detection, RISC-V, YOLOv8.

I. INTRODUCTION

Object detection, a pivotal computer vision technology, is dedicated to discerning instances of specific semantic objects within digital images and videos [1]. Its versatile applications span a spectrum of computer vision tasks, encompassing image annotation and retrieval, vehicle counting, activity recognition, face detection and video object co-segmentation. Moreover, object detection plays a pivotal role in autonomous driving systems [2].

In conjunction with object detection, instance segmentation represents a nuanced extension of this field. Unlike object detection, which localizes objects with bounding boxes, instance segmentation delves further into the delineation of object boundaries at the pixel level. In comparison to semantic segmentation, the task of instance segmentation is notably more fine-grained, intricate, and demanding [3]. Noteworthy methodologies, such as Mask R-CNN [4], initially perform object detection and subsequently assign a binary label to each pixel within the bounding box. The work presented in [5] seamlessly integrates with cutting-edge one-stage detection frameworks, surpassing Mask R-CNN under the same training schedule and achieving faster processing times.

The synergistic integration of object detection and instance segmentation has found applications in various domains, including scene understanding for robotics [6], medical image analysis for precise organ delineation [7], [8], video surveillance, augmented reality [9], and image compression. This amalgamation enhances the granularity of information extraction, proving particularly valuable in scenarios where object boundaries and spatial relationships play a pivotal role.

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

Pose detection, another key task in computer vision, involves estimating the spatial arrangement of key body joints or keypoints within an image or video. This critical field plays a pivotal role in diverse applications such as human-computer interaction, surveillance, and augmented reality. The extracted pose information is instrumental in tasks ranging from anomaly detection in surveillance systems to enhancing virtual try-on experiences in e-commerce [10], [11]. Additionally, pose detection contributes significantly to sports analytics, facilitating precise motion tracking for athletes and aiding in skill refinement and training optimization [12].

In this paper, we introduce the design of the Artificial Intelligence Visual System on Chip (AIV-SoC) with a focus on Int8 precision and a 352×352 input resolution. It introduces an innovative and versatile circuit designed for new computationally intensive layers in YOLOv8, including SiLU and softmax. Through software programming, the AIV-SoC exhibits proficiency in executing the entire end-to-end process of object detection, instance segmentation, and pose detection tasks.

The structure of the paper is as follows: Section II provides an overview of relevant works. Section III delves into a comprehensive examination of the hardware design architecture. Section IV discusses the specifics of the software design. Section V presents the simulation results and their analysis. Finally, in Section VI, we summarize the strengths and weaknesses of our AIV-SoC.

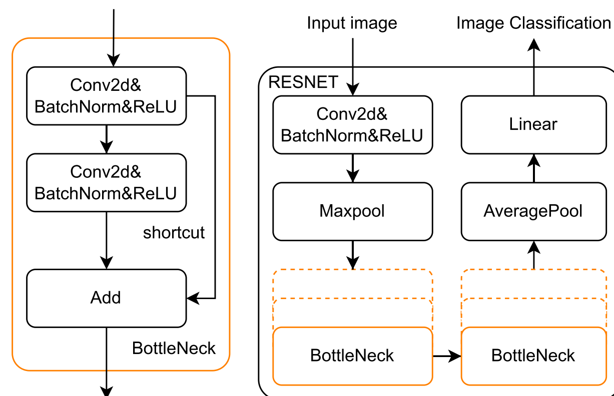


FIGURE 1. Structure of ResNet and Bottleneck.

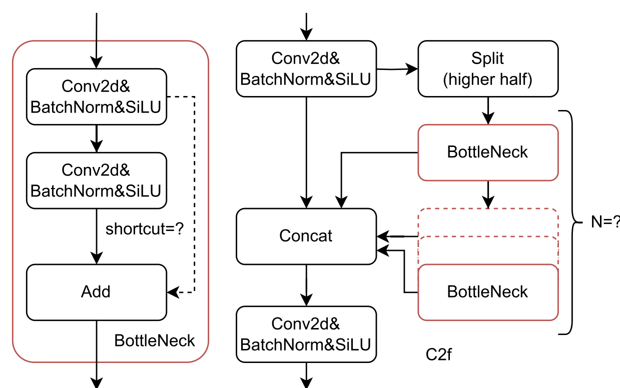


FIGURE 2. Structure of C2f and Bottleneck.

II. RELATED WORK

A. YOU ONLY LOOK ONCE MODEL

YOLO, an acronym for "You Only Look Once," stands out as a real-time object detection algorithm that partitions the input image into a grid. It predicts bounding boxes and class probabilities directly within each grid cell [13]. YOLOv8, the latest release in the YOLO series by Ultralytics [14], stands out for its ability to achieve a balance between accuracy and real-time inference speed, distinguishing itself from earlier YOLO versions and alternative object detection architectures. YOLOv8 exhibits competitive performance when evaluated against other state-of-the-art models such as EfficientDet and Cascade R-CNN.

YOLOv8 adopts a modified CSPDarknet53 architecture by replacing C3 (CSP Bottleneck with 3 convolutions) with C2f (Faster CSP Bottleneck with 3 convolutions). BottleNet architectures leverage residual connections, inspired by ResNet structures as shown in Figure 1, to facilitate the flow of information through the network. These connections enable the network to efficiently learn and propagate relevant information while mitigating the risk of vanishing or exploding gradients during training.

In CSPDarknet53, C3 functions as a module for constructing feature extraction layers [15]. The C2f structure addresses the challenge of insufficiently fused defect information. Illustrated in Figure 2, the C2f structure divides the input feature map into two segments, processes them through two branches, and subsequently merges the branch outcomes. This amplifies feature representation, facilitating information flow across diverse branches and strengthening the fusion effect of bearing defect features. Some C2f structures include shortcuts in the Bottleneck, while others do not. The 2D convolution employs batch normalization and utilizes the Sigmoid Linear Unit (SiLU) activation function to introduce non-linearity.The variable N is dependent on model sizes, such as YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra-large).

The YOLOv8 architecture incorporates the Spatial Pyramid Pooling Fast (SPPF) from YOLOv5, as introduced by Glenn Jocher [17]. As illustrated in Figure 3, the original Spatial Pyramid Pooling (SPP) structure utilizes pooling kernels of 5×5, 9×9, and 13×13 to extract features between first convolution and concatenation layer [16], preserving spatial information while reducing the image size. In contrast, SPPF performs three consecutive pooling layers with 5×5 kernels before concatenation, significantly enhancing computational efficiency compared to traditional SPP structures.

YOLOv8 employs classification and regression branches for its loss computation [21]. The classification branch utilizes Binary Cross-Entropy (BCE) loss, while the regression

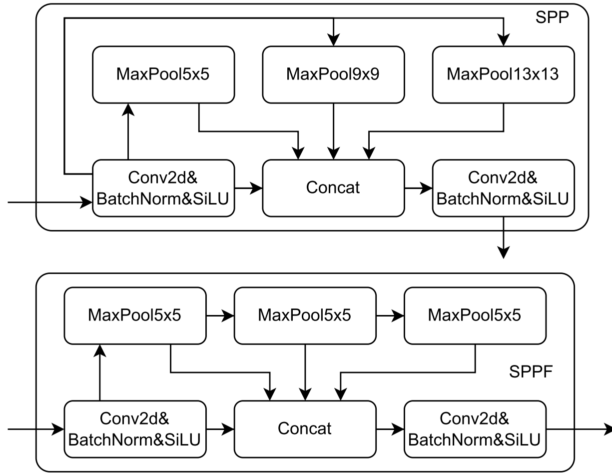


FIGURE 3. Comparism of SPP and SPPF.

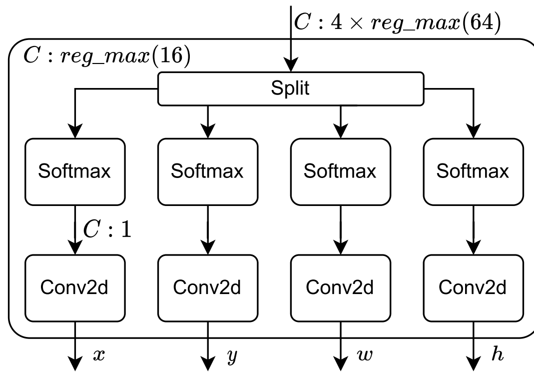


FIGURE 4. Structure of integral DFL.

branch in this paper employs the Distribution Focal Loss (DFL) [22] and Complete IoU (CIoU) loss functions. In the regression branch, YOLOv8 utilizes the integral module proposed in [23], illustrated in Figure 4, where C represents the channel, and the default value for *reg_max* is set to 16. The output of DFL comprises only 4 channels, representing the box's position (x, y) and dimensions (w, h) before being located in the grid cell.

YOLOv8 supports dynamic input resizing with image dimensions represented as $S_x \times S_y$, where both S_x and S_y must be integer multiples of 32. This improves adaptability to various object scales during inference.

The PANet, or Path Aggregation Network [18], functions as the core backbone and head network for YOLOv8, as illustrated in Figure 5. PANet represents a significant advancement in instance segmentation, introducing a unique path aggregation mechanism and contextual awareness module. The Feature Pyramid Network (FPN) comprises levels P_1 to P_5, where the feature map size in layer P_i is $(S_x/2^i, S_y/2^i)$. By augmenting the FPN, PANet effectively tackles the challenges of multi-scale object recognition, particularly excelling in improving object boundaries and capturing intricate details.

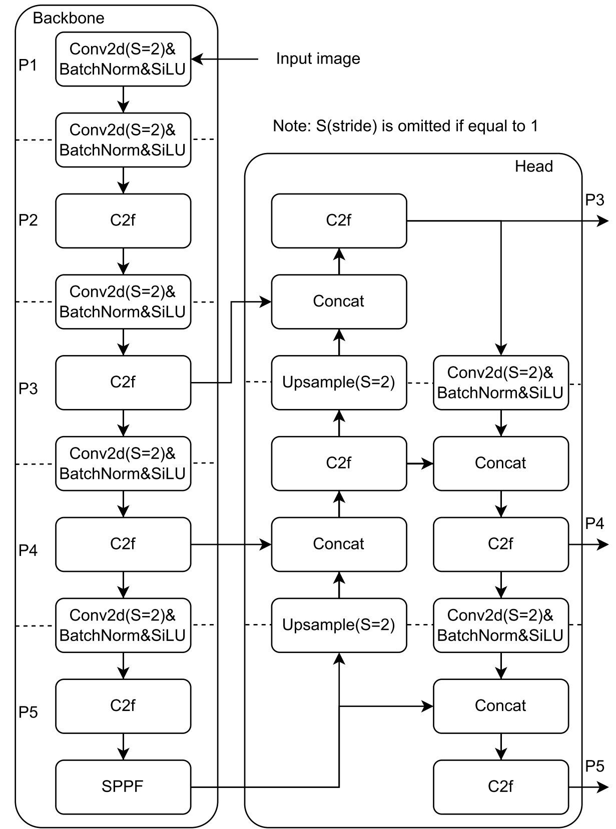


FIGURE 5. YOLOv8's backbone and head strucure.

In Figures 6, and 7, YOLOv8 supports multiple computer vision tasks, including object detection, instance segmentation, and pose estimation, by integrating branch module networks such as Keypoint, Mask, and prototype above the Head. Specifically, as depicted in Figure 6, the instance segmentation model can be regarded as an extension of object detection. In this paper, we utilize the YOLOv8 small model from [19]. The three models utilized are 'yolov8s.pt' for object detection, 'yolov8s-seg.pt' for instance segmentation, and 'yolov8s-pose.pt' for pose estimation. We decompose complex backbone and head structures, excluding DFL, NMS, and MASK, to derive fundamental deep learning operations, as illustrated in Table 1.

