POLITECNICO DI TORINO
Repository ISTITUZIONALE

Accelerating Mixed-Precision QNN Inference on RISC-V MCUs With the STAR-MAC Unit

Original
Accelerating Mixed-Precision QNN Inference on RISC-V MCUs With the STAR-MAC Unit / Manca, Edward; Urbinati, Luca; Casu, Mario R.. - In: IEEE ACCESS. - ISSN 2169-3536. - 13:(2025), pp. 208533-208548. [10.1109/ACCESS.2025.3641382]

Availability:
This version is available at: 11583/3006065 since: 2025-12-20T18:08:33Z

Publisher:
IEEE

Published
DOI:10.1109/ACCESS.2025.3641382

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

26 April 2026

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 21 November 2025, accepted 3 December 2025, date of publication 8 December 2025,
date of current version 12 December 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3641382

RESEARCH ARTICLE

Accelerating Mixed-Precision QNN Inference on RISC-V MCUs With the STAR-MAC Unit

EDWARD MANCA[1], (Graduate Student Member, IEEE), LUCA URBINATI[2], (Member, IEEE), AND MARIO R. CASU[1], (Senior Member, IEEE)

[1]Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
[2]Institute of Electronics, Information Engineering and Telecommunications, National Research Council of Italy (CNR-IEIIT), 40136 Bologna, Italy

Corresponding author: Edward Manca (edward.manca@polito.it)

ABSTRACT The demand for efficient edge machine learning (ML) in Internet of Things (IoT) applications has driven interest in Microcontroller Unit (MCU)-based TinyML solutions, especially with the rise of RISC-V. Although MCUs are power efficient, their limited resources challenge the deployment of complex ML models. Mixed-Precision Quantization (MPQ) offers the best trade-off between model size, energy consumption, and accuracy, by using different weights and activations precision across model layers. Recently, an increasing interest in hardware support for Mixed-Precision Quantized Neural-Networks (MP-QNN) in MCU-class processors has emerged. In this paper, we present a novel precision-scalable Multiply-and-Accumulate (MAC) unit, named *STAR-MAC*, that supports MPQ on 16-, 8- and 4-bit integers. Combining Sum-Together and Sum-Apart subword-parallel multiplications in a reconfigurable architecture, STAR-MAC accelerates Fully-Connected, 2D-Convolution, and Depth-wise layers. We integrate STAR-MAC into the low-power RISC-V Ibex core and validate it on the four MLPerf Tiny MP-QNN models trained with QKeras and using a modified TensorFlow Lite for Microcontrollers (TFLM) enhanced with MPQ kernels, that we open-source. Compared to the 8-bit standard TFLM runtime, all this combined hardware-software approach results in a FlatBuffer smaller by 27%, an average latency reduction of 68% across the four MLPerf Tiny models—measured on a Field-Programmable Gate Array(FPGA)-based System-on-Chip setup—, and a negligible accuracy loss on the corresponding test sets. Synthesis on 28-nm CMOS technology shows that our STAR-based Ibex has the highest energy efficiency per unit of Silicon area (175.7 GOPS/W/mm^2 at 4-bit) among various MPQ processors, with limited overhead over the original Ibex (+12.2% area, +7.3% power). Our solution enables low-power, low-latency inference of MP-QNNs on IoT nodes.

INDEX TERMS Mixed-precision quantization, precision-scalable mac unit, quantized neural networks, RISC-V, tensorflow lite for microcontrollers, TinyML.

I. INTRODUCTION

The proliferation of Internet of Things (IoT) devices and the increasing demand for real-time data analysis have driven the rise of TinyML, machine learning (ML) applications targeting edge devices. Low-power and compact Microcontroller Units (MCUs) are often the preferred choice for running TinyML applications. However, their limited processing power and memory hinder the deployment of complex ML models.

Quantization, which reduces the precision of the weights and activations of neural networks, is one of the most powerful optimization techniques to address these challenges. However, standard quantization methods often lead to accuracy degradation, which is unacceptable for TinyML tasks requiring precise data interpretation. Mixed-precision quantization (MPQ), on the other hand, offers a more flexible approach by allowing different levels of precision for different parts of a Quantized Neural Network (QNN) [1]. An application may use higher precision (e.g., 16 bits) for the most sensitive layers—those that affect accuracy the most—and lower precision for others (e.g., 4 bits) to

The associate editor coordinating the review of this manuscript and approving it for publication was Zhengmao Li.

© 2025 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/

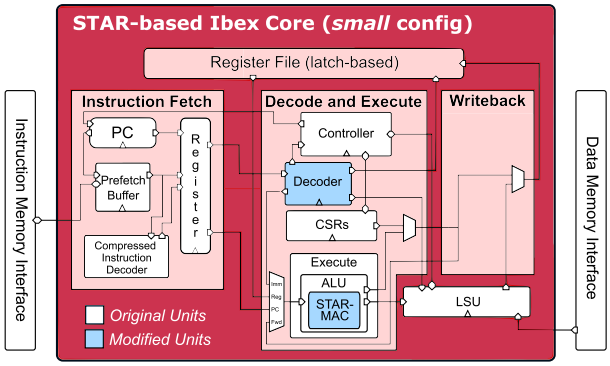

FIGURE 1. Ibex core in its *small* configuration [5] with our STAR-MAC unit instead of the original 17-bit *Fast* multiplier.

maintain low latency and power consumption. It may also trade accuracy for higher performance or minimal power usage. This flexibility is essential for TinyML applications on resource-constrained devices.

Embedded processors are the typical target for edge applications, but often lack hardware support for MPQ. This can lead to increased computational overhead, since MPQ operations must be implemented in software (e.g., with 32-bit multipliers and shift-and-mask operations), potentially negating the advantages of MPQ. Motivated by these considerations, our research focuses on designing a novel Precision-Scalable (PS) Multiply-and-Accumulate (MAC) unit for Mixed-Precision (MP) hardware acceleration, and integrating it into a low-power RISC-V core.

The proposed PS-MAC unit, called *STAR-MAC*, is based on STAR, a Sum-Together/Apart Reconfigurable 16-bit Baugh-Wooley (BW) multiplier that can be configured to operate in *Sum-Together (ST)* or in *Sum-Apart (SA)* mode [2]. In SA mode, it behaves as a single-instruction multiple-data (SIMD) unit that multiplies low-precision operands in parallel and maintain them separate in the output, while in ST mode it acts as a dot-product unit as it sums the low-precision products internally and generates a single output. While in prior literature SA and ST multipliers have typically been treated as alternative implementations [2], STAR combines ST and SA subword-parallel (SWP) multiplication techniques into a single hardware multiplier solution. The rationale behind this unification is that ST operations are better suited for PS workloads in Fully-Connected (FC) and 2D Convolution (2D-Conv) layers, while SA operations are more effective for Depth-wise Convolution (DW-Conv), commonly used in DW-Separable Convolutions [3], [4].

We integrate STAR-MAC in the Ibex core, an open-source RISC-V MCU for resource-constrained applications [5], resulting in the proposed *STAR-based Ibex* outlined in Fig. 1. For compatibility with the *Fast* multiplier of the original Ibex, we design STAR-MAC to produce either one multiplication with two 32-bit operands in three clock cycles, or two/four/eight parallel multiplications (in SA mode) or dot products (in ST mode) with four/eight/sixteen parallel 16-/8-/4-bit operands in two cycles. To fully take advantage of hardware-supported MPQ, a complete flow from training to deployment of Mixed-Precision QNNs (MP-QNNs) is highly desirable [6], [7], [8]. Hence, to ease the deployment of TinyML MP-QNN models on the STAR-based Ibex, we decide to leverage well-established tools: QKeras and AutoQKeras to train QNNs [9], and Google's TensorFlow Lite for Microcontrollers (TFLM) for MCU deployment, a state-of-the-art (SoA) lightweight inference engine recently renamed to *LiteRT for Microcontrollers*. However, the current deployment flow of TFLM relies on TensorFlow-Lite (TF-Lite), Google's high-performance runtime for on-device AI, which supports only 8 bits for both weights and activations. Therefore, we introduce *STAR-TFLM*, a modified version of TFLM which extends the original TFLM with new low-level MPQ kernels for 2D-Conv, DW-Conv, and FC layers, supporting any combination of 16-, 8-, and 4-bit data types for features and weights. We also introduce *QKeras-to-TFLM converter*, which is a method for converting an MP-QNN QKeras model into a TF-Lite *FlatBuffer*, a compact binary file that contains the architecture and parameters of Deep Neural Network (DNN) models.

In this paper, we integrate STAR-TFLM and QKeras-to-TFLM into a complete flow from MP-QNN QKeras training to MCU TFLM deployment. We then use this flow to validate our STAR-based Ibex processor by running the four MLPerf Tiny models of the MLPerf Tiny Benchmark [10] within a System-on-Chip (SoC) generated by Embedded Scalable Platform (ESP) [11], a framework that supports execution, debugging, and performance estimation on a Field-Programmable Gate Array (FPGA).

The results show an average reduction of the inference clock cycles across the four MLPerf Tiny MP-QNNs by 59% (and up to 68%) compared to the same models quantized on 8 bits and deployed with the standard TFLM. To showcase our solution's applicability for resource-efficient MCU deployment, we also report standard quality metrics—Power, Performance, and Area (PPA)—for Application-Specific Integrated Circuits (ASIC) implementation. As we explain in Sec. VII-F, Synthesis results on a 28-nm CMOS technology at 250 MHz show that our STAR-based Ibex outperforms other SoA MPQ processors in energy efficiency per unit area by at least 12.8%, achieving a peak of 175.7 GOPS/W/mm^2 at 4-bit precision, while incurring only modest area and power overheads (+12.2% and +7.3%, respectively) compared to the original Ibex core [5].

The remaining sections of this manuscript are seven. Sec. II reviews related work. Sec. III details the STAR-MAC unit and its integration in the Ibex processor. Secs. IV–VI outline the deployment flow, introduce the QKeras-to-TFLM converter, and report the TFLM modifications and custom kernels leveraging STAR instructions to enable MP-QNN execution, respectively. Sec. VII presents results on MP-QNNs MLPerf Tiny models, along with a comparison with SoA MPQ processors. Finally, Sec. VIII draw some conclusions.

All of our code is available on GitHub [12] to promote MP-QNN development on MCU-class devices.

II. RELATED WORK

A. MCU-CLASS PROCESSORS WITH HARDWARE SUPPORT FOR MPQ

The integration of specialized PS units to support quantized data with variable precision in low-power MCU-class processors has attracted significant attention in recent years.

The Divide-and-Conquer (D&C) approach uses low-precision multipliers that can be dynamically combined via shift-and-add logic to form higher-precision multipliers. The D&C MAC unit of [13] was designed for 32-bit RISC-V MCUs and supports operands from 2 to 32 bits, combining 256 2-bit low-precision multipliers. However, the authors did not actually present a processor integration. Our work is more complete, as it also explores the integration of our SWP MAC unit into the Ibex. Moreover, based on the comparison of PS multiplier architectures reported in [3] and [4], the D&C approach is suboptimal with respect to SWP at the target clock frequency considered in our work, i.e., 250 MHz.

Bit-Serial methods process operands serially, performing multiplications via add-shift operations that take a number of clock cycles proportional to the operands bit-width. The authors of BISDU [14] proposed a multiplier-less dot-product unit for bit-serial matrix multiplications. Their approach introduces a hardware extension made of a population-count unit and 2:1 multiplexers, which leverages the conventional arithmetic logic unit (ALU) resources (and, xor, xnor) of the RISC-V Instruction Set Architecture (ISA). This design supports binary, ternary, and MPQ. They integrated it into TinyRocket, a 32-bit configurable RISC-V core [15]. According to the authors' experiments, while this solution is competitive for operations involving operands with 1 to 3 bits, it is dominated by non-bit-serial solutions at 4 bits and above.

