IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 February 2024, accepted 17 March 2024, date of publication 21 March 2024, date of current version 28 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3380472

RESEARCH ARTICLE

High-Level Design of Precision-Scalable DNN Accelerators Based on Sum-Together Multipliers

LUCA URBINATI, (Graduate Student Member, IEEE), AND MARIO R. CASU, (Senior Member, IEEE)
Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Luca Urbinati (luca.urbinati@polito.it)

ABSTRACT Precison-scalable (PS) multipliers are gaining traction in Deep Neural Network accelerators, particularly for enabling mixed-precision (MP) quantization in Deep Learning at the edge. This paper focuses on the Sum-Together (ST) class of PS multipliers, which are subword-parallel multipliers that can execute a standard multiplication at full precision or a dot-product with parallel low-precision operands. Our contributions in this area encompass multiple aspects: we enrich our previous comparison of SoA ST multipliers by including our recent radix-4 Booth ST multiplier and two novel designs; we extend the explanation of the architecture and the design flow of our previously proposed ST-based PS hardware accelerators designed for 2D-Convolution, Depth-wise Convolution, and Fully-Connected layers that we developed using High-Level Synthesis (HLS); we implement the uniform integer quantization equations in hardware; we conduct a broad HLS-driven design space exploration of our ST-based accelerators, varying numerous hardware parameters; finally, we showcase the advantages of ST-based accelerators when integrated into System-on-Chips (SoCs) in three different scenarios (low-area, low-power, and low-latency), running inference on MP-quantized MLPerf Tiny models as case study. Across the three scenarios, the results show an average latency speedup of 1.46x, 1.33x, and 1.29x, a reduced energy consumption in most of the cases, and a marginal area overhead of 0.9%, 2.5% and 8.0%, compared to SoCs with accelerators based on fixed-precision 16-bit multipliers. To sum up, our work provides a comprehensive understanding of ST-based accelerators' performance in an SoC context, paving the way for future enhancements and the solution of identified inefficiencies.

INDEX TERMS Deep learning, hardware accelerators, high-level synthesis, mixed-precision quantization, precision-scalable MAC unit, sum-together multipliers.

I. INTRODUCTION

In the context of Deep Learning (DL) at the edge, quantization is an established method for reducing memory footprint and bandwidth, saving energy, and performing faster inference when dealing with Deep Neural Networks (DNNs) on resource limited devices [1]. Recently, there has been a growing interest in academia and industry towards Mixed-Precision Quantization (MPQ) [2]. This technique leverages the different sensitivity to quantization of each DNN layer [3], [4] to search for the optimal number of activation and weight bits for each individual layer, enabling accuracy vs latency and accuracy vs energy trade-offs [5].

To take advantage of MPQ, several Precision-Scalable (PS) multipliers, Multiply-and-Accumulate (MAC) units, and DNN accelerators have been recently proposed targeting Machine-Learning (ML) workloads [6], [7], [8], [9], [10], [11], [12]. In this paper, we focus on a particular family of PS multipliers called Sum-Together (ST). These are special reconfigurable subword-parallel multipliers that, depending

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

on the selected configuration, not only perform N multiplications in parallel among multiple low-precision operands, but also *sum together* the results of these N multiplications within the multiplier itself, i.e., without requiring external addition. At full precision, they perform $N = 1$ standard multiplications (e.g., on 16 bits), whereas at reduced precision they compute $N = 2$ or 4 parallel dot-products using the low-precision operands packed in the multiplier's inputs (e.g., operands on 8 or 4 bits). In other words, the bitwidth of the operands is inversely proportional to N (e.g., $16/N$ bits). ST multipliers are well-suited for integration within MAC units [6], [9]. They enable parallel multiplications of low-precision quantized data in a Single Instruction Multiple Data (SIMD) fashion and, at the same time, their *sum together* feature contributes to further speeding up MAC operations, by saving N-−1 MAC additions compared to conventional MAC units. For this reason, ST-based PS MAC units have recently found application in layer-specific DNN hardware accelerators, providing MPQ support and speeding up the overall layer computation by a factor up to N [8], [10], [11].

The contributions of this paper in this field cover multiple aspects:

1) We enrich our previous performance, power, and area (PPA) comparison of SoA ST multipliers [8] by introducing two novel designs. The first, which we name BW-ADD, is an improved Baugh-Wooley (BW) multiplier with a modified final adder that provides a shorter critical path than the original Ripple Carry Adder (RCA) used in [6]. The second, which we call HLS ST, is an ST multiplier derived from High-Level Synthesis (HLS). Indeed, one of our objectives is to evaluate the capability of HLS to generate a competitive ST multiplier in terms of PPA compared to manually-designed Register-Transfer Level (RTL) implementations. We also add to the PPA comparison our radix-4 Booth ST multiplier, recently presented in [7].
2) We provide an extended explanation of the architecture of our ST-based PS hardware accelerators for 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv) and Fully-Connected (FC) layers, developed using HLS and previously proposed in [8], [10], and [11], respectively. Specifically, we present a comprehensive overview of our accelerators design flow, from C/C++ to final hardware implementation.
3) We also add the hardware support for uniform integer quantization (UIQ) [1], [13], [14] to quantize the output activations, which was not present in our previous accelerators' designs.
4) We perform an extensive design space exploration (DSE) for our ST-based accelerators using HLS. This involves varying many knobs, including parallelism, clock frequency, and especially the type of ST multiplier, which is a novelty of this work with respect to [8].
5) We illustrate the advantages achieved by our ST-based accelerators in terms of reduced latency and energy consumption, comparing them to accelerators equipped with non-ST fixed-precision 16-bit multipliers, which we call *standard* accelerators and *standard* multipliers, respectively. For this assessment, we integrate the accelerators into System-on-Chips (SoCs) under three different scenarios (low-area, low-power, and low-latency) and, as case study, we execute the models of the MLPerf Tiny benchmark [15], previously quantized in mixed-precision (MP) with a custom version of QKeras [16] for which we release the source code.