For object detection and instance segmentation tasks, the COCO dataset [20] defines a class count (nc) of 80. However, in the context of pose detection within the COCO dataset, the parameter nc is restricted to 1. It is important to note that other datasets may exhibit variations in the number of classes as well. These tasks address distinct objectives and find applications in various scenarios, making YOLOv8 versatile for surveillance, autonomous vehicles, and robotics.

B. RISC-V ARCHITECTURE

RISC-V stands as a prominent open and free Instruction Set Architecture (ISA) in the realm of computer architecture, originating from the University of California, Berkeley

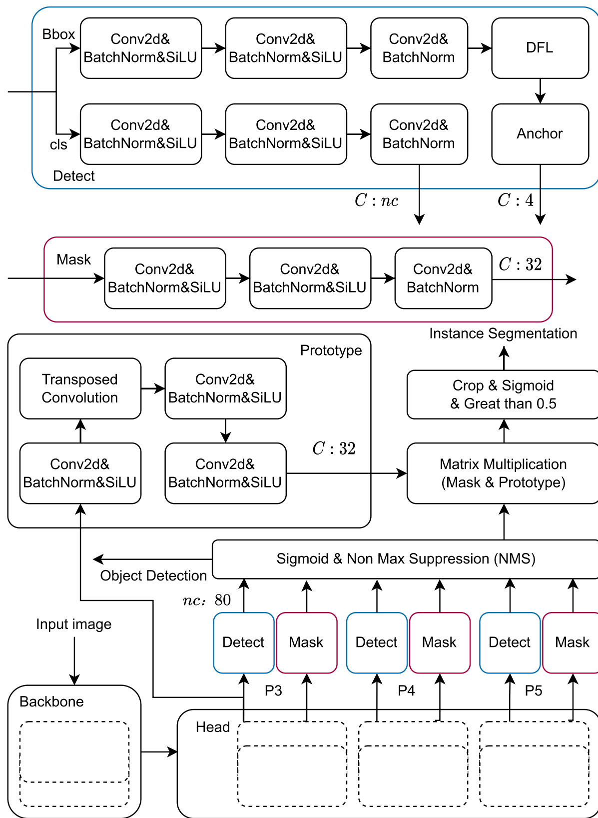


FIGURE 6. YOLOv8's structure for instance segmentation.

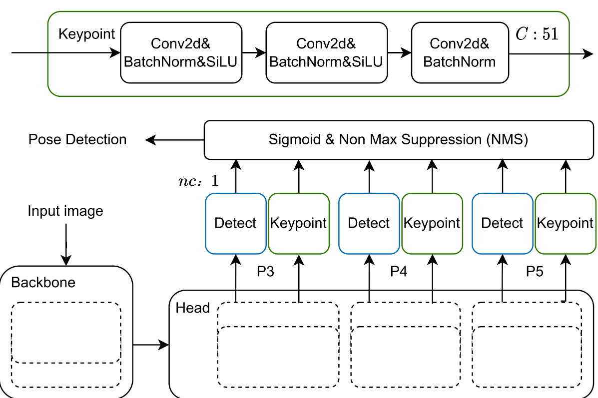


FIGURE 7. YOLOv8's structure for pose detection.

[24]. Diverging from proprietary architectures, RISC-V distinguishes itself through its open standard, fostering collaboration and innovation across researchers, developers, and industries, unburdened by licensing fees. Its paramount strength lies in its open-source nature, granting unfettered access to the core's source intellectual property (IP) without the constraints of licensing issues.

The traditional method of implementing computer architecture involves incremental updates to the ISA. RISC-V diverges from this by adopting a modular approach. Each RISC-V device's instruction set includes foundational Integer ISA modules and optional ISA extension modules. The

TABLE 1. Total number of basic functions(layers).

Model	yolov8s	yolov8s-seg	yolov8s-pose
Convolution 3x3	39	47	45
Convolution 1x1	32	36	35
Transposed convolution	0	1	0
Maxpool 5x5	3	3	3
Addition	6	6	6
Concat	13	13	13
SiLU	65	74	71
Sigmoid	3	3	3
Upsample	2	2	2

Extensions field encodes standard extensions, with each letter representing an instruction set. For instance, "I" designates RV32I and RV64I basic integer instruction sets, "A" signifies atomic extension, and "M" is allocated for integer multiplication and division. "S" and "U" respectively represent supervisor-level and user-level extensions.

Scalability is another key forte of RISC-V, accommodating implementations ranging from simple micro-controllers to sophisticated multi-core processors. This adaptability, coupled with the open architecture, has cultivated a flourishing ecosystem of tools, compilers, and software support.

Our prior investigation [25] introduced a re-configurable deep neural networks accelerating SoC (DNN-AS) using the RISC-V architecture. This SoC integrates a highly flexible and adaptable computing engine (CE) based on a 5-stage RISC-V pipeline, which functions as an Application Programming Interface (API) for custom instructions in ResNet and VGG16. Connected via the AXI bus, the CE efficiently transfers weights and feature map data through direct memory access (DMA) during computations, bypassing the primary RISC-V core. This approach seamlessly integrates specialized functionalities directly into the primary processor, surpassing conventional co-processor methods.

The current study aims to build upon the initial research by addressing limitations identified in the previous study and incorporating recent developments in the field.

C. LLVM-C2RTL DESIGN FRAMEWORK

In [26], the authors propose a novel method for directly describing the RTL (Register Transfer Level) structure of a pipelined RISC-V processor with cache, memory management unit (MMU), and AXI bus interface using the C++ programming language. This processor C++ model serves as a near cycle-accurate simulation model of the RISC-V core. Additionally, the authors introduce the C2RTL framework, which translates the processor C++ model into cycle-accurate RTL description in Verilog-HDL and an RTL-equivalent C model.

The unique aspect of their design methodology lies in the fact that both the simulation model and the RTL model are derived from the same C++ source, simplifying the design verification and optimization processes significantly. The effectiveness of this methodology is demonstrated on a RISC-V processor, which successfully runs Linux OS on an field-programmable gate array (FPGA) board.

Reference [27] use High-Level Synthesis (HLS) tools to implementat of YOLOv7-tiny network on FPGAs, however, it can be challenging to control their cycle-level behavior directly from the software description, often requiring tool-specific pragma annotations and coding styles. To address this, the proposed C2RTL framework allows designers to describe the cycle-level behavior of the logic design in C++ language using a data flow style, while specifying hardware attributes with GCC-style attribute keywords. This approach enables fast simulation, design transparency, and does not involve proprietary languages.

In conclusion, the C2RTL design framework proposed in the paper offers an efficient and accessible method for designing complex processors and system components. This framework is particularly well-suited for IoT devices that demand flexible and low-power instruction sets. Consequently, the toolkit has been chosen to facilitate the design of the AIV-SoC for YOLO.

D. ROOFLINE MODEL AND DATA FLOW

The Roofline model is an analytical framework proposed to establish a correlation between processor performance and off-chip memory traffic [28]. In the foreseeable future, off-chip memory bandwidth is expected to be a critical resource that limits system throughput. To address this concern, the concept of ''operational intensity'' is introduced, which quantifies the number of operations performed per byte of DRAM traffic. For certain tasks such as matrix operations and Fast Fourier Transformation (FFT), the operational intensity increases with the problem size. The on-chip cache size and optimizations will also reduce the number of memory accesses, thereby increasing the operational intensity.

Convolutional Neural Networks (CNNs) predominantly process data within the chip, and the primary bottleneck in input-output (IO) operations is associated with weight data. To address this IO bottleneck, a ping-pong DMA scheme is proposed in [25], enabling concurrent loading of weight data and feature map computation. It is noteworthy that the AXI bit-width of [25] has reached 256 bits. While higher bit-width can effectively reduce transfer times, it also imposes more stringent requirements on circuit pins and packaging. Our work will employ the concept of ''operational intensity'' to investigate the necessity of such high bit-width. Moreover,specifically tailored to YOLOv8, we propose an optimal bit-width configuration that utilizes the fewest possible pins while ensuring overall throughput.

E. ACTIVATION FUNCTION

Reference [29] provides a circuit design for configurable sigmoid and Tanh activation functions using second-order approximation and deviation compensation. This design exhibits superior speed and area efficiency compared to traditional lookup table or polynomial approximation methods. It also utilize the function to perform the post-processing of YOLOv3 and reduced about 0.06% of the MAP accuracy. YOLOv8 also utilize the sigmoid function to calculate the confidence value. However, YOLOv8 introduces more non-linear activity function other that signoid such as the SiLU after convolution, softmax in the DFL layer. Implementing the second-order approximation in hardware solely for the sigmoid function does not yield significant advantages.

Reference [30] presents an enhanced Rectified Linear Unit (ReLu) segmentation correction activation function, termed SignReLu. Experimental findings demonstrate that the proposed activation function expedites convergence, effectively mitigates the gradient vanishing issue, and substantially enhances the accuracy of neural network identification. Due to the ease of hardware implementation and versatility associated with piecewise linear functions (PLFs), we have devised a strategy employing general PLFs to approximate the diverse activation functions within YOLOv8.

III. HARDWARE DESIGN

A. RISC-V AND COMPUTING ENGINE

Based on [25], we present the architectural design of the AIV-SoC system. Specific modifications and enhancements have been introduced to handle the intricate and multifaceted YOLOv8. As depicted in Figure 8, the RISC-V module, initially RV32IMA, has been enhanced to RV32IMASU, introducing extensions for user and supervisor modes. This includes the addition of new extensions for both user and supervisor modes. The 4 data ports, namely the instruction cache (I-cache), data cache (D-cache), feature map caches (FMC), and weight, share the same 64-bit AXI4 primary port. Additionally, we have integrated a memory controller and a Universal Asynchronous Receiver/Transmitter (UART) module on the AXI4 bus to streamline execution and debugging on FPGA.

In Figures 1, 7 and 6, it is evident that YOLOv8 surpasses traditional ResNet networks in complexity. As a result, our AIV-SoC is equipped with a more versatile and efficient CE depicted in Figure 9. Table 2 outlines the implementation of these novel functions (layers) using reformed CE hardware or RISC-V software. Subsection II-C elaborates on our C2RTL-based design methodology, which facilitates easy hardware modifications. Therefore, even if there are new functionalities beyond those outlined in Table 2, we can readily implement them by adapting the hardware design in C++.

Moreover, the frequency of DNN-AS in [25] is diminished due to the prolonged critical path of the general Multiply-Accumulate (MAC) tree array. Our YOLOv8 CE tackles this issue by introducing a multi-cycle path. Concretely, we allocate the MAC tree array and its associated weight register to cycle 2, while the remaining calculation and storage modules are situated in cycle 1.

B. FEATURE MAP CACHE

As shown in Figure 9, we have designed 8 FMC pages, each composed of 16 32-bit 8192-depth static random-access

TABLE 2. Implementation of functions(layers).

function/layer	implementation
SiLU	PLF circuit
Sigmoid	PLF circuit
Softmax	PLF circuit; Softmax shift,sum,multiply circuit
Upsample	Copy circuit
Concat	Copy circuit
Non-maximum suppression	RV32IMASU software
Crop	RV32IMASU software
Greater than	RV32IMASU software
Matrix multplication	General multiply-Accumulate (MAC) tree array
Transposed convolution	General MAC tree array
Add	General MAC tree array
Anchor	RV32IMASU software; General MAC tree array

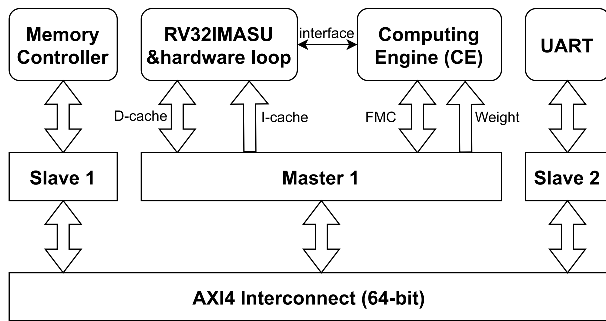


FIGURE 8. Overall architecture of AIV-SoC.