An alternative research area in specialized ML hardware explores reconfigurable multipliers and MAC units based on the PS concept. Notable contributions include [2], which introduced the SA and ST paradigms in BW multipliers; [16], which reviewed and benchmarks SoA PS MAC architectures; and [3], which compared PS multipliers in terms of PPA.

Recently, we introduced for the first time the concept of STAR as the combination of the SA and ST modes in a single multiplier structure in [4]. Specifically, we proposed and evaluated four different STAR architectures (based on PS multipliers proposed in [2], [17], [18]) to identify Pareto-optimal designs in terms of PPA for a given target clock frequency. Furthermore, we thoroughly described the microarchitecture of one of them, the so-called *STAR SWP (BW)*, which is the STAR multiplier at the basis of the STAR-MAC unit presented in this work.

Another relevant contribution on MPQ and the Ibex core is presented in [19]. In the original Ibex [5], designers choose at synthesis time between two alternative 32-bit multiplier implementations. One is called *Single-cycle* and is made of three 17-bit multipliers, and one is called *Fast* and uses a single 17-bit multiplier to compute 32-bit multiplications in 3 or 4 clock cycles. In [19], the authors extended the single-cycle implementation with a fourth 17-bit multiplier, resulting in significant hardware overhead. In contrast, we replace the *Fast* multiplier with our lightweight 16-bit STAR-MAC. Furthermore, their design supports 8-, 4-, and 2-bit weights, but only 8-bit activations, whereas ours supports symmetric 16-, 8-, 4-bit weights and activations. Finally, to preserve correctness, their design inserts guard bits between low-precision input operands and multiplier outputs, resulting in reduced hardware efficiency. Differently, our STAR-MAC unit avoids this overhead by natively supporting SWP operations.

In XPulpNN [20] and Dustin [21], the authors extended the dot-product unit of their earlier work [18] by supporting MP formats down to 2-bit precision and including symmetric and asymmetric configurations, respectively, through parallel multipliers with varying bitwidths and an adder tree for partial sum reduction. In contrast, our processor employs STAR, which supports symmetric multiplications not only in ST mode but also in SA mode—an essential feature for operations such as DW-Conv—and achieves higher area and power efficiency at sub-GHz frequencies, as proven in [4].

In this work, we did not consider SoCs with tightly-coupled MPQ accelerators, such as in [22], since we focused solely on the design and optimization of a PS MAC unit and its integration in an MCU with limited area overhead. A more complete literature review on MPQ hardware solutions is presented in [23].

B. OPTIMIZED INFERENCE LIBRARIES AND ENGINES FOR MPQ

To simplify DNN deployment, inference libraries offer kernels (usually C/C++ functions) for various layers that are optimized for specific hardware platforms and compiled alongside the user's code. The first three rows of Table 1 show the SoA inference libraries that support MP.

Cortex Microcontroller Software Interface Standard-NN (CMSIS-NN), developed by ARM [24] for Arm Cortex-M MCUs, is the most widely adopted library due to the company's widespread prevalence in the MCU market. It provides various kernels for 2D-Conv, DW-Conv, and FC using the INT16 and INT8 integer data type, with *symmetric* precision, meaning that weight/activation operands must have the same number of bits (i.e., 16/16, 8/8). It also supports kernels with *asymmetric* precision, where weight and activation operands can have different bit-widths. In particular, it supports two asymmetric combinations following the TFLM specifications [34], i.e., 16/8 and 8/4 [35]. In contrast to this approach, we also support the 4/4 precision for weights/activations.

Recently, other libraries have emerged to address the demanding requirement of precision scalability. CMix-NN [25] is based on CMSIS-NN, but offers additional kernels using asymmetric precisions on INT8, INT4, and INT2 datatypes, always targeting Arm Cortex-M cores.

PULP-NN implements INT8 down to INT1 symmetric kernels in the original version [27], and asymmetric INT8, INT4, and INT2 kernels in the extended one [28]. Although still derived from CMSIS-NN, kernels are optimized for

TABLE 1. State-of-the-Art optimized inference libraries and inference engines with MP support and processors as target platforms.

Name [paper]	Type	Supported Models	Supported Precisions (w: weights a: activations)	Based on	MPQ Search Algorithm	Supported Input Model Formats	Target Hardware Platforms
CMSIS-NN by Arm [24]	Optimized inference library	Matrix Multiplication, 2D/DW-Conv, FC	INT16/8 for w and a, symmetric; INT 16/8, 8/4 for w/a, asymmetric	—	—	—	Arm Cortex-M
CMix-NN by Capotondi et al. [25]	Optimized inference library	2D/DW-Conv, FC	INT8/4/2 for w and a, asymmetric (any combination)	CMSIS-NN	Iterative algorithms based on memory constraints [26]	PyTorch	Armv7-M ISA (Arm Cortex-M4/M7)
PULP-NN by Garofalo et al. [27] and Bruschi et al. [28]	Optimized inference library	2D/DW-Conv, FC	INT8 down to INT1 for w and a, symmetric [27]; In the extension from [28], also INT8/4/2 for w and a, asymmetric (any combination)	CMSIS-NN	—	—	PULP clusters of RISC-V based processors, e.g., the Ri5cy-based PULP cluster [29]
STM32-Cube.AI (X-Cube-AI) by STMicroelectronics [30]	Optimized inference library and inference engine	For 8-bit or FP32: subset of Keras, TF-Lite, ONNX layers[1]; For $<$ 8-bit: 2D/DW-Conv, FC; Classical ML models (k-means, SVM, RF, kNN, DT)[2]	For 8-bit or FP32: UINT8, INT8 or FP32 for layers in and out; UINT8/UINT8, INT8/INT8 or INT8/UINT8 for w/a; For $<$ 8-bit: many combinations of FP32, 8-bit signed fixed-point, binary signed (1-bit) for w and a, depending on the layer [31]	STM-32-Cube.AI runtime (based on CMSIS-NN) or TFLM runtime	Not present. The tool works with pre-trained FP or quantized models.	For 8-bit or FP32: ONNX, TF-Lite, PyTorch[3], TensorFlow[4], Keras, Lasagne, MATLAB, Caffe, ConvnetJS, Microsoft Cognitive Toolkit; For $<$ 8-bit: QKeras, Larq	STM32 Arm Cortex-M
N2D2 by CEA-List [32]	Training, optimization and deployment engine	2D/DW-Conv, FC and others[5,6]	INT8 down to binary, asymmetric	—	The user has to specify the precision of each layer manually.	ONNX, PyTorch[3], TensorFlow[3], Keras[3], N2D2 "INI" config. file	Plain or optimized C code for different targets: MCUs, CPUs, DSPs, GPUs and FPGAs[7]
This work	Optimized inference library; Training, optimization and deployment engine	For all supported precisions: STAR-based 2D/DW-Conv, FC; For 8-bit: all TFLM layers [33]	INT16/8/4 for w and a, asymmetric (any combination)	TFLM runtime	Bayesian Optimization (AutoQKeras)	QKeras, TensorFlow[8], Keras[8]	For supported precisions: STAR-based RISC-V Ibex or any 32-bit RISC-V by using emulated STAR instructions[9]; For 8-bit: standard TFLM

[1] STMicroelectronics, *X-CUBE-AI Keras Toolbox Support* (Documentation/supported_ops_keras.html in X-CUBE-AI Expansion Package), accessed on: Jun 16, 2025. [2] Quantization is supported for DNN models, but not for classical ML models. [3] Prior conversion to ONNX. [4] Prior conversion to TF-Lite or Keras h5 file. [5] CEA-List, *N2D2 - Quantization-Aware Training - Layer compatibility table*, https://cea-list.github.io/N2D2-docs/quant/qat.html, accessed on: Jun 16, 2025. [6] CEA-List, *N2D2 - Legacy - Layer compatibility table*, https://cea-list.github.io/N2D2-docs/export/legacy.html, accessed on: Jun 16, 2025. [7] Plain C++ code, C code for STM32 (Arm Cortex-M4/M7), C code with OpenMP or TensorRT, high-level synthesis C code for Xilinx FPGAs, C code using OpenCL for CPU/DSP or GPU, C code using Cuda or cuDNN for NVIDIA GPUs, C code for DNeuro and PNeuro (CEA-List's architectures for ASICs and FPGAs, resp.). [8] Prior conversion to QKeras. [9] For emulated STAR instructions, see Sec. VII-B.

PULP clusters of RISC-V processors, such as those based on Ri5cy [29].

Several tools for the automated conversion of DNN models from development frameworks to MCU-ready code for inference have emerged in recent years [6], [7], [8]. Our literature survey indicates that most of these tools support quantized 8-bit or 32-bit models, with only a few accommodating other data-type precisions for weights and activations, as reported in the fourth and fifth rows of Table 1.

X-Cube-AI by STMicroelectronics [30] generates optimized C code for STM32 Arm Cortex-M processors given a pre-trained Floating-Point (FP) or quantized model, e.g., ONNX or TF-Lite format. The runtime library is partially based on CMSIS-NN, although users can also deploy the network using TFLM. X-Cube-AI supports MP-QNN models with different combinations of FP32, 8-bit, and 1-bit data types for weights and features, depending on the type of layer [31]. In the latter case, the models should be pre-trained with QKeras or Larq [30], [36]. Unlike X-CUBE-AI, our solution also supports MP training using AutoQKeras. Moreover, although X-CUBE-AI supports QKeras, it restricts MP-QNNs to fixed-point formats of either 8 bits or 1 bit, without scaling factors for weights and activations. In contrast, our flow accommodates QKeras MP-QNNs with 16-, 8-, and 4-bit integer datatypes, including scaling factors. Finally, unlike our work and all the others in Table 1, X-CUBE-AI is not open-source.

Neural Network Design & Deployment (N2D2) by CEA-List [32] offers a complete solution to design, train, and optimize DNN models. It allows users to manually specify the precision of each DNN layer, from INT8 to INT1, and supports quantization-aware training. When targeting MCUs, it only generates plain C code or uses X-CUBE-AI for STM32 processors. In contrast, we target our custom STAR-MAC instructions tailored for MP-QNNs. Although N2D2 supports MPQ training, it lacks an automated precision search. Conversely, our deployment flow addresses this through the AutoQKeras' Bayesian Optimization engine.

In summary, our work integrates seamlessly with the TensorFlow toolchain, providing a deployment path to MCU-class processors that supports quantized layers with any combination of 16, 8, or 4 integer bits. This offers greater flexibility compared to the sole precision currently supported by TFLM, which is 8 bit. Our solution serves both as an

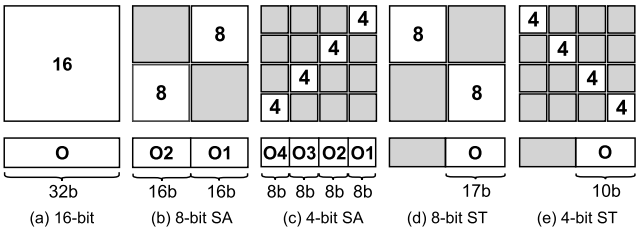

FIGURE 2. Working principle of the STAR multiplier [4]. The PPM is shown as the top square and the multiplier output as the bottom rectangle. When the operand precision is reduced, some regions of the PPM become inactive (grey), while the active ones (white) operate as smaller bit-width BW multipliers, receiving operands with a number of bits corresponding to the value indicated inside the PPM. In SA mode, STAR outputs multiple low-precision products in parallel; in ST mode, it performs dot products.