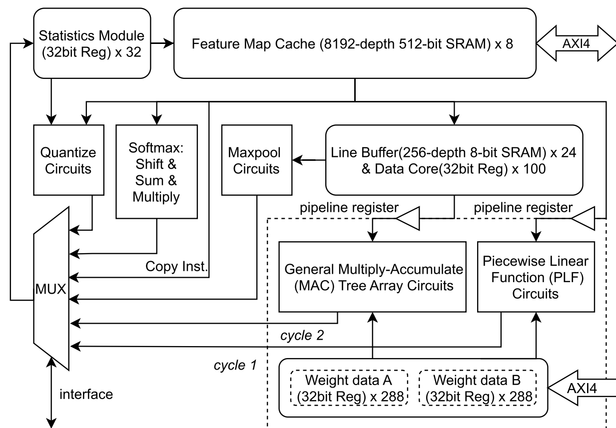


FIGURE 9. Structure of the CE.

memory (SRAM) cells, providing data widths of 512 bits. To accommodate different fixed point number precision requirements, the cache offers two input modes: 32-bit mode and 8-bit mode. In the 32-bit mode, the input is interpreted as 16 32-bit signed integers, which are commonly used to store intermediate calculation results that have not been quantized yet. In the 8-bit mode, the input is processed as 64 8-bit signed or unsigned integers, depending on a signed bit register. Typically, it stores the layer's result after quantization. The use of two memory modes enables more efficient use of on-chip memory with minimal loss of accuracy.

Consider the 32-bit mode. For instance, the pixel in the f channel at position (x, y) will be stored in the f-th cell's $(y \times Nix + x)$ depth. In this way, a single page can store a feature map of size 90×90. However, due to significant variations in channel height and width parameters across different layers in YOLOv8, directly storing data with such a mapping relationship may not be suitable. Therefore, we have adopted a stacked and carry mechanism to effectively manage the data in FMC.

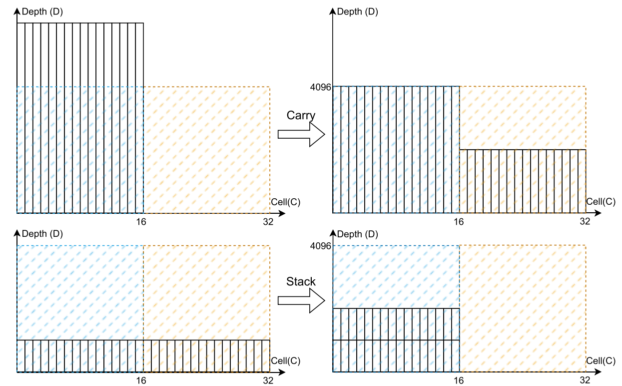


FIGURE 10. Carry and stack mechanisms.

Figure 10 illustrates these mechanisms in the 32-bit mode. The top half picture demonstrates a layer with $Nif = 16$ and $(Niy \times Nix) = 12288$, where the overflowing 4096 depth is carried to the next 16 cells, which resides in the other FMC page. The bottom half image showcases a layer with $Nif = 32$ and $(Niy \times Nix) = 2048$, where the 17th to 32nd layers are stacked up to the 1st to 16th layers. By adopting this methodology, the release of the yellow FMC page for alternative purposes will be facilitated. Given the limitations imposed by on-chip storage technology, this investigation has constrained the cache size to 4MB, thereby restricting the input image resolution of YOLOv8 to a maximum of 352.

While it is possible to concatenate pages to achieve a larger volume, the calculation process allows only one page to be selected for input data, and one page for output data storage at a time.

C. RECONSTRUCTION OF DATA AND ALGORITHMS

Similar to Graphics Processing Units (GPUs), Single Instruction, Multiple Data (SIMD) plays a pivotal role in the optimization of large-scale neural networks. Enabling SIMD necessitates the hardware-level implementation of certain loops. Numerous established loop optimization methodologies, including but not limited to loop unrolling, tiling, and interchange [31], prove to be highly effective in mitigating these challenges. These strategies involve the decomposition of the overarching loop into more manageable subsets, with subsequent unrolling of each subset directly onto the chip.

Reference [25] has compiled a comprehensive list detailing the convolution design variables of ResNet suitable for loop unrolling. These variables encompass (Pkx, Pky), Pif, (Pox, Poy), and Pof, symbolizing the degree of calculation parallelism. Constraints for these variables are defined as $1 \leq P^* \leq N^*$, where P^*, N^* represent any variable with

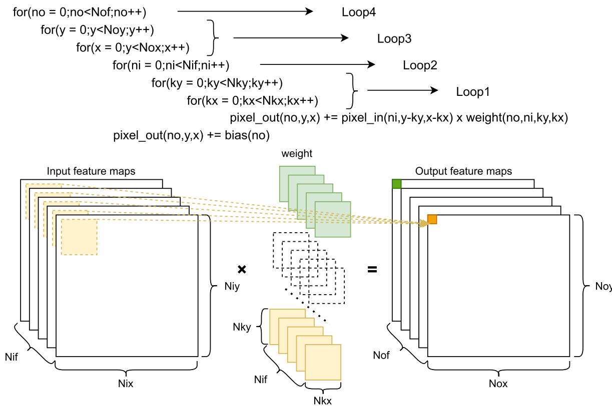


FIGURE 11. Structure of convolution.

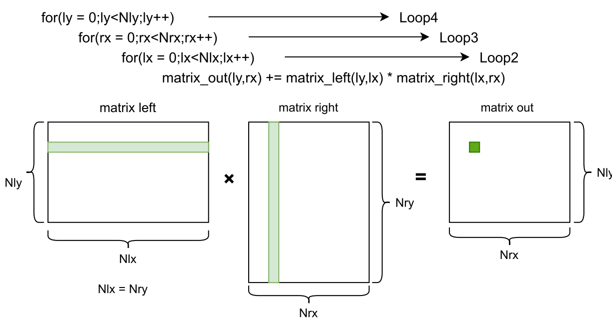


FIGURE 12. Structure of matrix multiplication.

a prefix of P, N. YOLOv8 incorporates various operations beyond convolution, including max-pooling, matrix multiplication, transposed convolution, and up-sampling. Despite the structural distinctions among these operations, they can be categorized into four loops based on their alignment with the addresses of the FMC.

In Figures 11 and 12, we provide pseudo-code representations of convolution and matrix multiplication, incorporating 32-bit mode FMC as delineated in Table 3. The implementation of max-pooling can also be inferred from Figure 11. The symbol N^* within the context of FMC signifies its maximal capacity to hold data along a specific dimension, whereas P^* denotes the upper limit of the attainable loop unrolling coefficient within its input/output bandwidth.

Additionally, data reuse plays a pivotal role in reducing the number of I/O operations. Two forms of data reuse are particularly relevant: spatial reuse and temporal reuse [32]. Spatial reuse implies that data is employed by multiple parallel multipliers, whereas temporal reuse denotes the use of data for multiple consecutive clock cycles. In convolution, weight data can be temporally reused in loop 3, input feature map data can be spatially reused in loop 3 and 4.

In Loop1 of Table 3, numerous unrolling coefficients extended the FMC limitation, introducing certain challenges. However, convolution and max-pooling operations rely on successive sliding windows, resulting in a significant overlap between adjacent windows. To leverage this crucial

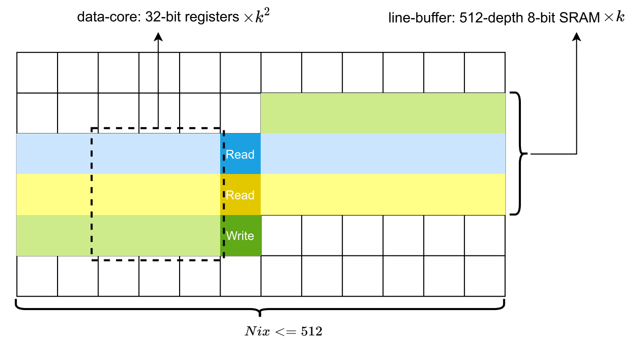


FIGURE 13. Line-buffer and data-core caching scheme.

TABLE 3. Loop dimensions with unrolling variable.

	Coefficient	Loop1	Loop2	Loop3	Loop4
FMC (32-bit mode)	Dimension(N*)	in-depth	in-cell	out-depth	out-cell
	Unrolling (P*)	1	16	1	16
Convolution	Dimension(N*)	(Nkx, Nky)	Nif	(Nox, Noy)	Nof
	Unrolling (P*)	(Pkx, Pky)	Pif	(Pox, Poy)	Pof
Max-pooling	Dimension(N*)	(Nkx, Nky)	/	(Nox, Noy)	Nof
	Unrolling (P*)	(Pkx, Pky)	/	(Pox, Poy)	Pof
Matrix Multiply	Dimension(N*)	/	Nlx	Nrx	Nly
	Unrolling (P*)	/	Plx	Prx	Ply

characteristic and enhance the efficiency of external memory bandwidth, an innovative line-buffer and data-core caching scheme were introduced by [25] and [33]. While [33] involves two-port RAMs, [25] employs a combination of SRAM and Register structures. It is noteworthy that such two-port RAMs entail greater area and power consumption compared to their single-port counterparts. Consequently, we opt to implement the sliding window's caching scheme as proposed by [25].

For a kernel with a square size denoted as (k, k), when $k > 1$, data from the FMC is transferred to line-buffer and data-core caching scheme. Figure 13 illustrates the hardware resource utilization for a $k \times k$ kernel when unrolling loop1. The existing data from the line-buffer (depicted in dark blue and yellow) and the new data from the FMC (depicted in dark green) are then transferred to the data-core register array, refreshing sliding windows at the next clock cycle.

Max-pooling consumes $Pof \times k$ line-buffers and $Pof \times k^2$ data-cores, while convolution requires $Pif \times k$ line-buffers and $Pif \times k^2$ data-cores. Considering the overall hardware resources and constraints, in this paper, we establish 24 32-bit 512-depth SRAM line-buffers, supporting a maximum input feature map size (Nix) of 512. The number of data-cores is set to 100 to accommodate max-pooling with $(5, 5)$ kernel size and $Pof = 4$. Detailed information regarding the P^* coefficient for various layers in YOLOv8 is presented in Table 4.

D. GENERAL MULTIPLY-ACCUMULATE TREE ARRAY

To unroll the convolution and matrix multiplication loops 1, 2, and 4, denoted as P^* in Table 4, we have developed a versatile MAC tree array comprising 16 independent MAC tree circuits. These 16 MAC trees share the input feature

TABLE 4. Unrolling Parameters (P*) of different layers.

	Loop1	Loop2	Loop3	Loop4
Maxpool 5×5	(5,5)	/	(1,1)	4
Matrix Multiplication	/	16	(1,1)	16
Convolution 3×3	(3,3)	8	(1,1)	16
Convolution 1×1	(1,1)	64	(1,1)	16

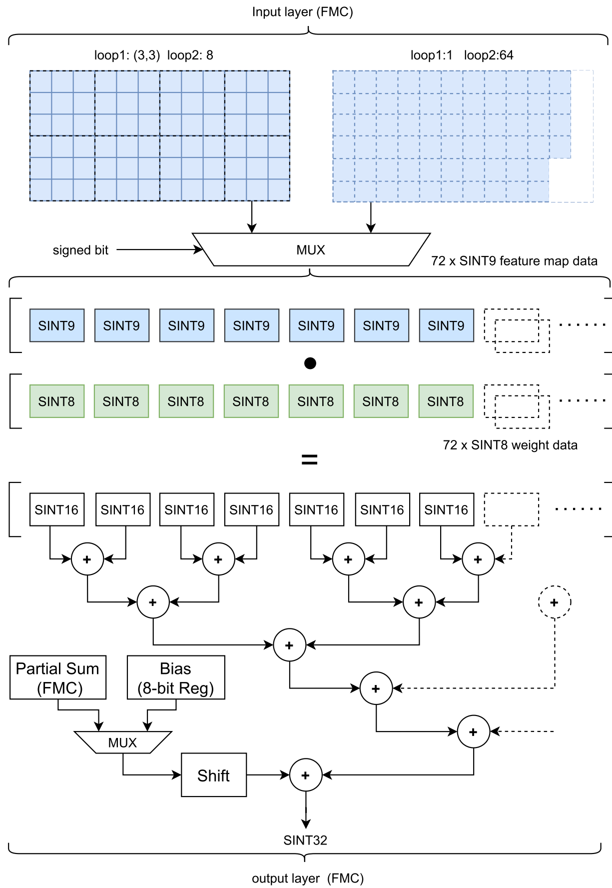


FIGURE 14. MAC tree.

map's data while employing distinct weights. Consequently, a set of 16 contiguous channels in loop 4 can be computed concurrently.

Each MAC tree is responsible for the hardware implementation of loops 1 and 2. Given that the majority of layers in YOLOv8 utilize traditional 3 × 3 convolutions, we set the number of MAC operations for each MAC tree as $loop1 \times loop2 = (3 \times 3) \times 8 = 72$. Figure 14 illustrates a MAC tree, showcasing incorporation of adjusted bias data or partial sums from the preceding group. The blue square represents a pixel on the feature map, while the white square indicates zero padding. Despite introducing some hardware inefficiency, given the widespread utilization of 3 × 3 convolutions in YOLOv8, the associated loss is considered negligible.

E. DIRECT MEMORY ACCESS SCHEME

To fully maximize idle clock cycles, we also inherit the ping-pong scheme from [25]. As shown in Figure 9, Weight data "b" and "a" are alternately chosen for MAC array

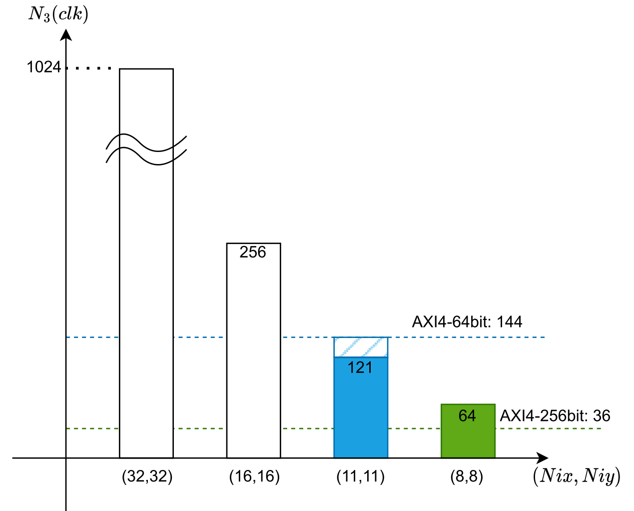


FIGURE 15. Operational intensity and weight loading clocks.

calculations and data loading for the subsequent group through the AXI4 port. This strategy ensures that calculations do not idle during the loading of weight data. Given that weight data can be reused in Loop3 during convolution and matrix multiplication, reloading weight data is only necessary for different instances of Loop2 and Loop4. The fixed size of weight "b" and "a" dictates that the data-loading cycles for each Loop2&4 depend solely on the AXI4 bidwidth. Hence, the Loop3's dimension is closed to the concept "operational intensity" proposed in [28]. Consequently, we have $N_3 = (Nox \times Noy)\ or\ Nrx$ un-idled clocks to load the weight of the next loop2&4.

As shown in Figure 9, single weight data is composed of 288 32-bit registers. Data writing is executed using unidirectional DMA via AXI port. As depicted in Figure 15, our AIV-SoC utilizes a 64-bit AXI port, requiring 144 clocks to load the subsequent weight data. In contrast, our previous conference proceeding [34] proposed a 256-bit configuration, resulting in a 36-clock requirement. Notably, the study in [34] assumes an input image size of 256 × 256, with the smallest feature map represented by the green column as (8, 8). In this paper, we have expanded the dimensions of the input image to 352 × 352, and the smallest feature map is identified by the blue column, sized at (11, 11), where the hatched column signifies the idle clocks awaiting weight loading. Despite the assertion in [34] that all layers avoid idle clocks introduced by weight loading, the incorporation of a 256-bit AXI4 port poses challenges in terms of packaging. In contrast, our latest design with a 64-bit configuration strikes a harmonious balance between idle clock cycles and the number of IO pins.

F. TRANSPOSED CONVOLUTION

Transposed convolution, also known as fractionally strided convolution or deconvolution, is a technique used in neural networks for up-sampling or increasing the spatial resolution of feature maps. Unlike standard convolution layers that perform down-sampling through the use of pooling operations,

transposed convolution layers increase the spatial resolution of input feature maps. The transposed convolution operation involves using learnable kernels to map input pixels to a larger output space. During the transposed convolution, the kernels are applied to the input with a specific stride, resulting in the expansion of the feature map.

This operation is particularly useful in tasks such as image segmentation and image generation, where the network needs to learn to generate high-resolution details from low-resolution inputs [35]. Transposed convolution layers are often used in decoder parts of neural networks, especially in architectures like U-Net and various generative models. In terms of implementation, transposed convolution layers are typically realized using operations like "convolution with fractional input stride" in deep learning frameworks. These layers have learnable parameters (weights) that are adjusted during training to enable the network to effectively up-sample and generate detailed information from coarser representations.

It's worth noting that the term "deconvolution" in the context of neural networks is a bit of a misnomer, as it doesn't perform true deconvolution in the mathematical sense. Instead, it refers to the transposed convolution operation, which has the effect of up-sampling the input.

References [36] and [37] introduces a hardware acceleration design transposed convolutions, enabling real-time execution of segmentation tasks. Given that the majority of transposed convolution in segmentation model employ a stride of 2, it implies the necessity to insert zeros between each datum in the input layer during convolution. To addresses the computational intensity and inefficiency issues, [37] introduced a dataflow exploration method by dividing filters and corresponding input feature maps into four patterns and applying the Winograd algorithm. Due to the additional latency and computational overhead introduced by image scaling, the majority of models tend to favor the first approach. The YOLOv8 architecture, employed in this study, also adopts the strategy of utilizing a kernel size divisible by the stride for its transposed convolutions, where both the stride and kernel size are set to 2.

Reference [38] highlights another challenge known as "uneven overlap" in transposed convolution, where certain areas receive more emphasis than others, especially when the kernel size is not divisible by the stride. This uneven overlap manifests as a checkerboard-like pattern, which is more pronounced in two dimensions due to the squared effect. Two suggested approaches to mitigate these issues include using a kernel size divisible by the stride (equivalent to "sub-pixel convolution") and separating up-sampling from convolution by resizing the image before applying convolution layers.

In light of the overarching constraints and implementation considerations, we formulated the transposed convolution with a stride of 2 in CE. This design specifically mandates a kernel size that is divisible by the given stride. Similar to the four configurations delineated in [37], we have also partitioned our transposed convolution kernel into four

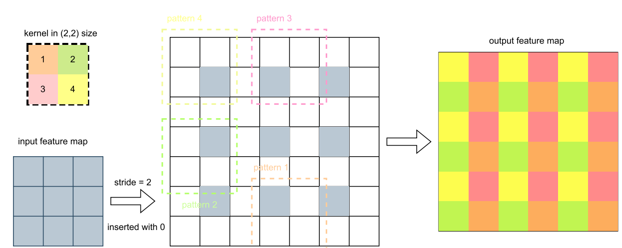


FIGURE 16. 4-pattern mode of Transposed Convolution.

TABLE 5. YOLOv8's transposed convolution N* and P*.

Mode	Coefficient	Loop1	Loop2	Loop3	Loop4
normal	Dimension(N*)	(2, 2)	128	(2*Nix, 2*Niy)	128
	Unrolling (P*)	(2, 2)	16	(1, 1)	16
4-pattern	Dimension(N*)	(1, 1)	512	(Nix, Niy)	128
	Unrolling (P*)	(1, 1)	64	(1, 1)	16

distinct patterns as shown in Figure 16. Each pattern is equivalent to a traditional convolution with a stride of 1 and a kernel size of $(\frac{Nkx}{2}, \frac{Nky}{2})$. Implementing this 4-pattern mode introduces a repetition of loop2, occurring four times. As shown in Table 5, we listed the P^*, N^* for the YOLOv8's transposed convolution in both normal and 4-pattern mode. By computing the expression in (1), where "Inst" denotes the number of instructions, and P_i, N_i denote the dimensions or unrolling variables under loop "i", it is discerned that employing the 4-pattern mode consumes only one-fourth of the number instruction compared to the normal mode.

$$Inst = \prod_1^4 \frac{N_i}{P_i} \tag{1}$$

G. MAXPOOLING CIRCUITS

Thanks to the new SPPF structure, we only need to focus on max-pooling with a 5×5 kernel. Taking into account register resources, we designate P_4 as 4 for the max-pool5×5 operation, resulting in 4 max-selection circuit trees, as illustrated in Figure 17.

H. STATISTIC BLOCK FOR DYNAMIC FIX-POINT SCHEME

In circuit design, the execution of floating-point multiplication operations necessitates the incorporation of numerous shift circuits, resulting in heightened circuit latency. Consequently, the fixed point scheme is employed as a strategy to optimize the circuit, as proposed by [25] and [39]. These researchers introduced a dynamic fixed point (DFP) scheme tailored for managing feature map and weight data, demonstrating superior accuracy compared to traditional unified tail quantization. Nevertheless, the quantization per layer (tensor) proves insufficiently meticulous for convolution layers with a substantial number of channels.

Consider a convolution layer composed of $L_f = Nof \times Noy \times Nox$ feature map data. In their approach, a single public exponent is assigned to each of these data sets, a practice that may lead to significant truncation errors when L_f is exceptionally large. To mitigate this issue, we propose the adoption of per-group quantization. The number of groups aligns with the FMC's P^* under loop4,

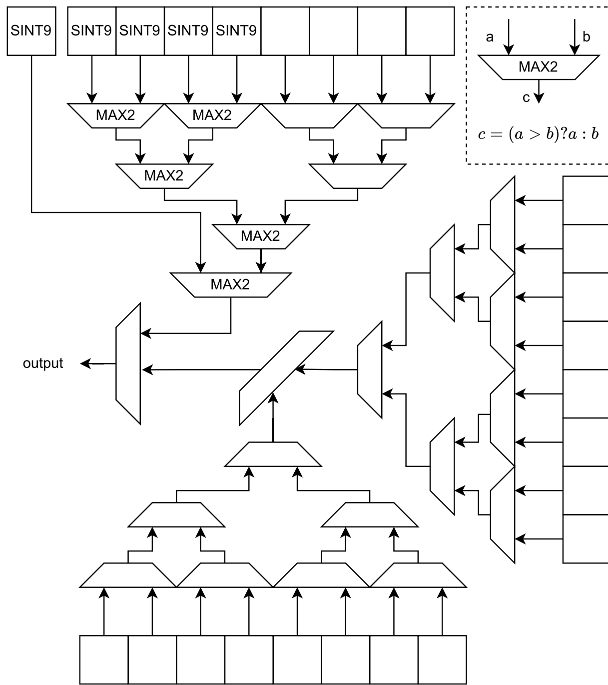


FIGURE 17. Miax-selection circuit tree.