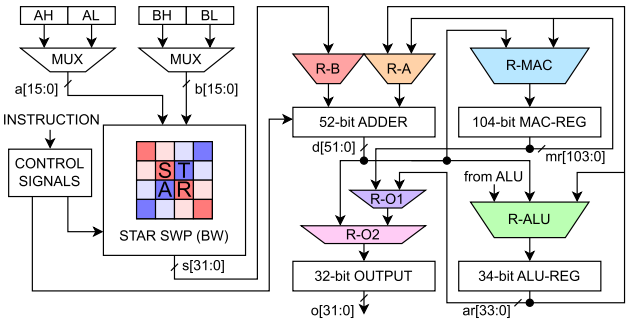

FIGURE 3. High-level view of the STAR-MAC unit.

TABLE 2. Operating modes of our STAR SWP (BW) multiplier [4]. [m:n]indicates the bit range of an operand or output.

Operating Mode	Mathematical Description
SA16(a,b) ST16(a,b)	$s_{[31:0]} = a_{[15:0]} \times b_{[15:0]}$
ST8(a,b)	$s_{[24:8]} = a_{[7:0]} \times b_{[15:8]} + a_{[15:8]} \times b_{[7:0]}$
ST4(a,b)	$s_{[21:12]} = a_{[3:0]} \times b_{[15:12]} + a_{[7:4]} \times b_{[11:8]} + a_{[11:8]} \times b_{[7:4]} + a_{[15:12]} \times b_{[3:0]}$
SA8(a,b)	$s_{[31:16]} = a_{[15:8]} \times b_{[15:8]}$ $s_{[15:0]} = a_{[7:0]} \times b_{[7:0]}$
SA4(a,b)	$s_{[31:24]} = a_{[15:12]} \times b_{[15:12]}$ $s_{[23:16]} = a_{[11:8]} \times b_{[11:8]}$ $s_{[15:8]} = a_{[7:4]} \times b_{[7:4]}$ $s_{[7:0]} = a_{[3:0]} \times b_{[3:0]}$

"optimized inference library", offering MP TFLM kernels for 2D-Conv, DW-Conv, and FC layers that support STAR-MAC instructions, and as a "training, optimization, and deployment engine", being based on QKeras, AutoQKeras, and TFLM, respectively.

III. STAR-MAC UNIT AND STAR-BASED IBEX

We begin with a short explanation of the STAR multiplier and continue with a background on Ibex. We then outline the STAR-MAC architecture and design choices, concluding with synthesis results on a 28-nm commercial technology.

A. STAR: BACKGROUND

Fig. 2 illustrates the operating principle of the 16-bit BW STAR multiplier [4] at different operand precisions. The top square represents the Partial Product Matrix (PPM) [37], while the bottom rectangle corresponds to the multiplier's output. For 16-bit operands (Fig. 2a), the PPM is fully active (white) and STAR behaves as a standard 16-bit BW multiplier. As the bitwidth of the operands decreases, some regions of the PPM become inactive (grey), while the remaining active blocks (white) start to behave as smaller PPMs, i.e., smaller BW multipliers with operand size indicated by the number inside the PPM itself. In SA mode, STAR keeps the results of multiple parallel low-precision multiplications separated in the final result, as shown for the case of 8-bit (Fig. 2b) and 4-bit (Fig. 2c) operands. In ST mode, STAR internally sums the results of these parallel multiplications, acting like a dot-product unit, producing a single result, as shown in Fig. 2d for the 8-bit case, and in Fig. 2e for the 4-bit one.

B. STAR-MAC INTEGRATION IN THE IBEX CORE

The open-source RISC-V Ibex core [5] supports the RV32I ISA with M, C, B, and E extensions [38] and offers different multiplier architectures. In this work, we use the *small*, two-stage pipeline version, with the *Fast* multiplier. Thanks to the RISC-V ISA, Ibex can be extended with custom instructions to accelerate domain-specific applications like ML workloads. Fig. 1 is an overview of the architecture of our STAR-based Ibex. We replace the original 3-cycle multiplier inside the ALU with our STAR-MAC unit, adding the minimum glue-logic required to interface with the existing signals. Its impact on area and power will be evaluated in Sec. VII-F.

C. STAR-MAC UNIT: OVERVIEW

The foundation of our new STAR-MAC unit is the STAR SWP (BW) multiplier introduced in [4]. An overview of the operations supported by this multiplier is reported in Table 2. Among the various multiplier architectures that we analyzed in [4], we select STAR SWP (BW) because it is Pareto-optimal in terms of area and power at our target frequency of 250 MHz.

A high-level diagram of the STAR-MAC architecture is presented in Fig. 3. The STAR SWP (BW) multiplier receives 16-bit sub-operands from registers A and B, which are the two source registers for a given RISC-V instruction, along with the control signals from the RISC-V decoder. These signals determine the 16-, 8-, or 4-bit operations and decide between SA or ST according to Table 2, and select which chunk of the MAC register (MAC-REG) to return according to Table 3.

The output of STAR, $s_{[31:0]}$ in Fig. 3, goes to one input of the 52-bit adder through a dedicated routing logic called R-B. This adder input is reserved to receive the multiplier's output. The second adder input comes from the ALU register

TABLE 3. STAR-MAC unit operations using the STAR operating modes of Table 2. $mr_{[103:0]}$ corresponds to the MAC register (MAC-REG in Fig. 3).

Operation	Description
mac16st	$mr_{[47:0]}$ += ST16($A_{[15:0]}$, $B_{[15:0]}$) + ST16($A_{[31:16]}$, $B_{[31:16]}$)
mac8st	$mr_{[31:0]}$ += ST8($A_{[15:0]}$, $B_{[15:0]}$) + ST8($A_{[31:16]}$, $B_{[31:16]}$)
mac4st	$mr_{[23:0]}$ += ST4($A_{[15:0]}$, $B_{[15:0]}$) + ST4($A_{[31:16]}$, $B_{[31:16]}$)
mac16sa	$mr_{[36:0]}$ += SA16($A_{[15:0]}$, $B_{[15:0]}$) $mr_{[88:52]}$ += SA16($A_{[31:16]}$, $B_{[31:16]}$)
mac8sa	$mr_{[46:42,15:0]}$ += SA8($A_{[15:0]}$, $B_{[15:0]}$)$_{[15:0]}$ $mr_{[36:16]}$ += SA8($A_{[15:0]}$, $B_{[15:0]}$)$_{[31:16]}$ $mr_{[98:94,67:52]}$ += SA8($A_{[31:16]}$, $B_{[31:16]}$)$_{[15:0]}$ $mr_{[88:68]}$ += SA8($A_{[31:16]}$, $B_{[31:16]}$)$_{[31:16]}$
mac4sa	$mr_{[51:47,7:0]}$ += SA4($A_{[15:0]}$, $B_{[15:0]}$)$_{[7:0]}$ $mr_{[46:42,15:8]}$ += SA4($A_{[15:0]}$, $B_{[15:0]}$)$_{[15:8]}$ $mr_{[41:37,23:16]}$ += SA4($A_{[15:0]}$, $B_{[15:0]}$)$_{[23:16]}$ $mr_{[36:24]}$ += SA4($A_{[15:0]}$, $B_{[15:0]}$)$_{[31:24]}$ $mr_{[103:99,59:52]}$ += SA4($A_{[31:16]}$, $B_{[31:16]}$)$_{[7:0]}$ $mr_{[98:94,67:60]}$ += SA4($A_{[31:16]}$, $B_{[31:16]}$)$_{[15:8]}$ $mr_{[93:89,75:68]}$ += SA4($A_{[31:16]}$, $B_{[31:16]}$)$_{[23:16]}$ $mr_{[88:76]}$ += SA4($A_{[31:16]}$, $B_{[31:16]}$)$_{[31:24]}$

(ALU-REG) signal $ar_{[33:0]}$, or from the MAC-REG signal $mr_{[103:0]}$, through R-A. The original Ibex multiplier already uses the 34 least significant bits of ALU-REG to perform multiplication instructions that take 3 or 4 cycles to complete. We reuse that connection and insert MAC-REG to store the accumulated values produced by STAR instructions. The role of these two registers will be further discussed in Sec. III-D2. The adder output $d_{[51:0]}$ is routed back to MAC-REG, through R-MAC; to ALU-REG, through R-ALU; or goes directly to the 32-bit output $o_{[31:0]}$, through R-O2. The other paths that reach the output come from ALU-REG and MAC-REG, through R-O1. The details of the routing system are described in Sec. III-D3.

Table 3 lists the six STAR-MAC operations and Figs. 4a–f give more details about their implementation and execution on the STAR-based Ibex. Each figure shows that every operation is executed in two clock cycles (CC1 and CC2) and highlights the involved routing paths. The wide rectangle represents MAC-REG, which is read and updated at each CC. The STAR multiplier changes its configuration according to the functional modes of Fig. 2: the active areas of its BW PPM and output rectangle, white in Fig. 2, now follow the coloring of the inputs that are multiplied and accumulated. For ST instructions (Figs. 4a–c), MAC-REG stores a single accumulated value, colored in orange. For SA instructions (Figs. 4d–f), it stores different values in parallel, highlighted by different colors.

Fig. 4a shows the operations required for the *mac16st* instruction. In CC1, STAR-MAC multiplies the 16 least significant bits of A and B, namely $A_{[15:0]}$ and $B_{[15:0]}$, and accumulates the product in the 48 least significant bits of MAC-REG. In CC2, STAR-MAC performs the same operation with the 16 most significant bits of A and B, namely $A_{[31:16]}$ and $B_{[31:16]}$. A similar approach is used for *mac8st* and *mac4st*, shown in Fig. 4b and Fig. 4c, respectively. The differences lie in the configuration of the STAR multiplier and the number of accumulated bits in MAC-REG: for *mac8st*, the STAR configuration is set to ST8 with 32 accumulated bits, while for *mac4st*, the STAR configuration is set to ST4 with 24 accumulated bits. For all three ST cases, STAR-MAC can accumulate up to 2^{16} partial results without overflow. Note that this means that, for example, a 2D Convolution with 3×3 kernels and up to more than seven thousand channels could be processed without overflow ($2^{16} > 3 \times 3 \times 7000$). To the best of our knowledge, typical models that run on MCUs have a much smaller number of channels per convolutional layer, thus overflow will never occur in practice.

Fig. 4d shows the operations involved in the *mac16sa* instruction. In CC1, $A_{[15:0]}$ and $B_{[15:0]}$ are multiplied and the result is accumulated in the blue portion of MAC-REG. In CC2, $A_{[31:16]}$ and $B_{[31:16]}$ undergo the same operation, with the accumulated result now stored in the green part of MAC-REG. This effectively keeps the results of each parallel multiplication apart. For *mac8sa* and *mac4sa* instructions, shown in Figs. 4e–f, the multiplier of the STAR-MAC unit performs SA8 and SA4 operations, respectively. In each CC, the parallel partial results are accumulated in independent portions of MAC-REG. To handle these two SA cases, a configurable adder is required, which can split into two or four sub-adders. For all SA operations, regardless of the bitwidth, our STAR-MAC unit can accumulate up to 2^5 partial results without overflow. This guarantees the correct execution of DW-Conv kernels up to 5×5 (i.e., each output needs $25 < 2^5$ accumulations). Since larger kernels are not usually present in TinyML models, we decide not to support them.

D. STAR-MAC UNIT: DESIGN CHOICES

We now explain in detail our design choices for the STAR-MAC unit. Specifically, we illustrate how we determine the number of bits required for the adder unit and the MAC register, and how we implement the routing system.