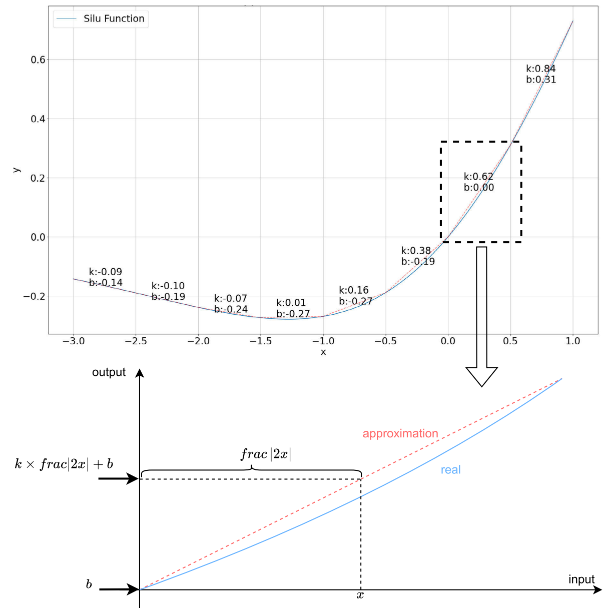


FIGURE 18. PLF approximation scheme.

enabling the insertion of a statistic module in CE. The number of each group's data will be $G_f = 16 \times Noy \times Nox$. The statistical module tallies the number of highest bits, representing the absolute values of output results from the MAC array. Reference [25] also introduced the concept of a tolerance threshold, allowing for the truncation of certain pixels. In this paper, we utilize a tolerance threshold of 1/8192 for YOLOv8 following extensive validation. Subsequently, each group leverages statistical information and the tolerance threshold to determine its respective exponent. This refined quantization methodology is designed to mitigate truncation errors effectively.

Moreover, since unquantized data (32-bit mode) occupies four times more space than quantized data (8-bit mode), the implementation of per-group quantization also diminishes the on-chip cache requirement. As demonstrated in Figure 9, when the WE signal of the FMC is active, the write data's copy is automatically sent to the statistics module.Given that not all data serves as the final output layer's result, we have introduced a double-level register mechanism. The first register automatically accumulates distribution information related to the highest bit. Custom instructions are employed to either accumulate the first register data into the second register or clean the data in the first register. Upon the completion of all calculations within the designated group, the secondary register discerns and generates the optimal exponent (truncation bit) for the current group.

I. PIECEWISE LINEAR FUNCTION MODULE

The YOLOv8 framework incorporates various non-linear mapping functions, specifically SiLU, softmax, and sigmoid. The softmax function can be deconstructed into its inverse function and the exponential function. Traditionally, the exponential function is approximated through the Taylor series method [40]. However, due to the necessity of introducing additional multiplication and addition terms in the Taylor expansion, this significantly amplifies the overall computational workload. To mitigate this challenge, we employ the PLF module to approximate the values of the non-linear functions.

To accommodate the input range of the majority of non-linear functions, we have divided the interval $[-8, 8]$ into 32 segments with a step size of 0.5. For each segment, we derived the corresponding slope k and intercept b from the weight register. Assuming that the $frac()$ function returns the fractional portion of a tensor, the resulting computation is expressed as $k \times frac(|2x|) + b$. As illustrated in Figure 18, a limited number of mantissa bits are adequate for accurately representing high-precision input. The outputs are quantized on a per-group basis, following the same procedure as other operations utilizing the statistics module.

IV. SOFTWARE DESIGN

In this study, we employed the pre-trained YOLOv8s model from Ultralytics to serve as the neural network running on the AIV-SoC platform. To optimize the network architecture, we integrated the batch normalization layers with convolution layers using the official 'fuse()' function.

A. MODEL ADJUSTMENTS AND QUANTIZATION

The YOLOv8's weights are quantized using the same quantization scales as feature map data. For example, the convolution layer's weight data consists of $L_w = Nkx \times Nky \times Nif \times Nof$ data points. The size of L_w becomes prohibitively

large as the dimensions of Nif and Nof increase in the latter part of YOLOv8. Consequently, we propose dividing them into groups to reduces the number of data sharing the same public exponent. Similar to layer's per-group quantization, each group have $G_w = Nkx \times Nky \times Nif \times 16$ data points.

As this paper primarily concentrates on the hardware design for model inference, we choose a straightforward post-training symmetric static quantization of the model. In the future, incorporating Quantization Aware Training (QAT) could potentially enhance accuracy further.

Notably, while per-channel quantization achieves accuracy closest to the original float, it necessitates a substantial number of exponents. Additionally, given that the coefficients for the majority of computation layer loop4 are set at 16, per-channel quantization will result in increased computational load within the loop. In light of the inherent trade-off between precision and circuit complexity, we have, therefore, incorporated per-group quantization into the AIV-SoC framework.

B. CONVOLUTION

Algorithm 1 delineates the configuration of the nested loop structure governing the convolution operation with a kernel size of (3, 3). In each iteration, the *load_weights*() commands are employed to retrieve the necessary weight data for the subsequent group. In most cases, the *load_weights*() commands occurs seamlessly without disrupting the computational process, thereby enabling the concurrent execution of *calc_conv*3() alongside the weight-loading procedure.

The inner loop, featuring the *ni* statement, accumulates the partial sum from 8 adjacent input channels. As a result, we only need to save the statistics of the final iteration. Hence, we incorporate the *clean_statistics*() function to reset the data in the first register and the *save_statistics*() function to accumulate the data from the first register into the second register. Additionally, to enhance calculation efficiency, we also implement the zero overhead loop mechanism from [25] in software design.

Algorithm 1 Convolution3 × 3 Loop

```
1:  iteration ← sizein × sizein
2:  load_weights()
3:  wait()
4:  for no = 0 to Nof by 16 do
5:      load_bias()
6:      for ni = 0 to Nif by 8 do
7:          set_layer()
8:          clean_statistics()
9:          load_weights()
10:         for zero_over_headloop(iteration) do
11:             calc_conv3 × 3()
12:         end for
13:     end for
14:     save_statistics()
15:     generate_exp()
16: end for
```

C. TRANSPOSED CONVOLUTION

In Section III-F, our approach involves employing a 4-pattern mode for transposed convolution. Within YOLOv8's segmentation model, the transposed convolution is characterized by a kernel size of 2 and a stride of 2. This operation is specifically applied to ''up-sample'' the prototype layer. The implementation details of this mechanism are elucidated in Algorithm 2.

Algorithm 2 Transposed Convolution Loop

```
1:  iteration ← sizein × sizein
2:  load_weights()
3:  wait()
4:  for no = 0 to Nof by 16 do
5:      load_bias()
6:      for pattern = 0 to 3 do
7:          for ni = 0 to Nif by 64 do
8:              clean_statistics()
9:              set_layer()
10:             load_weights()
11:             for zero_over_headloop(iteration) do
12:                 calc_convtranspose2 × 2()
13:             end for
14:         end for
15:         save_statistics()
16:     end for
17:     generate_exp()
18: end for
```

D. MATRIX MULTIPLICATION

The segmentation model of YOLOv8 incorporates the task of object detection. Following the object detection process, the obtained results necessitate matrix multiplication with predictions post Non-Max Suppression (NMS) and the prototype layer. The matrix representing predictions post NMS is positioned on the left, with its variable Nly denoting the count of objects remaining after NMS. Conversely, the prototype layer is situated on the right, assuming that the input image is of dimensions $(imgsx, imgsy)$. Here, $Nrx = imgsx \times imgsy/16$, and $Nlx = Nry = 32$.

To execute the matrix multiplication, predictions after NMS must initially be retrieved from the CE, followed by re-composition, and subsequently flushed back as weight data. A more in-depth understanding of this process is elucidated in Algorithm 3.

E. MAXPOOL

The spatial pyramid pooling fast (SPPF) in YOLOv8s utilizes three max-pool layers with a kernel size of 5×5. According to Table 3, max-pool5×5 is composed of $Nof/4$ groups. As the max-pool operation outputs only the maximum pixel value from sliding windows in the line buffer and data core, quantization is not required. Algorithm 4 presents the pseudo code for the max-pool5×5 operation.

Algorithm 3 Matrix Multiplicatio Loop

```
1:  iteration ← Nrx
2:  fetch_prediction_after_NMS()
3:  reconstruct()
4:  load_weights()
5:  wait()
6:  for no = 0 to Nly by 16 do
7:      for ni = 0 to Nlx by 64 do
8:          set_layer()
9:          load_weights()
10:         clean_statistics()
11:         for zero_over_headloop(iteration) do
12:             calc_convtranspose2 × 2()
13:         end for
14:     end for
15:     save_statistics()
16:     generate_exp()
17: end for
```

Algorithm 4 Maxpool5 × 5 Loop

```
1: iteration ← sizein × sizein
2: for no = 0 to Nof by 8 do
3:     set_layer()
4:     for zero_over_headloop(iteration) do
5:         calc_maxpool5 × 5()
6:     end for
7: end for
```

F. CONCAT AND UPSAMPLE

The Concat and Upsample layers do not perform calculations; they simply copy the channels of the input layer's data. The data-flow for these layers is represented as the "Copy Inst." shown in Figure 9. Assuming that the first channel of the input layer will be shifted to the beginning of the output channels (*Bof*), the software algorithm is illustrated in Algorithm 5.

Algorithm 5 Pseudo Code for Copy

```
1: iteration ← sizeout × sizeout
2: for no = 0 to Bof by 16 do
3:     set_layer()
4:     for zero_over_headloop(iteration) do
5:         copy()
6:     end for
7: end for
```

G. SHORTCUT CONNECTION (RESIDUAL LAYER)

Some bottlenecks in YOLOv8 involve shortcuts, which entail an element-wise addition of two layers. Due to the potential discrepancy in the DFP's exponents of the two input layers, we employ the MAC to perform the addition. One layer is directed to the MAC's input feature map data, while the other layer is sent to the partial sum port, where it will be appropriately shifted to align with the former. In the end, the sum result also necessitates per-group quantization. The software data-flow is illustrated in Algorithm 6.

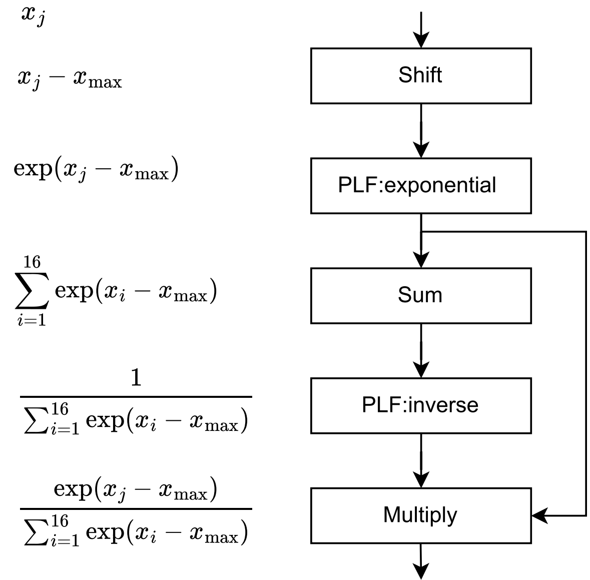


FIGURE 19. Decomposition of max-shifted softmax.

Algorithm 6 Pseudo Code for Addition

```
1:  iteration ← sizein × sizein
2:  load_weights()
3:  wait()
4:  for no = 0 to Nof by 16 do
5:      set_layer()
6:      clean_statistics()
7:      for zero_overhead_loop(iteration) do
8:          calc_add()
9:      end for
10:     save_statistics()
11:     generate_exp()
12: end for
```

H. SILU, SIGMOID AND SOFTMAX

YOLOv8 incorporates various non-linear mapping functions, including SiLU, softmax, and sigmoid. Modifying SiLU and sigmoid for the required task is as simple as adjusting parameters k and b within the PLF module's software. However, softmax necessitates an custom instruction beyond PLF, as depicted in Figure 19. To mitigate issues related to overflow and underflow, we introduce the concept of max-shifted softmax. This technique entails subtracting the maximum value from each element before applying the exponential function, ensuring both numerical stability and computational efficiency.

I. SWAP MEMORY

After completing each layer, the software will check if the data can be freed if the following layer does not require them.

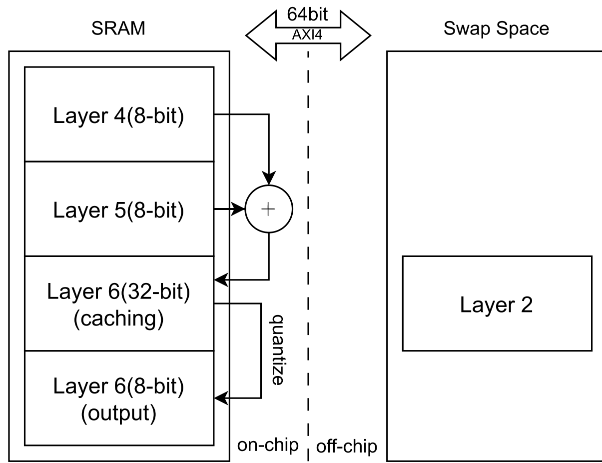


FIGURE 20. Swap memory mechanism.

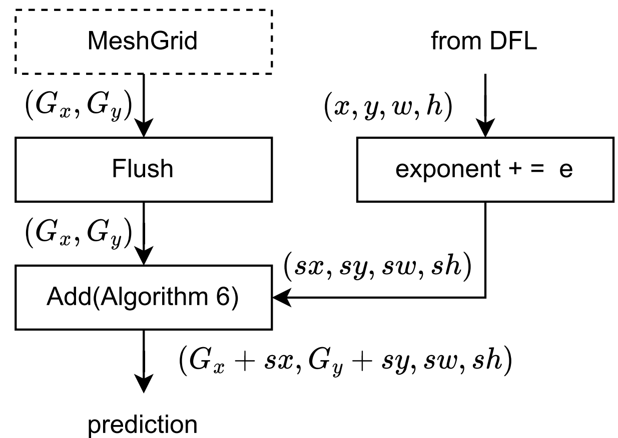


FIGURE 21. Anchor point data flow.

However, in some layers, there is no available space in the FMC to be freed, and the remaining memory is insufficient for the current layer. To address this issue, we have introduced a swap memory mechanism, similar to the swap memory in computers. In this context, the swap memory serves as virtual memory or paging space, which is a specialized memory area within the computer system used to temporarily store infrequently accessed feature map data when the physical on-chip SRAM becomes insufficient.

Assuming the current layer is Layer 6 with Addition operation, and its inputs are 4 and 5. Layer 2 will be used later. Layer 6 has 2 space, one the the output result of 8 bit, the other is 32-bit un-quantized data temporarily cached. Figure 20 depicts the operational mechanism of swap memory, involving alternate processes of addition and quantization. Upon completion of layer6 calculations, the 32-bit cache space is released, allowing Layer 2 to be flushed back into on-chip SRAM. Given the opportunity to modify YOLOv8, we would consider adjusting the layer sizes and structures to mitigate the need for swapping memory. This adjustment would allow the AIV-SoC to reduce its reliance on AXI bandwidth.

J. ANCHOR POINT

YOLOv8 employs grid cells as anchors for determining box positions in an image. To account for this, grid cell bias information needs to be integrated into the boxes from the DFL layer. In Figure 6, there are three detect head blocks with varying layer sizes and corresponding strides: 8, 16, and 32. Consequently, the values of (x, y, w, h) from the DFL are scaled by 8, 16, or 32, depending on the associated detect block. The *torch.meshgrid*() API is then utilized to generate three feature maps for grid cells, which are subsequently added to the scaled boxes.

In our AIV-SoC, leveraging the public exponent mechanism allows us to simplify the scale operation by adding an exponent of 3, 4, or 5. Following this, we create an equivalent grid cell in the off-chip swap memory. Once a softmax layer among the three is completed, its corresponding grid cell layer is transferred to on-chip SRAM. Ultimately, an addition layer, illustrated in Algorithm 6, is utilized to add the scaled boxes and grid cell. The complete data flow about anchor point is illustrated in Figure 21, where the dotted rectangle denotes off-chip operations.

V. SIMULATION RESULT

A. HARDWARE SIMULATION

The hardware design is implemented using C++ and compiled using LLVM 11.0.0 and C2RTL 4.2.8 versions. The C++ code is then converted into an RTL Verilog file. The Front-End hardware design is synthesized with Synopsys Design compiler. Initially, we employ the TSMC 28nm SRAM memory compiler, TSN28HPCPD127SPSRAM_20120200, to produce 8 individual single port 32-bit 8192-depth SRAM cells. Subsequently, we utilize the compiler TSN28HPCPUHDSPSRAM_20120200 to synthesize the circuit incorporating these memory cells.

The simulation constraint and outcomes for AIV-SoC in comparison to the basic RISC-V SoC without CE are illustrated in Table 6. In comparison to [25] and [34], our enhancements involve the incorporation of a memory controller and UART module within the AXI4 bus architecture. Consequently, the area and power consumption of the base RISC-V design have increased. However, the power differential between the AIV-SoC and the basic RISC-V SoC remains nearly constant.

Prior works ([25], [34]) adopt traditional single-cycle path designs, wherein achieving higher clock frequencies can be challenging due to the stringent timing requirements imposed on critical paths. To address this challenge, we employ a multi-cycle patch design in CE, segmenting the critical paths across two clock cycles. This approach provides flexibility in meeting timing constraints and opens the possibility of

TABLE 6. Hardware resource utilization.

	AIV-SoC	RISC-V SoC
Frequency	595Mhz	632Mhz
Clk period	1.68ns	1.58ns
Area	$7.91mm^2$	$0.34mm^2$
Memory size	4,276KB	69KB
Power	202.5mW	114.3mW
Temperature	25°C	
Voltage	0.9V	
Clk uncertainty	0.3ns	
Max transition	0.3ns	
Max fanout	32	

achieving higher clock frequencies. Consequently, the SoC presented in this paper attains a higher frequency than those in the referenced works. Furthermore, the flexibility offered by the multi-cycle path design contributes to improved design scalability. The segmentation strategy can be readily adapted to accommodate changes in technology, environment, or other design constraints.

To underscore the power efficiency of our design, we conducted a comprehensive comparison between our AIV-SoC and other contemporary platforms. This comparative analysis centered on the average energy consumption and frame processing time when handling unprocessed images with YOLOv8s, featuring a 352-input image size. To show our AIV-SoC's compatibility, we listed the results from resnet50. although throughput is not as good as YOLOv8.

Given the absence of existing literature on FPGA implementation of YOLOv8, we referred to related models such as YOLOv2 [33] and YOLOv7-tiny [27]. We also present the results for the GTX1080Ti (GP102) using the official PyTorch API function *val*() from [19]. Its using the default COCO dataset. The input resolution of the measurement is 352 and the default batch size is 16. Since each platform deploys different models, we standardized the evaluation metrics to ensure fairness, utilizing throughput (TOP/s) and energy consumption per billion operations (Energy per GOPs) to assess performance. To begin, we utilize the "thop" library to acquire the FLOPs (Floating Point Operations Per Second) of each model. By encompassing the entire execution cycle and frequency, we can derive the throughput of the platform. Subsequently, dividing the power by the throughput enables us to ascertain the Energy per GOPs.

Due to the absence of chip fabrication and validation, as well as the cost implications tied to chip production volume, we are currently unable to estimate the retail price of our AIV-SoC. As a reference point, we have compared it with the K230 chip, the latest generation SoC product in Canaan Technology's Kendryte series of AIOT chips,It has 2 CPU with max 800Mhz and 1600hz respectively, it can deal with many AI model including YOLOv5. In Table 7 we present the result with int8 yolov5s. The k230 is priced at approximately $50. Therefore, if our design were to enter commercial production at scale, leveraging the cost advantages of utilizing the more economical TSMC28nm process, our production costs would undoubtedly be lower.

As depicted in Table 7, our AIV-SoC achieved a throughput (TOP/s) equivalent to 13% of the GTX1080Ti. Moreover, our power efficiency, defined by the Energy per GOPs, was approximately 160 times superior to that of the GTX1080Ti, 3 times better than K230, and 37 times superior to the Arria-10 GX1150 FPGA. Compared to FPGA-based designs, our AIV-SoC has the potential for lower costs on a mass production scale. With these advantages, Considering these factors, our AIV-SoC has great potential for applications in lower-end devices such as autonomous vehicles or agricultural monitoring systems.

B. COCO OBJECTION DETECTION SIMULATION

Compared to the PLF, Int8 quantization introduces a higher level of error. In this paper, we are undertaking a comparative analysis among three distinct quantization models: per-tensor, per-group, and per-channel INT8 quantization models, alongside the original official model in [19]. Table 8 presents a juxtaposition of the mean Average Precision (mAP) scores at input resolutions of 256, 352, and 640 for YOLOv8s Object Detection on the COCO [20] val2017 dataset. The simulation results of an exemplary image at an input resolution of 352 × 352 are presented in Figure 22, utilizing both the AIV-SoC's per-group quantization model and the original floating-point model, visualizing a 0.8% mAP gap.

C. GLOBAL WHEAT 2000 OBJECTION DETECTION SIMULATION

The Global Wheat 2000 [42] dataset is pivotal for wheat genetics and agricultural research. It offers high-resolution images and detailed annotations, enabling researchers to develop algorithms for various tasks like wheat variety identification and disease detection. Additionally, the Global Wheat Head Dataset is widely used for wheat head detection tasks, providing diverse images for plant phenotyping and crop management. These datasets play a crucial role in advancing wheat production and food security initiatives.

We trained the Global Wheat 2000 object detection model using the YOLOv8s architecture, employing the same training methodology as the example program detailed in [19], with 100 epochs and a default input image size of 640. In the Global Wheat 2000 dataset, the number of classes (nc) is 1, which impacts the head detection process. Consequently, the overall inference time is slightly smaller compared to the COCO val2017 dataset. However, during our testing, the time difference on both AIV-SoC and GTX1080ti devices is negligible. Therefore, we only present the mAP table and a pair of example image.

While training with an image size of 352 may yield better precision for inference at 352 resolution, it is noteworthy that deploying pre-trained generic models tailored to specific resolutions presents challenges. Hence, we opted to adhere to the default resolution of 640 for training, aligning with typical real-world application scenarios.

TABLE 7. Comparison of hardware metrics.