1) ADDER DESIGN

In the original Ibex, the *Fast* multiplier includes a 34-bit adder to sum partial multiplication results. However, to implement the accumulations described in Sec. III-C, we must replace it with a more complex adder. To determine its size, we must consider two possible worst cases: one for the 16-bit ST operation, represented by *mac16st*, and one for the 4-bit SA operation, represented by *mac4sa*. In the first case, performing 2^{16} accumulations of 32-bit partial products requires 16 additional bits, resulting in a 48-bit adder. In the second case, performing 2^5 accumulations of four independent 8-bit partial products leads to four distinct

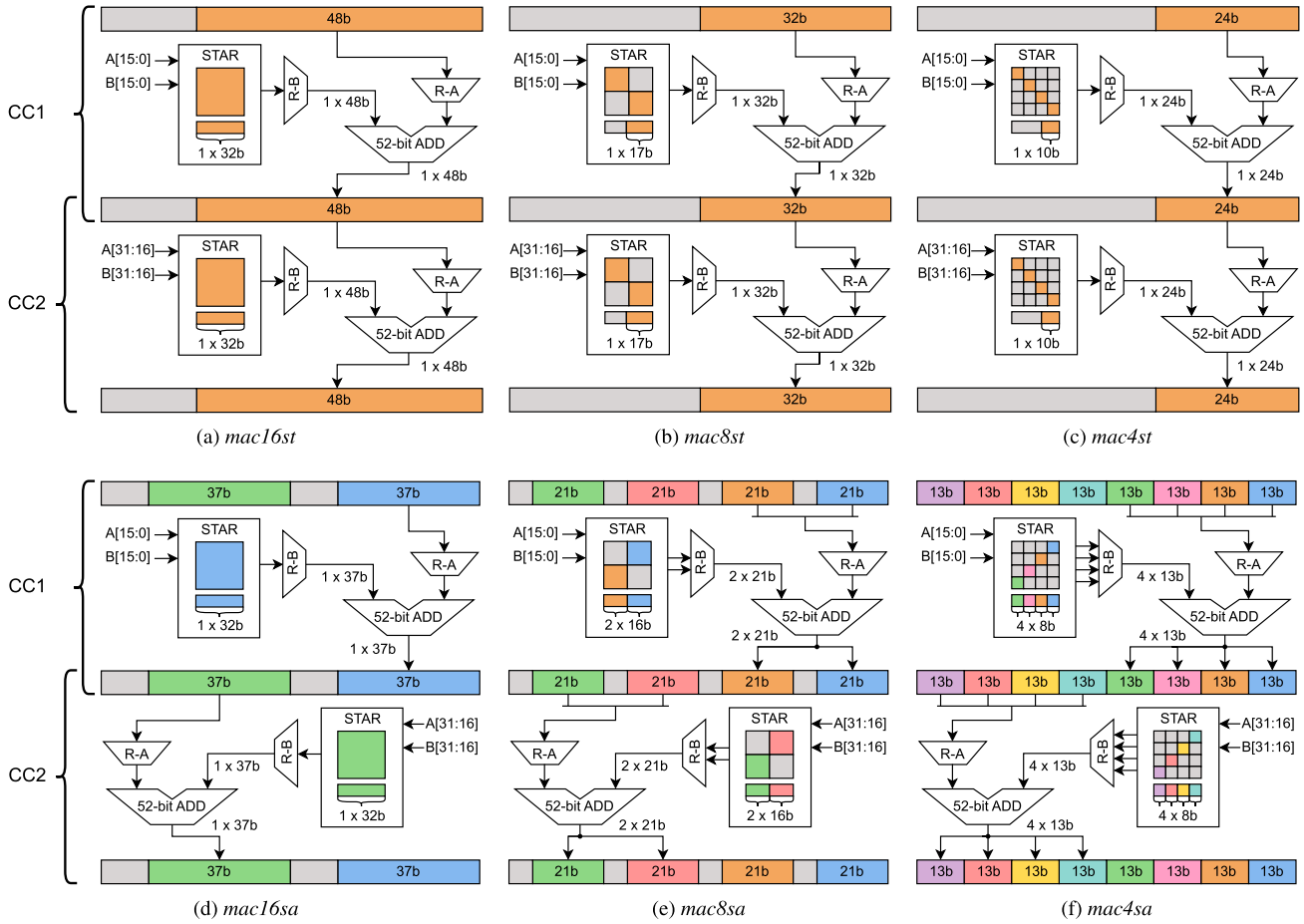

FIGURE 4. STAR-MAC operations for the instructions reported in Table 3: (a)–(c) MAC operations in ST mode for 16-, 8-, and 4-bit precision, respectively; (d)–(f) MAC operations in SA mode for 16-, 8-, and 4-bit precision, respectively. Each STAR-MAC operation uses the STAR multiplier in one configuration as reported in Fig. 2. The wide rectangle represents MAC-REG.

13-bit adders. To maximize resource sharing, we implement a single, reconfigurable 52-bit adder that can accommodate both worst-cases. The adder topology, shown at the top of Fig. 5, consists of four 8-bit and four 5-bit sub-adders, all connected through a reconfigurable carry-chain path. For standard instructions, as well as for our ST custom MAC instructions, the carry chain connects all the sub-adders, creating a single 48-bit adder. Conversely, for SA instructions, the carry chain path is configured depending on the specific instruction. For *mac16sa*, one single 37-bit adder is obtained by combining all the 8-bit sub-adders and one 5-bit sub-adder. For *mac8sa*, two 21-bit adders are formed each by combining two 8-bit and one 5-bit sub-adders. Finally, for *mac4sa*, four 13-bit adders are obtained by pairing each 8-bit sub-adder with a 5-bit one.

2) MAC REGISTER

We design this register to accumulate partial multiplication results, even for non-consecutive MAC instructions. As we saw in Sec.III-D1, *mac4sa* is the instruction responsible for producing the worst-case number of bits—i.e., 52 bits to store four 13-bit accumulations of four parallel 4-bit MAC operations. Since SA operations keep each low-precision result separate in the multiplier's output and require two clock cycles to complete, we need a MAC register size twice this worst case number of bits, resulting in a 104-bit register.

3) THE ROUTING SYSTEM

Figs. 5-6 show more in detail the routing paths presented in Fig. 3 and discussed in Sec. III-C. The output $s_{[31:0]}$ of the STAR multiplier is directly connected to the least-significant 32 bits of the rightmost input of the 52-bit adder via R-B, highlighted in red in Fig. 5. The remaining 20 bits of the rightmost input are either set to zero or set equal to the most significant bit of $s_{[31:0]}$, which can be $s_{[31]}$, $s_{[23]}$, $s_{[15]}$, or $s_{[7]}$, depending on the instruction. The leftmost input of the adder is provided through R-A, highlighted in orange in Fig. 5. It comes either from the output of ALU-REG, by sign-extending it (through the *ext* blocks) or concatenating it (through the *cnct* blocks), or from the output of MAC-REG.

The routing signals to ALU-REG, shown in green in Fig. 5, simply pass through 3-to-1 multiplexers. Notice that ALU-REG is partitioned in sub-registers to match the various sub-adder components. For the same reason, MAC-REG is

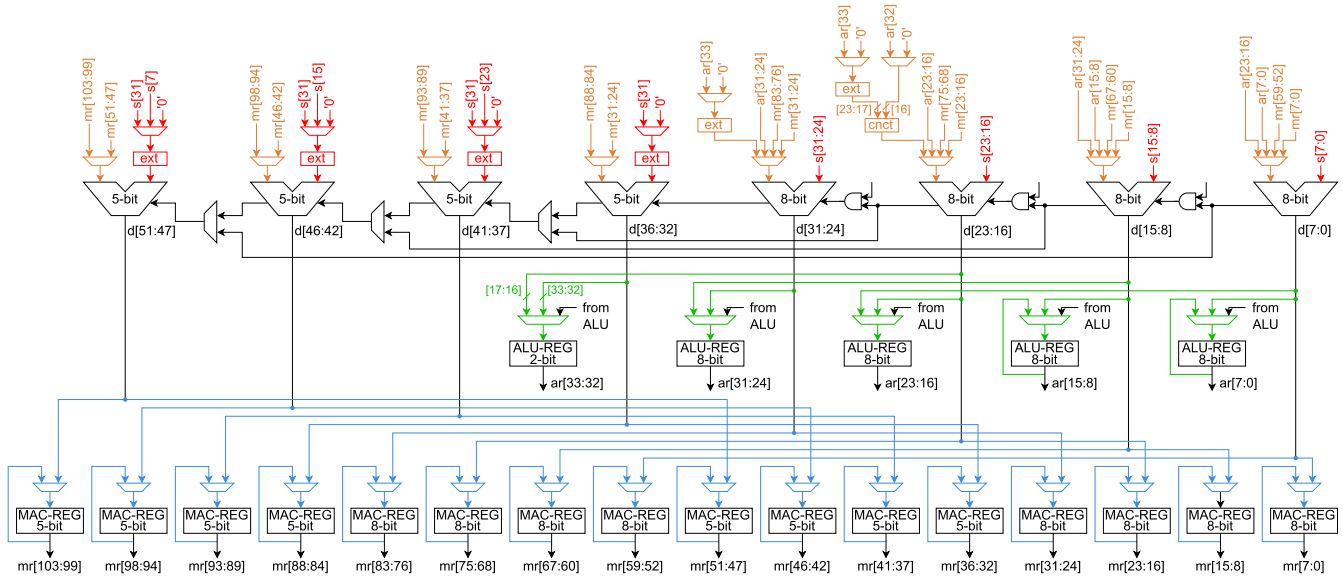

FIGURE 5. Reconfigurable adder, ALU-REG, MAC-REG and routing paths of STAR-MAC: R-A (orange), R-B (red), R-ALU (green), and R-MAC (light blue).

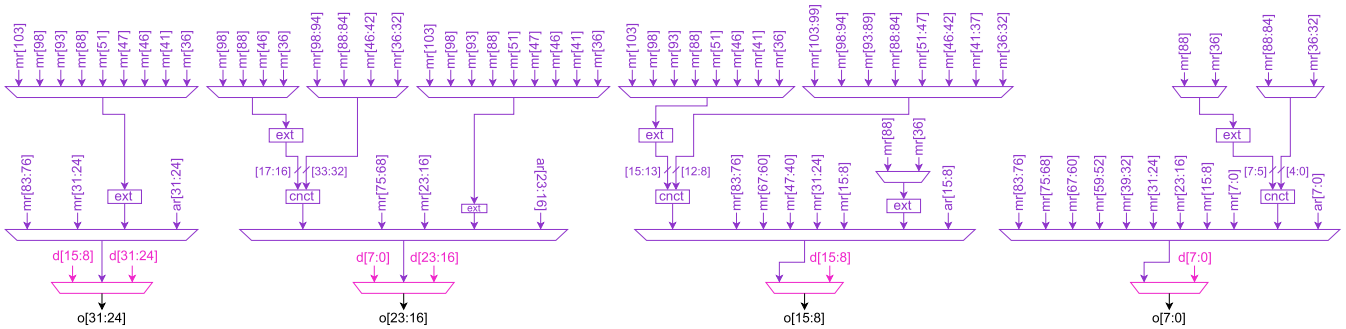

FIGURE 6. Routing paths R-O1 (purple) and R-O2 (pink) for producing the final output $o_{[31:0]}$ of the STAR-MAC unit.

partitioned into 8-bit and 5-bit sub-registers. The routing signals of MAC-REG, in light blue in Fig. 5, are made of interconnects with a fanout of two that go into 2-to-1 multiplexers.

Fig. 6 provides more details about the routing paths R-O1 and R-O2 that produce the final output $o_{[31:0]}$ of the STAR-MAC unit. R-O1, in purple, depending on the instruction, forwards ALU-REG or a selected portion of MAC-REG to $o_{[31:0]}$. R-O2, in pink, is simply one level of multiplexers to select between the output of R-O1 or the output of the adder.