	This work				NVIDIA GTX1080Ti			Arria-10 GX1150 [33]	XC7Z7100 [27]	K230 [41]
Frequency (MHz)	595				1481			204	100	800/1600
Technology	TSMC 28nm				TSMC 16nm			Intel 20 nm	TSMC 28nm	TSMC 12nm
Bandwidth(GB/s)	4.76				484			-	-	-
Precision	Int8				Float32			Int8	Int13	Int8/16
Area (mm^2)	7.9				471			-	-	-
Power (W)	0.2				250			26	10.79	1
Model	yolov8s	yolov8s-seg	yolov8s-pose	resnet50	yolov8s	yolov8s-seg	yolov8s-pose	yolov2	yolov7-tiny	yolov5s
Throughput(TOP/s)	0.58	0.71	0.592	0.455	4.35	5.35	4.34	1.98	0.918	0.912
Energy/GOPs (mJ)	0.345	0.282	0.338	0.43	57.5	46.7	57.6	13.12	11.72	1.09

TABLE 8. YOLOv8s mAP 0.5-0.95 of objection detection.

Image size	Quantization	mAP 0.5-0.95			
		all	small	medium	large
640	Float	44.9%	25.7%	49.9%	61.0%
	Per-channel	44.5%	25.5%	49.6%	60.6%
	Per-group	43.8%	24.9%	48.8%	60.0%
	Per-tensor	43.2%	24.5%	48.0%	59.2%
352	Float	37.3%	13.6%	40.8%	61.7%
	Per-channel	37.1%	13.3%	40.6%	61.6%
	Per-group	36.5%	12.8%	39.7%	60.9%
	Per-tensor	36.0%	12.7%	38.8%	59.8%
256	Float	31.3%	8.0%	32.8%	57.6%
	Per-channel	31.1%	8.0%	32.3%	57.2%
	Per-group	30.6%	7.6%	31.6%	56.1%
	Per-tensor	30.0%	7.5%	31.0%	55.0%

TABLE 9. YOLOv8s objection detection mAP in global wheat 2000.

Image size	Quantization	mAP 0.5-0.95	mAP 0.5
640	Float	70.6%	98.2%
	Per-channel	69.5%	98.2%
	Per-group	69.4%	98.1%
	Per-tensor	68.3%	98.1%
352	Float	64.1%	96.3%
	Per-channel	63.1%	96.1%
	Per-group	62.2%	95.7%
	Per-tensor	60.5%	95.7%
256	Float	50.2%	88.6%
	Per-channel	49.6%	88.3%
	Per-group	49.1%	87.9%
	Per-tensor	47.4%	87.4%

D. COCO INSTANCE SEGMENTATION SIMULATION

In parallel with Table 8, we conducted a comparative analysis of four distinct quantization models applied to the YOLOv8 instance segmentation model, as illustrated in Table 10. The test dataset is COCO-seg [20], an extension of the COCO dataset. Given the absence of prior research dedicated to the instance segmentation of YOLO, our power efficiency comparison is limited to our AIV-SoC and the 1080Ti, as presented in Table 7. Similar to Figure 22, Figure 24 visualizes the 0.8% mAP loss observed in instance segmentation when using an image size of 352, attributable to quantization.

E. ROBOFLOW UNIVERSE CRACK INSTANCE SEGMENTATION SIMULATION

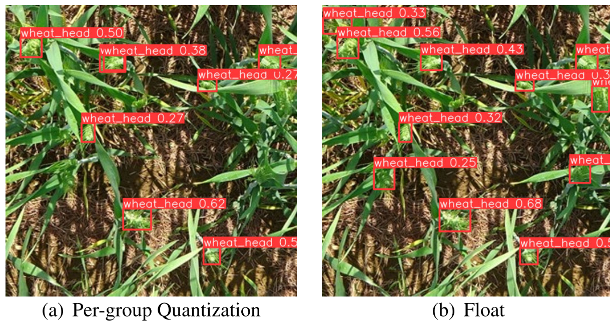
The Roboflow Universe Crack [43] Segmentation Dataset emerges as an extensive repository tailored specifically for professionals engaged in transportation and public safety

(a) Per-group Quantization

(b) Float

FIGURE 22. COCO objection detection mAP loss of quantization.

analyses, comprising a total of 4029 static images captured from diverse road and wall scenarios. Crack segmentation holds practical relevance in infrastructure upkeep, facilitating



(a) Per-group Quantization (b) Float

FIGURE 23. **Global Wheat 2000 objection detection mAP loss of quantization.**

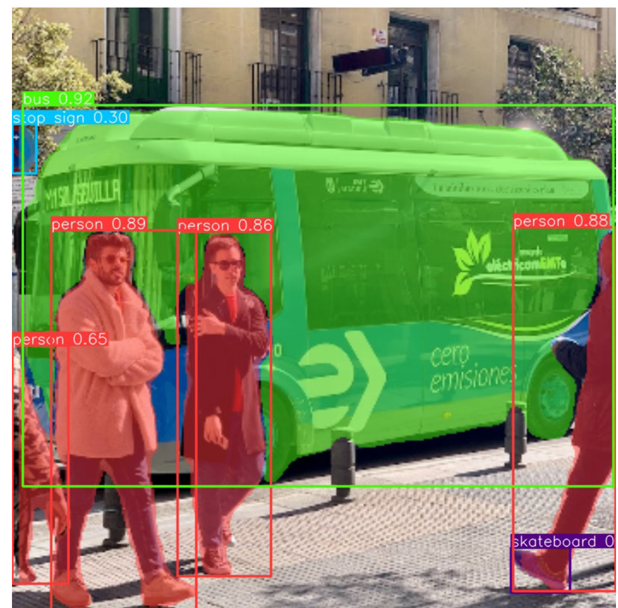
TABLE 10. **YOLOv8s instance segmentation mAP 0.5-0.95 in COCO.**

Image size	Quantization	mAP 0.5-0.95			
		all	small	medium	large
640	Float	36.8%	17.0%	41.2%	53.7%
	Per-channel	36.3%	16.9%	40.9%	53.3%
	Per-group	35.9%	16.0%	40.6%	52.9%
	Per-tensor	35.4%	16.1%	40.0%	52.6%
352	Float	30.3%	8.1%	32.6%	53.0%
	Per-channel	30.1%	8.0%	32.3%	53.1%
	Per-group	29.5%	7.8%	31.8%	52.2%
	Per-tensor	29.1%	7.7%	31.2%	51.8%
256	Float	24.8%	4.4%	24.3%	48.8%
	Per-channel	24.5%	4.1%	23.9%	48.5%
	Per-group	24.1%	3.9%	23.3%	48.2%
	Per-tensor	23.8%	3.9%	22.9%	47.2%

TABLE 11. **YOLOv8s instance segmentation mAP in Roboflow Universe Crack.**

Image size	Quantization	mAP 0.5-0.95	mAP 0.5
640	Float	22.7%	64.6%
	Per-channel	22.2%	62.2%
	Per-group	22%	64.1%
	Per-tensor	21.5%	63.5%
352	Float	25.6%	71.3%
	Per-channel	25.1%	69.7%
	Per-group	25.6%	73.4%
	Per-tensor	26.0%	73.5%
256	Float	25.0%	64.7%
	Per-channel	24.8%	64.5%
	Per-group	25.2%	66.5%
	Per-tensor	25.6%	66.0%

(a) Per-group Quantization



(b) Float

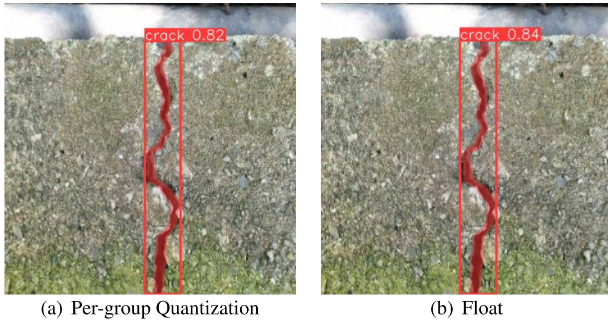
FIGURE 24. **COCO instance segmentation mAP loss of quantization.**

the detection and evaluation of structural deterioration. Moreover, it plays a pivotal role in bolstering road safety by empowering automated systems to identify and address pavement cracks promptly for timely repairs.

The number of classes (*nc*) for the Roboflow Universe Crack instance segmentation model is 1. We trained it using the same methodology employed for the Global Wheat 2000 dataset. Additionally, the overall inference time difference between COCO instance segmentation and our model is negligible. Furthermore, we present the mean Average Precision (mAP) table alongside a pair of example images. Due to the relatively small size of most images in the Roboflow Universe Crack dataset (often below a resolution of 416×416), employing an image resolution of 352 results in superior overall precision compared to using 640.

F. COCO POSE DETECTION SIMULATION

In a similar manner, we extended our investigation to YOLOv8 pose detection. We performed a comparative analysis involving four distinct quantization models, as portrayed in Table 12. The test dataset is COCO-Pose [20], a specialized version of the COCO dataset, designed for pose estimation tasks. Due to the absence of prior research dedicated to the pose detection of YOLO, our power efficiency analysis is singularly focused on a comparative evaluation between our



(a) Per-group Quantization (b) Float

FIGURE 25. Roboflow Crack Instance Segmentation mAP loss of quantization.

(a) Per-group Quantization

(b) Float

FIGURE 26. COCO Pose detection mAP loss of quantization.

AIV-SoC and the 1080Ti, as elucidated in Table 7. Consistent with Figure 22, Figure 26 illustrates the simulation outcomes for pose detection across 2 models at an image size of 352.

TABLE 12. YOLOv8s pose detection mAP 0.5-0.95 in COCO.

Image size	Quantization	mAP 0.5-0.95		
		all	medium	large
640	Float	59.9%	53.7%	70.3%
	Per-channel	58.7%	52.2%	69.1%
	Per-group	53.5%	45.2%	65.7%
	Per-tensor	48.9%	41.3%	60.1%
352	Float	47.4%	34.7%	65.5%
	Per-channel	45.9%	33.1%	64.3%
	Per-group	42.0%	29.9%	58.8%
	Per-tensor	36.7%	27.2%	50.7%
256	Float	37.0%	21.9%	58.2%
	Per-channel	36.0%	21.2%	57.1%
	Per-group	32.4%	19.4%	51.0%
	Per-tensor	28.0%	17.9%	42.7%

Notably, there is a certain degree of mAP loss (5.4%) in pose detection, which may be attributed to its unique key-point branch. The adoption of Int8 quantization results in a greater loss of information compared to other fully convolution networks (FCN). In our future work, we will explore the feasibility of employing per-group and per-channel hybrid quantization.

VI. CONCLUSION

Previous researches on RISC-V [25] were confined to providing support exclusively for convolution, pooling and fully connected layers. This limitation constrained its utility to elementary models like VGG and ResNet, restricting its applicability to a narrow spectrum of models and impeding its adaptability to contemporary trends in neural network evolution. Moreover, due to its exclusive consideration of image classification tasks, the maximum input resolution was confined to 256 × 256. Additionally, its SoC design also lacks the inclusion of a memory controller and UART module, rendering it impractical for debugging scenarios and real-world industrial applications.

Our prior conference proceeding [34] only support YOLOv8's objection detection. In this paper, we introduce AIV-SoC, which extends support to instance segmentation and pose detection. Moreover, we have improved the input image resolution from 256 × 256 to 352 × 352, while simultaneously reducing the AXI4 bus width from 256 to 64 bits. The achieved FPS for these tasks are 67.1, 55.2, and 64.9, respectively. Despite the mAP experiencing a reduction for smaller targets, constituting half of the original 640 × 640 size, the mAP for larger targets remains nearly unchanged from the original 640 × 640 size. The intermediate-sized accuracy loss is also limited to approximately 9%. It is noteworthy that the marginal benefits of further increasing the resolution are diminishing.