IV. OVERVIEW OF THE MP-QNN DEPLOYMENT FLOW

Fig. 7 depicts the seven-step flow that we used to train and deploy MP-QNNs on the STAR-based Ibex, together with collecting results for SoA comparisons.

Block A) is the starting point. It takes as input a pre-trained QKeras DNN model and some user-defined constraints, such as the allowed bitwidths for weights/activations and the maximum accepted accuracy drop [9]. Then, an automated optimization loop, driven by AutoQKeras's Bayesian Optimization engine, selects a tentative MPQ policy to meet the constraints and achieve the desired trade-off between accuracy and model size, measured as the total number of bits in weight and activation tensors. Notice that, in our case, a smaller model size is beneficial not only to reduce memory occupation, but it also translates directly in a reduction of its execution time on MCU, as we take advantage of the PS hardware with STAR-MAC.

The best MPQ model is processed in block B) by the QKeras-to-TFLM converter. This Python script uses QKeras APIs to extract model parameters and topology, and so build a FlatBuffer file similar to the one generated by TF-Lite when converting a TensorFlow/Keras model. Notice that our modular approach allows users to replace A) and B) with their preferred alternatives (e.g., a custom flow from PyTorch to FlatBuffer) [1], as shown by the dashed lines on the left of Fig. 7.

In point C), we create and integrate in TFLM a few custom kernels, including 2D-Conv, DW-Conv and FC, to exploit the acceleration provided by the STAR-MAC unit, along with some minor modifications to enable MP execution of QNNs.

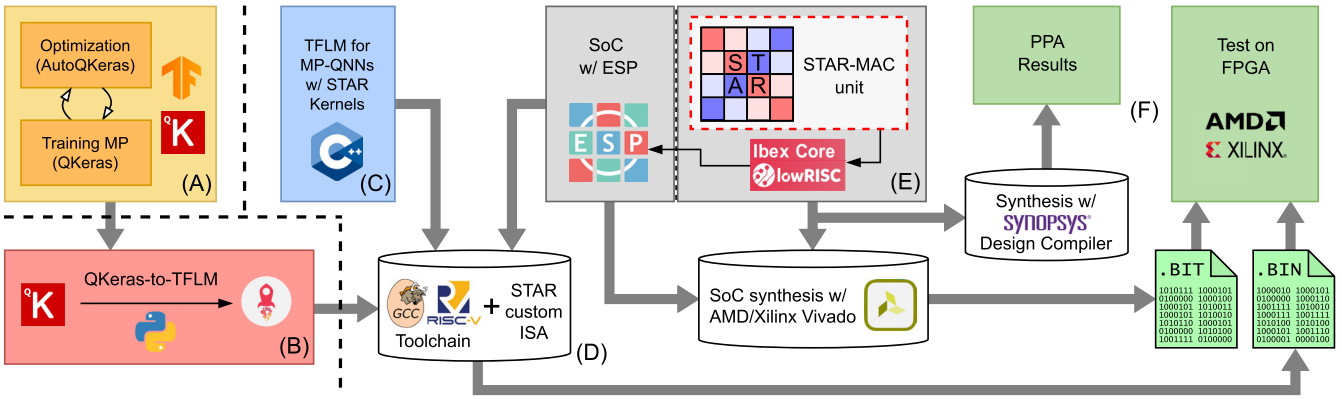

FIGURE 7. The flow we used to train, deploy and accelerate TinyML MP-QNN models on our STAR-based RISC-V Ibex MCU.

In block D), the new FlatBuffer file generated in B), the source code of TFLM modified in C), and some platform-dependent files (e.g., linker scripts for memory organization, platform-dependent support functions such as the printf support, etc.) are used by the GCC RISC-V Toolchain [39] to generate the processor's executable binary file.

To design an SoC that integrates our STAR-based Ibex, we leverage the silicon-proven ESP [40] shown in block E). ESP facilitates the integration of processors, accelerators, memory interfaces, and peripherals. It already supports the Ibex, along with the necessary platform-dependent files required to run a baremetal application. Since the STAR-based Ibex retains the original top-level interfaces, we seamlessly integrate it into ESP by simply replacing the Ibex Verilog source code.

In point F), the final SoC can be synthesized targeting both ASIC and FPGA technologies. In the former case, our flow provides PPA estimations using Synopsys Design Compiler, invoked via custom synthesis scripts. In the latter case, it leverages ESP's integration with AMD/Xilinx Vivado HLS for bitstream generation. Finally, the bitstream is used to program the FPGA, in which the STAR-based Ibex softcore runs the bare-metal application that includes the final MP-QNN.

It is worth noting that the flow used in this paper can also target other MPQ processors, as long as block E), and optionally blocks C) and D), are adapted accordingly.

V. QKERAS-TO-TFLM CONVERTER

Since the minimum precision supported by TF-Lite when generating the Flatbuffer is 8-bit, and it does not support QKeras QNNs, our QKeras-to-TFLM converter replaces TF-Lite in producing the FlatBuffer. With this crucial step, we establish for the first time a path to deployment from a QKeras MP-QNN model to TFLM.

Our new converter takes as input a QKeras model and builds an equivalent JSON description of its topology, including weights and quantization parameters, such as scaling factor (S) and zero point (Z) of weights and activations. The conversion from JSON to FlatBuffer uses the original *schema* file provided with the TFLM framework. This ensures that the FlatBuffer remains fully compatible with the TFLM execution flow.

To perform such a conversion, some preliminary adaptations are needed to account for the differences in quantization between TF-Lite and QKeras. First, TF-Lite assumes *scale quantization* ($Z = 0$) for weights and *affine quantization* ($Z \neq 0$) for activations [41]. However, QKeras does not natively support affine quantization for activations among its quantizers [9]. Therefore, according to [3], [42], users need to modify their QKeras models and insert new QActivation layers, as detailed in [3] after 2D-Conv, DW-Conv, FC, element-wise addition (Add) and average pooling (Avg-P) layers, to have independent S and Z values for their output tensors. Moreover, users need to replace batch normalization (BN) layers that follow 2D-Conv, DW-Conv, and FC layers with their folded versions: *QConv2DBatchnorm*, *QDepthwiseConv2DBatchnorm*, and *QDenseBatchnorm*.

Next, while S and Z of weight layers are easily extracted with the corresponding QKeras methods, extracting S and Z for activation layers requires a post-training calibration step, as also required by the standard TF-Lite flow.

Another difference between QKeras and TF-Lite concerns the quantization of the bias in a weight layer. TF-Lite quantizes it using a scaling factor (S_b) that is the product of the input (S_i) and weight (S_w) scaling factors of that layer [43], [44]. QKeras, instead, quantizes the bias with its own scaling factor, independent from S_i and S_w. To solve this mismatch, we force $S_b = S_i \times S_w$ before writing it in the FlatBuffer. This solution guarantees a proper QKeras-to-TFLM conversion, but may affect accuracy as shown later in Sec. VII. Finally, for Conv, DW-Conv, and FC layers, if the number of channels is not a multiple of the parallelism of the corresponding STAR instruction, in the final iteration the last of such instructions would have fewer inputs than its operands. In such cases, we increase the channels of the weight tensors to the nearest multiple of the STAR instruction parallelism and fill these extra channels with zeroes. We quantify the impact of these changes on the FlatBuffer size in Sec. VII.

In summary, the QKeras-to-TFLM converter supports 2D-Conv, DW-Conv, FC, Add, max pooling (Max-P), Avg-P, and Softmax operators, with 16-, 8-, and 4-bit integer quantization for all layers except Softmax, which uses 16 bits to preserve accuracy.

VI. STAR-TFLM

In this section, we introduce TFLM and present the changes made to support MPQ, which result in the custom version that we call *STAR-TFLM*. In short, STAR-TFLM supports: a) kernels for MPQ that implement the most computationally intensive layers, namely 2D-Conv, DW-Conv, and FC, leveraging the STAR-MAC instructions; b) other kernels with MPQ support, namely Add, Max-P, and Avg-P. Although in our experiments and to ensure reproducibility we select a specific TFLM version [12], adaptation to newer versions is straightforward.

A. TFLM BACKGROUND

TFLM is a framework to easily execute DNNs on MCUs. Its success stems from a lightweight and platform-agnostic implementation of the engine that manages the underlying DNN computational graphs, while also offering high customizability of the computational kernels to execute DNN operators associated with each node of that graph, i.e., the DNN layers. Internally, TFLM goes through two phases. First, we have an *initialization* phase in which the FlatBuffer file is parsed by the MCU. For each node of the graph, the MCU extracts the layer information, such as S, Z, and the pointers to the weights memory location. Then, it allocates and fills the corresponding per-layer C-structs needed in the next phase. Second, we have the *invoke* phase. At this stage, an input sample is written into the input memory location and the DNN inference process starts. Specifically, for each operator of the execution graph, an evaluation method is called, which is in charge of performing the operations associated with that specific layer of the DNN.

B. STAR MODIFICATIONS

Inspired by vendor-specific source files that provide specialized computational kernels tailored to their commercial MCUs, we include our custom kernels into the TFLM compilation flow to exploit the STAR-MAC unit of our modified Ibex processor. Specifically, we create two folders named *star* in the TFLM directories *./tensorflow/lite/micro/kernels/* and *./tensorflow/lite/kernels/reference/integer_ops/*, containing specialized MP-kernels that use STAR-MAC instructions to support MP-QNNs in TFLM. Here, we report only the most relevant modifications to the original TFLM.

1) GENERAL MODIFICATIONS

We add the INT4 data type and expand the symmetric kernels (s) to support INT16, INT8, and INT4 data types, as well as asymmetric kernels (a) to support any combination of INT16, INT8, and INT4, for the following layers: 2D-Conv (s, a), DW-Conv (s, a), FC (s, a), Add (s, a), Max-P (s), and Avg-P (s). We also expand the set of TFLM unpacking functions by adding unpacking schemes 4-to-16 and 8-to-16 bits. These adjust the bitwidth of input and weight operands for STAR-MAC, when these are quantized asymmetrically (e.g., 16×4): operands with fewer bits are extended to match the size of those with more bits.

2) MODIFICATION TO THE INITIALIZATION PHASE

For each DNN layer, we modify the corresponding TFLM files[1] by adding memory allocation requests for the missing asymmetric precision cases. In addition to the original per-layer quantization, we further modify the FC files to add support for per-channel quantization [41], which typically results in better accuracy [41]. In fact, we use it for FC layers in our experiments.

3) MODIFICATION TO THE INVOKE PHASE

In this phase, for each layer of the DNN graph, a specific computational kernel is called. Like the original TFLM, STAR-TFLM checks input, weight, and output data types to determine which kernel and unpacking function to call. For the Add layer with 4-bit input precision, we use the 4-to-8 unpacking function. For inputs with 8- or 16-bit precision, no changes are required. For Max-P and Avg-P layers, we apply the unpacking functions as for the Add layer. Moreover, for Avg-P, we add the code to manage S and Z for the output re-quantization, which is not present in the original TFLM. For the accelerated layers 2D-Conv, DW-Conv, and FC, when inputs and weights have asymmetric precision (e.g., 16×4), we use the proper unpacking functions described in Sec. VI-B1—i.e., if an accelerated layer has 16-bit inputs and 4-bit weights, our implementation unpacks the 4-bit weights to 16 bits, and the 16×16 kernel is invoked.

4) MODIFICATIONS TO THE COMPUTATIONAL KERNELS

We report in Alg. 1 a simplified pseudo-code that represents our STAR-based TFLM kernel for 2D-Conv, of which the FC kernel can be considered a special case. We highlight in orange the *for loop* where the parallelization enabled by STAR occurs. This is the inner-most one, as TFLM kernels use an output stationary dataflow. The number of iterations of these loops gets divided by the number of parallel MAC operations that STAR-MAC can compute (N_p), which is a function of the number of bits of its input and weight operands (N_b). Inside the inner-most loop, *macst{N_b}* is called, where $N_b = \{16, 8, 4\}$ bits.

ST operations suit perfectly FC/2D-Conv kernels. In fact, in ST mode, STAR can perform the accumulation of the products between inputs and weights along the input neurons/channels, saving external additions, as discussed in [3], [4]. At the end of the accumulations, the content of MAC-REG is retrieved, as summarized by the *retrieve* operation in the pseudo-code, visible at line 15 of Alg. 1.

[1]Modified files are at relative path *./tensorflow/lite/micro/kernels* and named {layer_name}.cc, {layer_name}.h, and {layer_name}_common.cc.

Algorithm 1 2D-Conv Layer Pseudo-Code

```
1:  Require: N_b = 16, 8, 4
2:  N_p = 32/N_b
3:  for b = 0 to batch_size do
4:    for oh = 0 to out_height do
5:      for ow = 0 to out_width do
6:        for och = 0 to out_channels do
7:          macrst()
8:          for fh = 0 to filter_height do
9:            for fw = 0 to filter_width do
10:             for fch = 0 to filter_channels/N_p do
11:               w_idx ← f(oh, ow, och, fh, fw, fch)
12:               i_idx ← f(b, oh, ow, och, fh+oh, fw+ow, fch)
13:               macst{N_b}(W[w_idx], I[i_idx]) // call the proper macst
    (Tab. 3)
14:         o_idx ← f(b, oh, ow, och)
15:         O[o_idx] ←retrieve(MAC-REG)
```

Algorithm 2 Depthwise Conv Pseudo-Code

```
1:  Require: N_b = 16, 8, 4
2:  N_p = 32/N_b
3:  for b = 0 to batch_size do
4:    for oh = 0 to out_height do
5:      for ow = 0 to out_width do
6:        for och = 0 to out_channels/N_p do
7:          macrst()
8:          for fh = 0 to filter_height do
9:            for fw = 0 to filter_width do
10:             w_idx ← f(oh, ow, och, fh, fw)
11:             i_idx ← f(b, oh, ow, och, fh+oh, fw+ow)
12:             macsa{N_b}(W[w_idx], I[i_idx]) // call the proper macsa
    (Tab. 3)
13:           for np = 0 to N_p do
14:             o_idx ← f(b, oh, ow, och, np)
15:             O[o_idx] ←retrieve(MAC-REG[np])
```

We omit the bias addition and the re-quantization of the output—i.e., the operations involving S, Z, rounding and clipping that adjust the output bitwidth for the next layer—to keep the pseudo-code clean and simple.

The pseudo-code of our STAR-based DW-Conv TFLM kernel is shown in Alg. 2. Accelerated loops are highlighted in orange. Here, N_p channels are computed in parallel and their accumulated results are stored separately in MAC-REG, until all the convolutions between the N_p kernels and the corresponding receptive fields are computed. Since no accumulation along the channels occurs, SA instructions fit perfectly this task [3], [4]. Similarly to the FC/2D-Conv kernels, we develop a DW-Conv kernel for each *macsa{N_b}* instruction, where $N_b = 16, 8, 4$ bits. In this case, a loop is needed to retrieve all independent results from MAC-REG, as shown at lines 13–15.

VII. EXPERIMENTAL RESULTS

In our experiments, we use the well-known MLPerf Tiny benchmark [10], developed collaboratively by academia and industry to evaluate TinyML software and hardware solutions. It comprises four models, each designed for a specific task representing typical edge applications executed on MCU-based platforms: anomaly detection (AD), image classification (IC), visual wake words (VWW), and keyword spotting (KS).

The AD task uses an autoencoder, which is a sequence of FC layers. This model uses Area Under the receiver operating characteristics Curve (AUC) as performance metric, while the others use accuracy. IC task uses a custom version of ResNet-8. VWW uses a smaller version of MobileNetV1, which stacks several DW-Separable layers, whose structure consists of a DW-CONV layer followed by a Point-Wise layer, i.e., a standard 2D-CONV with 1×1 filters. KS uses the model named *ds_cnn*, also made of DW-Separable layers.

A. MODEL TRAINING AND ACCURACY ANALYSIS

For each model, we use AutoQKeras to find the corresponding MP-QNN model that maximizes the performance on the validation set and minimizes the total number of bits for each layer. More information on the search strategy and the per-layer precision distribution of the resulting models can be found in [3] and [9], respectively. Table 4 reports the test accuracies of the four MLPerf Tiny MP-QNN models from AutoQKeras (*MPQ* in Table 4) and the flat-quantized models, where all layers use the same precision. Our baseline is the test accuracy obtained in QKeras after training the 4-, 8-, 16-bit flat models and the MP-QNN models, as reported in column (1). Column (2) reports the test accuracy of QKeras, while column (3) reports the test accuracy after the calibration process. We also report the results of an ablation study aimed at assessing two optimizations introduced in STAR-TFLM with respect to the original TFLM, as previously described

TABLE 4. Test accuracy for the (1) flat- and MP-quantized MLPerf Tiny models obtained in (2) QKeras, (3) QKeras after calibration, (4) QKeras after calibration without output re-quantization for Avg-P layers, and (5) QKeras after calibration with bias S (S_b) forced like in TFLM. In bold the accuracy obtained with the strategy selected for STAR-TFLM.

(1) MLPerf Tiny Model	(2) QKeras	(3) QKeras w/ calibr	(4) QKeras w/ calibr w/o Avg-P re-quant	(5) QKeras w/ calibr w/ $S_b{=}S_i{\times}S_w$
Image Classification (IC) [%]				
4-bit flat	79.0	76.5	62.0	**72.0**
8-bit flat	89.5	90.5	90.0	**90.0**
16-bit flat	89.5	89.0	89.0	**84.5**
MPQ	87.5	88.0	88.0	**88.0**
Visual Wake Words (VWW) [%]				
4-bit flat	50.0	50.0	50.0	**50.0**
8-bit flat	87.2	87.5	87.5	**87.3**
16-bit flat	87.3	87.4	87.4	**84.3**
MPQ	85.3	84.3	75.7	**83.4**
Keyword Spotting (KS) [%]				
4-bit flat	70.7	33.9	50.9	**35.4**
8-bit flat	91.4	91.3	91.3	**91.2**
16-bit flat	91.3	91.3	91.3	**91.3**
MPQ	89.4	88.4	88.4	**89.6**
Anomaly Detection (AD) [AUC]				
4-bit flat	0.68	0.54	–	**0.58**
8-bit flat	0.91	0.88	–	**0.88**
16-bit flat	0.88	0.84	–	**0.65**
MPQ	0.89	0.88	–	**0.88**

in Secs. V–VI. In the first one, we disable in QKeras the re-quantization of the output tensor of the Avg-P layer (column (4)): this affects all models except AD, which has no pooling layers. In the second, we force in QKeras the method used in TFLM for the scaling factor of bias values, i.e., $S_b = S_i \times S_w$ (column (5)).

In general, we found that setting S and Z of the activation layers to fixed values obtained through calibration does not significantly impact accuracy. An exception is the 4-bit flat-quantized models, which already exhibit clearly suboptimal accuracy. Nonetheless, we include them in the performance evaluation to represent a lower bound on computational cost.

The results in column (4) show that, while IC and KS retain the same accuracy of column (3), the MP-QNN VWW model experiences a 9% drop. The reason is that this model uses 4-bit for the Avg-P output. These findings highlight the importance of re-quantizing pooling outputs to preserve accuracy.

The results reported in column (5) show that using the TFLM choice for S_b has a detrimental impact for three out of four models (IC, VWW, AD) in the 16-bit flat case. In contrast, the MPQ versions are minimally affected, even though some of our MP-QNN models indeed contain 16-bit layers. Therefore, we decide to keep the TFLM choice for S_b, and thus the results in bold represent those obtained with our STAR-TFLM.

B. TEST METHODOLOGY

We use ESP [11] to integrate our modified Ibex in an SoC and test it on an FPGA. We compare the original TFLM against our STAR-TFLM when running the flat-quantized and the MP-QNN MLPerf Tiny models, under seven different experiments. In each experiment, we measure the total inference time in clock cycles using the RISC-V internal performance counter, discarding the results of the first samples for cache warm-up. As for the ESP configuration, we use three tiles: processor, I/O, memory. Here we explain the seven experiments, whose results are reported in Figs. 8–10.

1) In the *standard* case, we follow the original 8-bit TFLM deployment flow to deploy the four 8-bit flat-quantized MLPerf Tiny QKeras models without STAR instructions.
2) In *standard unroll*, we still use the same 8-bit flat-quantized models of the standard case without STAR instructions. However, we manually unroll the loops in which the STAR parallelization would occur if we used the STAR instructions (loops highlighted in orange in Algs.1–2). In this way, the resulting TFLM code executes these loops the same number of times as the STAR-TFLM code would do when using the 8-bit STAR-TFLM kernels. This experiment represents our baseline to show the benefits of using the STAR-MAC unit: by comparing the STAR-accelerated cases with this case, we will appreciate the hardware speed-up given solely by STAR instructions. Thus, we normalize all the other results in Figs. 9–10 to this experiment.
3) In *emulate star*, again we use the 8-bit flat-quantized models, this time with the STAR-TFLM code. However, STAR custom instructions are emulated by a sequence of standard RISC-V 32-bit instructions mimicking the arithmetic operations in Table 3. This setup allows us to assess the overhead of the STAR-TFLM code with respect to the original TFLM. Moreover, it enables users to run the entire flow, or STAR-TFLM alone, on standard MCUs without the STAR-MAC.

4-6) In *STAR int16*, *STAR int8*, and *STAR int4* we use the 16-, 8-, and 4-bit flat-quantized models, respectively, and we use the STAR-TFLM code with our custom STAR instructions. It is precisely from these experiments that we expect a significant reduction in execution time.

7) From *STAR MPQ*, where we use MP-QNN models with our STAR-TFLM custom code we expect the best trade-off of execution time, accuracy, and FlatBuffer size.

In Figs. 8–10, we use different patterns to highlight the fraction of time spent executing the main kernels (2D-Conv, DW-Conv, and FC), unpacking (as required by STAR-TFLM), and other non-accelerated TFLM kernels (Add and Softmax).

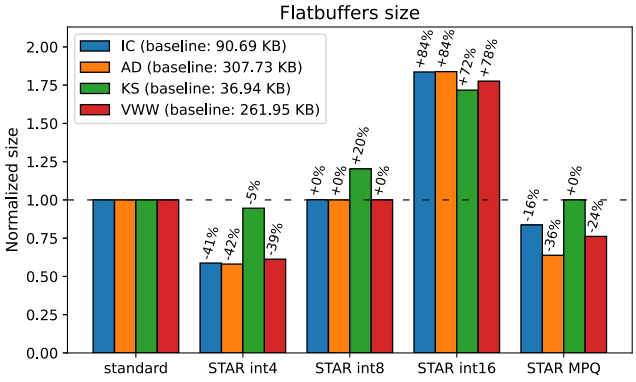

FIGURE 8. Comparison of FlatBuffer size of the four MLPerf Tiny models (IC, AD, KS, and VWW) in the different experiments explained in Sec. VII-B.

C. FLATBUFFER SIZE

Fig. 8 reports the FlatBuffer size for all experiments and for all MLPerf Tiny models. We omit *standard unroll* and *emulate STAR* because their sizes are the same as the *standard* version.