In contrast to alternative architectures, our RISC-V architecture demonstrates the capability to seamlessly execute various branches of YOLOv8 on-chip, encompassing the post-processing step. Unlike ASIC or FPGA architectures tailored for CNNs, which often accommodate only object detection, our design adeptly manages YOLOv8 Object

Detection, Instance Segmentation, and Pose Detection. This signifies a superior level of versatility in comparison to existing solutions. Additionally, we provide benchmark results for specific areas, a feature lacking in their designs. Furthermore, as illustrated in Table 7, our architecture exhibits outstanding power efficiency compared with other platforms.

While our architecture may not match the versatility of GPUs, its primary strength lies in adaptability and scalability, facilitated by the utilization of the C2RTL design toolkit and a modular RISC-V platform. These features empowers us to efficiently design and simulate the hardware. Reference [2] indicates that YOLO can be enhanced with Swin Transformer, thus augmenting the environmental perception capabilities of autonomous vehicles. Hence, we can further customize instructions to accommodate increasingly intricate neural networks in the future.

REFERENCES

[1] C. P. Papageorgiou, M. Oren, and T. Poggio, "A general framework for object detection," in *Proc. 6th Int. Conf. Comput. Vis.*, Jun. 1998, pp. 555–562.

[2] Y. Cao, C. Li, Y. Peng, and H. Ru, "MCS-YOLO: A multiscale object detection method for autonomous driving road environment recognition," *IEEE Access*, vol. 11, pp. 22342–22354, 2023.

[3] C. Yin, J. Tang, T. Yuan, Z. Xu, and Y. Wang, "Bridging the gap between semantic segmentation and instance segmentation," *IEEE Trans. Multimedia*, vol. 24, pp. 4183–4196, 2022.

[4] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2980–2988.

[5] H. Chen, K. Sun, Z. Tian, C. Shen, Y. Huang, and Y. Yan, "BlendMask: Top-down meets bottom-up for instance segmentation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 8570–8578.

[6] C. Cadena, A. R. Dick, and I. D. Reid, "Multi-modal auto-encoders as joint estimators for robotics scene understanding," in *IEEE Robot. Automat. Soc. (RAS)*, vol. 5. Ann Arbor, MI, USA: Univ. of Michigan, 2016.

[7] Y. Zhou, O. F. Onder, Q. Dou, E. Tsougenis, H. Chen, and P. A. Heng, "CIA-Net: Robust nuclei instance segmentation with contour-aware information aggregation," in *Proc. 26th Int. Conf. Inf. Process. Med. Imag.* Hong Kong: Springer, Jun. 27, 2019, pp. 682–693.

[8] S. Zhou, D. Nie, E. Adeli, J. Yin, J. Lian, and D. Shen, "High-resolution encoder–decoder networks for low-contrast medical image segmentation," *IEEE Trans. Image Process.*, vol. 29, pp. 461–475, 2020.

[9] H. A. Alhaija, S. K. Mustikovela, L. Mescheder, A. Geiger, and C. Rother, "Augmented reality meets deep learning for car instance segmentation in urban scenes," in *Proc. British Mach. Vis. Conf.*, 2017, vol. 1, no. 2, pp. 1–12.

[10] C.-W. Hsieh, C.-Y. Chen, C.-L. Chou, H.-H. Shuai, J. Liu, and W.-H. Cheng, "FashionOn: Semantic-guided image-based virtual try-on with detailed human and clothing information," in *Proc. 27th ACM Int. Conf. Multimedia*, Oct. 2019, pp. 275–283.

[11] H. Dong, X. Liang, X. Shen, B. Wang, H. Lai, J. Zhu, Z. Hu, and J. Yin, "Towards multi-pose guided virtual try-on network," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 9025–9034.

[12] C. Wang, Y. Wang, and A. L. Yuille, "An approach to pose-based action recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2013, pp. 915–922.

[13] J. Terven and D. Cordova-Esparza, "A comprehensive review of YOLO architectures in computer vision: From YOLOv1 to YOLOv8 and YOLO-NAS," 2023, *arXiv:2304.00501*.

[14] M. Sohan, T. S. Ram, R. Reddy, and C. Venkata, "A review on YOLOv8 and its advancements," in *Proc. Int. Conf. Data Intell. Cogn. Inform.*, Singapore. Springer, 2024, pp. 529–545.

[15] Y. Zhao, B. Chen, B. Liu, C. Yu, L. Wang, and S. Wang, "GRP-YOLOv5: An improved bearing defect detection algorithm based on YOLOv5," *Sensors*, vol. 23, no. 17, p. 7437, 2023.

[16] P. Yan, Q. Sun, N. Yin, L. Hua, S. Shang, and C. Zhang, "Detection of coal and gangue based on improved YOLOv5.1 which embedded scSE module," *Measurement*, vol. 188, Jan. 2022, Art. no. 110530.

[17] M. Qiu, L. Huang, and B.-H. Tang, "ASFF-YOLOv5: Multielement detection method for road traffic in UAV images based on multiscale feature fusion," *Remote Sens.*, vol. 14, no. 14, p. 3498, Jul. 2022.

[18] K. Wang, J. H. Liew, Y. Zou, D. Zhou, and J. Feng, "PANet: Few-shot image semantic segmentation with prototype alignment," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 9196–9205.

[19] *Ultralytics YOLOv8 Docs*. Accessed: Mar. 24, 2024. [Online]. Available: https://docs.ultralytics.com/

[20] T. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollšr, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.* Zurich, Switzerland: Springer, Sep. 2014, pp. 740–755.

[21] X. Wang, H. Gao, Z. Jia, and Z. Li, "BL-YOLOv8: An improved road defect detection model based on YOLOv8," *Sensors*, vol. 23, no. 20, p. 8361, Oct. 2023.

[22] X. Li, W. Wang, L. Wu, S. Chen, X. Hu, J. Li, J. Tang, and J. Yang, "Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, vol. 33, no. 2020, pp. 21002–21012.

[23] X. Li, C. Lv, W. Wang, G. Li, L. Yang, and J. Yang, "Generalized focal loss: Towards efficient representation learning for dense object detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 3, pp. 3139–3153, Mar. 2023.

[24] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," Dept. EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-54, 2014.

[25] H. Wang, D. Li, and T. Isshiki, "Reconfigurable CNN accelerator embedded in instruction extended RISC-V core," in *Proc. 6th Int. Conf. Electron. Technol. (ICET)*, May 2023, pp. 945–954.

[26] T. Sadasue and T. Isshiki, "LLVM-C2RTL: C/C++ based system level RTL design framework using LLVM compiler infrastructure," *IPSJ Trans. Syst. LSI Design Methodol.*, vol. 16, pp. 12–26, Jan. 2023.

[27] A. Hosseiny and H. Jahanirad, "Hardware acceleration of YOLOv7-tiny using high-level synthesis tools," *J. Real-Time Image Process.*, vol. 20, no. 4, p. 75, Aug. 2023.

[28] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[29] F. Liu, B. Zhang, G. Chen, G. Gong, H. Lu, and W. Li, "A novel configurable high-precision and low-cost circuit design of sigmoid and tanh activation function," in *Proc. IEEE Int. Conf. Integr. Circuits, Technol. Appl. (ICTA)*, Nov. 2021, pp. 222–223.

[30] G. Lin and W. Shen, "Research on convolutional neural network based on improved relu piecewise activation function," *Proc. Comput. Sci.*, vol. 131, pp. 977–984, Jan. 2018.

[31] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2015, pp. 161–170.

[32] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

[33] Z. Wang, K. Xu, S. Wu, L. Liu, L. Liu, and D. Wang, "Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2," *IEEE Access*, vol. 8, pp. 116569–116585, 2020.

[34] H. Wang, D. Li, and T. Isshiki, "A power-efficient end-to-end implementation of YOLOv8 based on RISC-V," in *Proc. 4th Int. Conf. Comput. Artif. Intell. Technol. (CAIT)*, Dec. 2023, pp. 217–225, doi: 10.1109/cait59945.2023.10469637.

[35] A. Litvin, K. Nasrollahi, S. Escalera, C. Ozcinar, T. B. Moeslund, and G. Anbarjafari, "A novel deep network architecture for reconstructing RGB facial images from thermal for face recognition," *Multimedia Tools Appl.*, vol. 78, no. 18, pp. 25259–25271, Sep. 2019.

[36] K.-W. Chang and T.-S. Chang, "Efficient accelerator for dilated and transposed convolution with decomposition," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[37] X. Di, H.-G. Yang, Y. Jia, Z. Huang, and N. Mao, "Exploring efficient acceleration architecture for winograd-transformed transposed convolution of GANs on FPGAs," *Electronics*, vol. 9, no. 2, p. 286, Feb. 2020.

[38] A. Odena, V. Dumoulin, and C. Olah, "Deconvolution and checkerboard artifacts," *Distill*, vol. 1, no. 10, p. e3, Oct. 2016.
[39] S. Li, C. Yu, T. Xie, and W. Feng, "A power-efficient optimizing framework FPGA accelerator for YOLO," in *Proc. 15th Int. Congr. Image Signal Process., BioMed. Eng. Inform.*, Nov. 2022, pp. 1–6.
[40] P. Bader, S. Blanes, and F. Casas, "Computing the matrix exponential with an optimized Taylor polynomial approximation," *Mathematics*, vol. 7, no. 12, p. 1174, Dec. 2019.
[41] *Canaan, Kendryte K230 Docs*. Accessed: Mar. 31, 2024. [Online]. Available: https://developer.canaan-creative.com/
[42] E. David, S. Madec, P. Sadeghi-Tehran, H. Aasen, B. Zheng, S. Liu, N. Kirchgessner, G. Ishikawa, K. Nagasawa, M. A. Badhon, C. Pozniak, B. de Solan, A. Hund, S. C. Chapman, F. Baret, I. Stavness, and W. Guo, "Global wheat head detection (GWHD) dataset: A large and diverse dataset of high-resolution RGB-labelled images to develop and benchmark wheat head detection methods," *Plant Phenomics*, vol. 2020, pp. 1–12, Jan. 2020.
[43] Roboflow Universe. (2022). *Crack Dataset*. Accessed: Mar. 24, 2024. [Online]. Available: https://universe.roboflow.com/university-bswxt/crack-bphdr/

HANSEN WANG received the B.E. degree in information science and electronic engineering from Zhejiang University, China, in 2019, and the M.E. degree in information and communications engineering from Tokyo Institute of Technology, Japan, in 2021, where he is currently pursuing the Ph.D. degree with the Department of Information and Communications Engineering. His research interests include RISC-V architectures, application-specific processors, artificial intelligence visual systems, and deep neural networks.

DONGJU LI (Member, IEEE) received the Ph.D. degree in electrical and electronics from Tokyo Institute of Technology, in 1998. Currently, she is an Assistant Professor with the Department of Information and Communications Engineering, Tokyo Institute of Technology. Her research interests include embedded systems and solutions for fingerprint authentication and architecture design for system-on-chip. She has been a member of IEEE CAS and IEICE, since 1998.

TSUYOSHI ISSHIKI (Member, IEEE) received the B.E. and M.E. degrees in electrical and electronics engineering from Tokyo Institute of Technology, in 1990 and 1992, respectively, and the Ph.D. degree in computer engineering from the University of California, Santa Cruz, in 1996. He is currently a Professor with the Department of Information and Communications Engineering, Tokyo Institute of Technology. His research interests include design methodologies for system-on-chip architectures, application-specific processors, image processing systems, MPSoC, and fingerprint authentication algorithms. He is a member of IEEE CAS, IPSJ, and IEICE.