In *standard*, the weights are quantized with 8 bits and packed in 32-bit words (4 weights per word). For this reason, its FlatBuffer size is similar to that of *STAR int8*, except for KS that shows an increment of 20% that will be explained shortly. As expected, the size of the FlatBuffer for *STAR int16* almost doubles that of the *standard* version (+80% on average over the four MLPerf Tiny models). In contrast, since the *STAR int4* case packs eight 4-bit weights in 32-bit words,

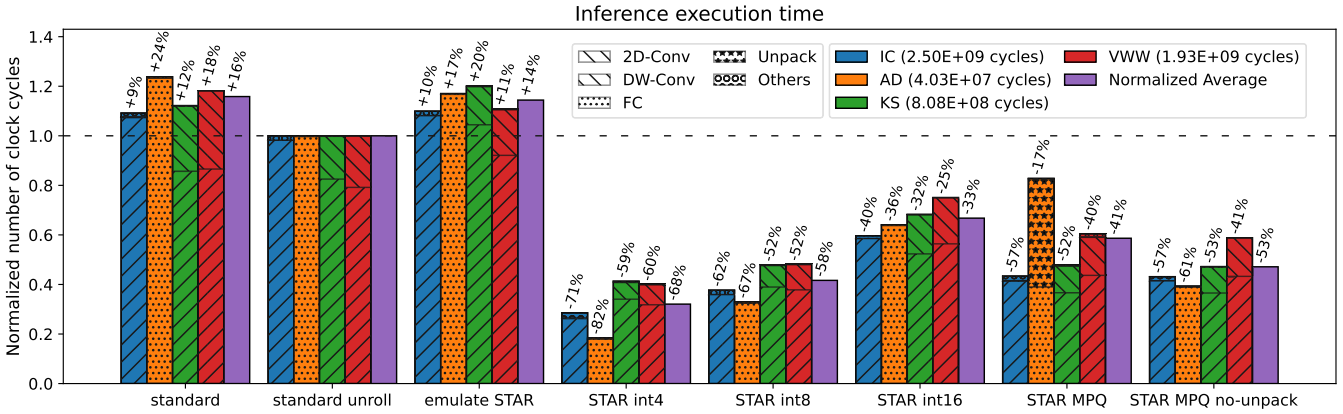

FIGURE 9. Execution times, in terms of clock cycles, of the four quantized MLPerf Tiny models (IC, AD, KS, VWW) and their average. Results obtained on the STAR-based Ibex for the seven experiments explained in Sec. VII-B. Clock cycles are normalized to the *standard unroll* case. The number of clock cycles to execute each of the network in the *standard unroll* case is reported in parenthesis in the top-right legend. Execution times of layers (2D-Conv, DW-Conv, etc.) are shown through patterns. *STAR MPQ no-unpack* removes the cycles of the unpacking functions from *STAR MPQ*.

it almost halves the memory requirements, in most cases, compared to the *standard* configuration. In fact, the average reduction for this case is −32%, with KS showing a small reduction of −5% even though the overall network is flat-quantized at 4 bits. Notice that the obtained reduction is less than the theoretical −50% decrease of model size for an ideal INT4 quantization. Notice also the +20% increase of KS in the *STAR int8* case, which we would expect to have the same size of the *standard* case. This is because of the padding that must be introduced to fulfill the requirements on the number of operands of the STAR-MAC unit, as explained in Sec. V. In particular, the +20% increase and the −5% decrease of the FlatBuffer size of KS against *standard* is due to its particular model architecture, which has only one channel in the input tensor. This means that the QKeras-to-TFLM converter needs to pad the weight tensor of the first layer with many zeros (seven in the 4-bit case, three in the 8-bit case, and one in the 16-bit case). Moreover, the kernel size of these filters is 10×4, resulting in 40 elements per channel, which is much greater than the 9 elements of an usual 3×3 filter. Thus, the padding significantly burdens the final FlatBuffer size for KS. Overall, *STAR MPQ* shows a smaller FlatBuffer size than the *standard* case, primarily because the majority of layers are quantized with 4 and 8 bits.

D. EXECUTION-TIME RESULTS

Fig. 9 reports the execution times of the four quantized MLPerf Tiny models running on the STAR-based Ibex in the seven experiments. Each result is normalized with respect to the *standard unroll* case. The number of clock cycles of this case is reported in parenthesis in the top-right legend of the figure. For each experiment, we also add a fifth bar for the average of the normalized values.

The *standard* version of TFLM and the *emulate STAR* version require, on average, 16% and 14% more clock cycles, respectively, than the *standard unroll* baseline. This result quantifies the unavoidable software overhead of STAR-TFLM, as estimated by *emulate STAR* in comparison with the baseline. When exploiting the MAC-STAR unit and its custom instructions, we observe a reduction by −33%, −58%, and −68% for the *STAR int16*, *STAR int8*, and *STAR int4*, respectively, with a peak reduction of −82% for AD in the *STAR int4* case. The *STAR MPQ* shows an average reduction of −41%. However, this average result is largely impacted by the high number of unpacking operations of the MP-QNN model in the AD case. This is evident from the starred orange pattern in Fig. 9, which shows that the cycles spent on unpacking operations account for more than half of the execution time. The reason is that the autoencoder has layers with a large number of weights that need to be unpacked at runtime when its layers are quantized with a different bit-widths for inputs and weights.

Note that if the inference is run consecutively for a number of times, weights can be unpacked only once. To assess the performance in the best case in which the unpacking is amortized over a very large number of runs, in the right-most experiment called *STAR MPQ no-unpack* in Fig. 9, we remove the unpacking time. In this experiment, the execution time for AD is halved and the average number of clock cycles for executing MP-QNN MLPerf Tiny models using STAR-TFLM against standard unroll is reduced from −41% to −53%.

Fig. 10 shows the normalized reduction in the number of clock cycles per type of layer, namely 2D-Conv, DW-Conv, and FC, averaged over the four MLPerf Tiny models. The graph compares the execution time of the three types of layer that can be accelerated by our STAR-MAC unit. Overall, we observe a similar trend as in Fig. 9: *standard* is the slowest, and *STAR int4* is the fastest. The FC layer achieves the highest acceleration factor, mostly due to the higher ratio of accelerated vs total operations in its algorithm compared to 2D-Conv and DW-Conv. For the same, yet opposite reason, the DW-Conv layer benefits less from acceleration than the others.

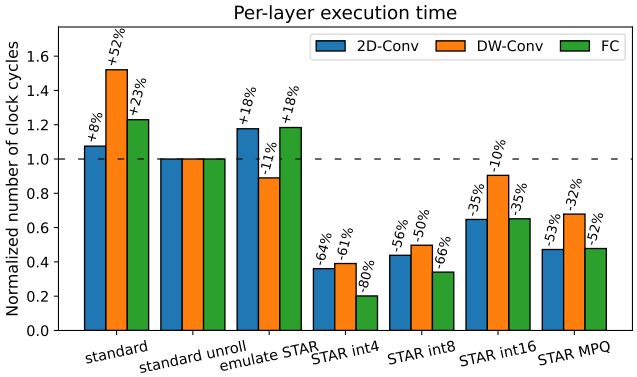

FIGURE 10. Normalized number of clock cycles per type of layer (2D-Conv, DW-Conv, and FC) averaged over the four MLPerf Tiny models.

E. DISCUSSION

From the results reported in Table 4 and Figs. 8-10, we can make the following observations:

- MP-QNNs obtained with AutoQKeras retain the FP accuracy and are competitive with other quantization techniques such as 8-bit flat quantization;
- STAR can provide a significant acceleration, with *STAR MPQ* achieving an average latency reduction of 53%.
- The FlatBuffer size for these MP-QNNs is on average 27% lower than the 8-bit flat quantization strategy.

These results highlight a trade-off between accuracy, inference time, and FlatBuffer size. As expected, the 4-bit quantized model has the best latency (−68%), the smallest FlatBuffer (−32%), and the lowest accuracy. Conversely, the 16-bit flat case has the same accuracy of the FP case, but shows the worst latency (−33%) and the largest FlatBuffer size (+80%). Our proposed MP-QNNs [3] fall between the 16- and 8-bit in inference time, and between the 8- and 4-bit in FlatBuffer size. Notice that, despite the small power overhead reported in Table,5, the energy is also significantly reduced due to the large reduction in latency.

F. COMPARISON AGAINST STATE-OF-THE-ART MPQ PROCESSORS

In Table 5, we compare our STAR-based Ibex with other SoA processors that support MPQ. Our version of the Ibex is synthesized at 250 MHz using Synopsys Design Compiler on a commercial 28-nm standard cell library, while area and power figures of the other processors are taken from the corresponding publications. Notice that all processors are directly comparable because they share the same technology node, except for [20] which is synthesized using a newer technology but, despite this, reports sub-optimal results when compared to our solution. Moreover, we do not compare with Dustin [21] because it was proven in [14] that it is sub-optimal in area and power with respect to XPulpNN [20].

From the analysis, it is evident that our solution achieves the smallest silicon area. At an equivalent clock frequency of 250 MHz, it occupies approximately 46% less area than

TABLE 5. Comparison against other SoA MPQ processors.

		XPulp-NN [20]	XPulp-NN TR‡ [14]	Bisdu TR [14]	ICCAD'24 Ibex [19]	Ibex [This work]
Technology [nm]		FDX 22†	FDSOI 28†	FDSOI 28†	TSMC 28 (ASAP 7)†	TSMC 28
Precision [bit]		32 16/8/4/2 sym.	32 16/8/4/2 sym.	range 32 to 1 asym.	32 8a. 8/4/2w. asym.	32 16/8/4 sym.
Clk Freq. [MHz]		250	1'150	1'150	250 (core) 500 (MAC)	250
Area [µm²]		21'913†	27'528†	23'159†	17'361 (38'000)†	11'893
Power [mW]		1.22†	19.77†	16.71†	1.50 (0.58)†	0.96
MACs/clk	16b	2	2	0.125	–	1
GOPS	16b	1	4.6	0.29	–	0.50
GOPS/W	16b	820	233	17	–	522
GOPS/W/mm²	16b	37.4	8.5	0.7	–	**43.9**
MACs/clk	8b	4	4	0.5	4	2
GOPS	8b	2	9.2	1.15	2	1
GOPS/W	8b	1639	465	69	1330	1045
GOPS/W/mm²	8b	74.8	16.9	3.0	76.6	**87.9**
MACs/clk	4b	8	8	2	8	4
GOPS	4b	4	18.4	4.6	4	2
GOPS/W	4b	3279	931	275	2660	2090
GOPS/W/mm²	4b	149.6	33.8	11.9	153.2	**175.7**

GOPS = 2 [OPs/MAC] × [MACs/clk] × Clk Freq. [GHz].
‡ In [14], XPulpNN is implemented on a TinyRocket (TR) core.
† Values from the authors' corresponding paper.

XPulpNN, and 31% less than the Ibex variant presented in [19], after normalizing its values to the our technology node to ensure a fair comparison.

This advantage comes from two contributions: the initial choice of processor—we intentionally selected a minimal-area baseline to better suit ultra-constrained IoT edge applications with MP workloads—and the STAR-MAC efficiency, as we discuss at the end of the subsection.

Our design also shows the lowest power consumption (0.96 mW at 250 MHz), outperforming XPulpNN and [19] by 21% and 36%, respectively, despite being synthesized on an older technology node (28 nm). Furthermore, compared to the original Ibex [5], the STAR-based Ibex incurs only a 7.3% increase in power consumption.

Nonetheless, our integrated STAR-MAC unit results also in a limited area overhead of just 12.2% compared to the original Ibex, synthesized with the same technology and clock frequency, allowing the final design to maintain a competitive area footprint.

When considering theoretical energy efficiency (GOPS/W) (i.e., excluding software and compiler overheads), STAR-MAC outperforms Bisdu [14] by 30.71× at 16-bit, 15.14× at 8-bit, and 7.60× at 4-bit, and XPulpNN on the TinyRocket (TR) core by 2.24× at each precision. Nonetheless, its energy efficiency is 36% lower than XPulpNN across all

precisions, and 21% lower than [19] at 8- and 4-bit. However, if we use a metric that incorporates area together with performance and power, such as energy efficiency per unit area (GOPS/(W·mm^2)), our STAR-based Ibex improves over [20] by 14.8% at 16-bit, and over [19] and [20] by 14.9% and 12.9% at 8-bit, and by 14.9% and 12.8% at 4-bit, respectively.

To conclude, we compare the solutions of Table 5 in terms of MAC unit.

Comparing STAR-MAC with [19] and XPulp-NN [29], our solution results in a smaller area occupation because it uses a single multiplier to support different precisions through internal reconfiguration [4]. In fact, [19] proposes a MAC design with four 17-bit multipliers, while [29] implements precision-scalability via many mutually exclusive dataphats, each composed of N multipliers according to the supported operands precision, e.g., N={1,2,4} for 16- 8- and 4-bit operands. Finally, since the results of [14] show that, on the same TR core, the Bisdu MAC unit is less efficient than the XPulp-NN one at 4-, 8-, and 16-bit precision, we conclude that STAR-MAC outperforms Bisdu as we just proved that STAR-MAC is more area efficient than [29].

VIII. CONCLUSION

In this paper, we presented STAR-MAC, a novel precision-scalable Multiply-and-Accumulate (MAC) unit, and integrated it into the low-power RISC-V Ibex core, to meet the growing need for energy-efficient, low-latency near-sensor inference in IoT applications. We also introduced a custom version of TensorFlow Lite for Microcontrollers (TFLM) and designed a method for generating a TF-Lite FlatBuffer from a trained MP-QNN QKeras model. Using our combined hardware-software approach based on the STAR-based Ibex and a modified version of TFLM, we show an average 68% latency reduction across the four MLPerf Tiny MP-QNN models running on an FPGA-based System-on-Chip, a FlatBuffer smaller by 27%, and a negligible accuracy loss. Moreover, synthesis results on 28-nm CMOS technology revealed that the STAR-based Ibex is the most energy-efficient processors per unit area among state-of-the-art solutions, with a peak of 175.7 GOPS/W/mm^2 at 4-bit. It also showed little to no accuracy loss, and a minimal overhead in area (+12.2%) and power (+7.3%) over the original Ibex.

REFERENCES

[1] M. Rakka, M. E. Fouda, P. Khargonekar, and F. Kurdahi, "A review of state-of-the-art mixed-precision neural network frameworks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 46, no. 12, pp. 1–20, Dec. 2024.

[2] L. Mei, M. Dandekar, D. Rodopoulos, J. Constantin, P. Debacker, R. Lauwereins, and M. Verhelst, "Sub-word parallel precision-scalable MAC engines for efficient embedded DNN inference," in *Proc. IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Hsinchu, Taiwan, Mar. 2019, pp. 6–10.

[3] L. Urbinati and M. R. Casu, "High-level design of precision-scalable DNN accelerators based on sum-together multipliers," *IEEE Access*, vol. 12, pp. 44163–44189, 2024.

[4] E. Manca, L. Urbinati, and M. R. Casu, "STAR: Sum-together/apart reconfigurable multipliers for precision-scalable ML workloads," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Valencia, Spain, Mar. 2024, pp. 1–6.

[5] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *Proc. 27th Int. Symp. Power Timing Model., Optim. Simul. (PATMOS)*, Thessaloniki, Greece, Sep. 2017, pp. 1–8.

[6] T. Sipola, J. Alatalo, T. Kokkonen, and M. Rantonen, "Artificial intelligence in the IoT era: A review of edge AI hardware and software," in *Proc. 31st Conf. Open Innov. Assoc. (FRUCT)*, Apr. 2022, pp. 320–331.

[7] S. S. Saha, S. S. Sandha, and M. Srivastava, "Machine learning for microcontroller-class hardware: A review," *IEEE Sensors J.*, vol. 22, no. 22, pp. 21362–21390, Nov. 2022.

[8] P.-E. Novac, G. Boukli Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors*, vol. 21, no. 9, p. 2984, Apr. 2021.

[9] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers, "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors," *Nature Mach. Intell.*, vol. 3, no. 8, pp. 675–686, Jun. 2021.

[10] C. Banbury et al., "MLPerf tiny benchmark," 2021, *arXiv:2106.07597*.

[11] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, "Agile SoC development with open ESP," in *Proc. 39th Int. Conf. Computer-Aided Design*, Nov. 2020, pp. 1–9.

[12] *STAR-TFLM*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/timeless-spark/STAR-TFLM

[13] G. Devic, M. France-Pillois, J. Salles, G. Sassatelli, and A. Gamatié, "Highly-adaptive mixed-precision MAC unit for smart and low-power edge computing," in *Proc. 19th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Toulon, France, Jun. 2021, pp. 1–4.

[14] D. Metz, V. Kumar, and M. Själander, "BISDU: A bit-serial dot-product unit for microcontrollers," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 5, pp. 1–22, Sep. 2023.

[15] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, and J. Koenig, "The rocket chip generator," Dept. Electrical Eng. Comput. Sci. (EECS), University of California Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-17, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[16] V. Camus, L. Mei, C. Enz, and M. Verhelst, "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 4, pp. 697–711, Dec. 2019.

[17] R. Lin, "Reconfigurable parallel inner product processor architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 2, pp. 261–272, Apr. 2001.

[18] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[19] G. Armeniakos, A. Maras, S. Xydis, and D. Soudris, "Mixed-precision neural networks on RISC-V cores: ISA extensions for multi-pumped soft SIMD operations," in *Proc. 43rd IEEE/ACM Int. Conf. Comput.-Aided Design*, New York, NY, USA, Oct. 2024, pp. 1–9, doi: 10.1145/3676536.3676840.

[20] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Grenoble, France, Mar. 2020, pp. 186–191.

[21] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, A. D. Mauro, L. Benini, and D. Rossi, "Dustin: A 16-cores parallel ultra-low-power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 6, pp. 2450–2463, Jun. 2023.

[22] E. Reggiani, A. Pappalardo, M. Doblas, M. Moreto, M. Olivieri, O. S. Unsal, and A. Cristal, "Mix-GEMM: An efficient HW-SW architecture for mixed-precision quantized deep neural networks inference on edge devices," in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Montreal, QC, Canada, Feb. 2023, pp. 1085–1098.

[23] L. Urbinati, "Accelerating quantized DNNs with dedicated hardware accelerators and RISC-V processors using precision-scalable multipliers," Ph.D. dissertation, Dept. Electronics Telecommunications (DET), Politecnico di Torino, Turin, Italy, Jul. 2024. [Online]. Available: https://hdl.handle.net/11583/2990842
[24] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm cortex-M CPUs," 2018, *arXiv:1801.06601*.
[25] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 871–875, May 2020.
[26] M. Rusci, A. Capotondi, and L. Benini, "Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers," in *Proc. Mach. Learn. Syst.*, vol. 2, 2019, pp. 326–335.
[27] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 378, no. 2164, Feb. 2020, Art. no. 20190155.
[28] N. Bruschi, A. Garofalo, F. Conti, G. Tagliavini, and D. Rossi, "Enabling mixed-precision quantized neural networks in extreme-edge devices," in *Proc. 17th ACM Int. Conf. Comput. Frontiers*, Catania, Italy, May 2020, pp. 217–220.
[29] A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "XpulpNN: Enabling energy efficient and flexible inference of quantized neural networks on RISC-V based IoT end nodes," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 3, pp. 1489–1505, Jul. 2021.
[30] *Getting Started With X-CUBE-AI Expansion Package for Artificial Intelligence*. Accessed: Jun. 16, 2025. [Online]. Available: https://www.st.com/resource/en/usermanual/um2526-getting-started-with-xcubeai-expansion-package-for-artificial-intelligence-ai-stmicroelectronics.pdf
[31] *Optimized C-Kernel Configurations*. Accessed: Jun. 16, 2025. [Online]. Available: https://wiki.st.com/stm32mcu/wiki/AI:Deep_Quantized_Neural_Network_support#Optimized_C-kernel_configurations
[32] *N2D2*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/CEA-LIST/N2D2
[33] *LiteRT for Microcontrollers–Operation Support*. Accessed: Jun. 16, 2025. [Online]. Available: https://ai.google.dev/edge/litert/microcontrollers/buildconvert
[34] *CMSIS-NN—Quantization Specification*. Accessed: Jun. 16, 2025. [Online]. Available: https://arm-software.github.io/CMSIS-NN/latest/index.html
[35] *CMSIS NN*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/ARM-software/CMSIS-NN/tree/411ff0d7607aa9eccb7e683b81c8f1bc7c086df2
[36] T. Bannink, A. Bakhtiari, A. Hillier, L. Geiger, T. de Bruin, L. Overweel, J. Neeven, and K. Helwegen, "Larq compute engine: Design, benchmark, and deploy state-of-the-art binarized neural networks," 2020, *arXiv:2011.09398*.
[37] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design*, 4th ed., Reading, MA, USA: Addison-Wesley, 2011, ch. 11.
[38] *RISC-V ISA Manual*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/tag/20240411
[39] *RISC-V GNU Compiler Toolchain*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain
[40] T. Jia, P. Mantovani, M. C. D. Santos, D. Giri, J. Zuckerman, E. J. Loscalzo, M. Cochet, K. Swaminathan, G. Tombesi, J. J. Zhang, N. Chandramoorthy, J.-D. Wellman, K. Tien, L. Carloni, K. Shepard, D. Brooks, G.-Y. Wei, and P. Bose, "A 12 nm agile-designed SoC for swarm-based perception with heterogeneous IP blocks, a reconfigurable memory hierarchy, and an 800MHz multi-plane NoC," in *Proc. IEEE 48th Eur. Solid State Circuits Conf.*, Milan, Italy, Sep. 2022, pp. 269–272.
[41] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020, *arXiv:2004.09602*.
[42] *Qkeras-Mod*. Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/LucaUrbinati44/qkeras-mod.git
[43] *TensorFlow Lite for Microcontrollers*. Accessed: Jun. 16, 2025. [Online]. Available: https://ai.google.dev/edge/litert/microcontrollers/overview
[44] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

EDWARD MANCA (Graduate Student Member, IEEE) received the B.Sc. degree in electronics engineering and the M.Sc. degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2021 and 2023, respectively, where he is currently pursuing the Ph.D. degree. His research interests include digital architectures to accelerate machine learning algorithms in embedded/edge devices, architectures for computational alternatives to the von Neumann paradigm, and neural network quantization and model optimization for resource-constrained applications.

LUCA URBINATI (Member, IEEE) received the B.Sc. degree in electronics and telecommunications from the Universitá di Bologna, Bologna, Italy, in 2017, the M.Sc. degree in electronic engineering from the Politecnico di Torino, Turin, Italy, in 2019, and the Ph.D. degree in the hardware acceleration of quantized neural networks. He is currently a Researcher with the National Research Council of Italy (CNR), Italy, with interests in digital integrated architectures for artificial intelligence and machine learning. He has collaborated in various disciplines, including food safety, embedded systems, and high-level synthesis.

MARIO R. CASU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications engineering from the Politecnico di Torino, Turin, Italy, in 2001. He is currently an Associate Professor with the Politecnico di Torino. His research interests include systems-on-chip with specialized accelerators, system-level design and design methodology for FPGAs and ASICs, and embedded machine learning. He is also interested in the design of circuits, systems, and platforms for industrial applications, such as biomedical, automotive, and food. His past work focused mostly on latency-insensitive design of systems-on-chip (SoC) and networks-on-chip. He regularly serves in the Technical Program Committee for international conferences, such as DAC, ICCAD, and DATE.