The results of the PPA comparison of ST multipliers showed that architectures having dedicated multipliers for each precision configuration tend to be less area-efficient than those employing a single but reconfigurable multiplier. Moreover, there is not a single winner that satisfy all PPA scenarios, but rather a set of optimal ST multipliers depending on the specific PPA constraints.

The results of the execution of the four MP-quantized MLPerf Tiny networks, using SoCs integrating ST-based accelerators tailored to different PPA scenarios (i.e., low-area, low-power, and low-latency), revealed: an average inference latency speedup, across the four models, of 1.46x, 1.33x, and 1.29x, respectively; a reduced average energy consumption, in most of the cases; and a marginal area overhead compared to SoCs equipped with standard accelerators.

The article is structured as follows. In Sec. II we present the related work, whereas in Sec. III we provide some background on DNN quantization, UIQ and the four MLPerf Tiny models. In Sec. IV we describe the concept of ST multiplier, we outline the architectures of the SoA ST multipliers, and we present the newly proposed ones. In Sec. V we detail the working principle and hardware architecture of our ST-based DNN accelerators, whereas in Sec. VI we describe the accelerators design flow. Finally, in Sec. VII we present the results in three parts: in the first we compare the ST multipliers in terms of PPA; in the second we report the Pareto-optimal accelerators, resulting from the HLS-driven DSE, in Latency vs Area and Power vs Area spaces; in the last we showcase the achievable latency speedup and energy reduction of the ST-based accelerators when running inference on MP-quantized MLPerf Tiny models, against standard accelerators.

II. RELATED WORK

Although the definition of ST mode was introduced with the subword-parallel BW ST multiplier of [6], earlier works already proposed reconfigurable multipliers that support both single high-precision multiplications and parallel low-precision dot-products.

The authors of [17] and [18] introduce SIMD extensions to the Instruction Set Architecture (ISA) of a RISC-V processor featuring a multiplication unit that behaves like an ST multiplier.

In [19], a general-purpose systolic array for DL is proposed. It is made of reconfigurable Fusion Units (FUs) that exploit low-precision multipliers by dynamically merging or

keeping their results separate. The architecture of these FUs falls within the *divide-and-conquer* (D&C) category, as per the taxonomy outlined in [9]. To this category also belong the optimized versions in [12] and [20], and their ancestor in [21].

In [22], the authors present a reconfigurable fixed-point multiplier originally designed for digital signal processing (DSP) applications.

In [9], various PS MAC unit (PSMAC) architectures are benchmarked and categorized in subword-parallel, D&C and bit-serial. However, in [6] and [19] are the only ST-based PSMACs considered here. In [8], instead, we compared all the main previously described SoA ST multipliers in PPA.

Recently, we have also contributed to the SoA of ST multipliers with a subword-parallel radix-4 Booth architecture that requires a light-weight reconfiguration logic [7].

Regarding ST-based DNN accelerators, there are a few examples in the literature: in [19] the authors proposed a general-purpose systolic array architecture, in [6] the authors describe their implementation of an FC kernel, whereas we derived 2D-Conv [11], DW-Conv [10] and FC [8] layer-specific accelerators using an HLS flow. In particular, in [8] we also carried out a DSE varying several hardware knobs, from clock frequency to HLS directives, to explore a wide range of Pareto-optimal solutions in area, power or latency. The authors of [9] and [12] have already conducted an exhaustive comparison of various PS hardware accelerators, including [6], [19], [23], [24], [25], and [26]. To the best of our knowledge, no other works have focused on employing HLS techniques for the development of ST-based hardware accelerators and PS accelerators in general.

In this work, we enrich the SoA portfolio by introducing two novel ST multipliers (a BW multiplier with a modified final adder and an ST multiplier derived from a functional C/C++ description by an HLS tool) and we perform a comprehensive PPA analysis of all the SoA ST multipliers. We then derive ST-based PS DNN accelerators, like in [8]; however, a distinctive feature of our work is the support for UIQ for the quantization of the final result, which is not mentioned in any of the previously cited accelerators. In this regard, we propose an accelerator design flow that includes minimizing the bitwidths of the fixed-point variables required by the UIQ formulas. We also expand our previous DSE [8] by introducing new hardware knobs. One of these is the selection of the type of ST multiplier inside the accelerators' MAC units, which can be chosen among all the manually-designed RTL descriptions of SoA ST multipliers (like in [8]), and also among the ST multipliers inferred by the HLS tool from a high-level description. Furthermore, we show the latency and energy benefits of ST-based accelerators, against equivalent accelerators based on standard multipliers, when running entire MP-quantized DNNs. Such comparison, except for our previous work that focused solely on isolated DNN layers [10], [11], has not been extensively examined in the literature.

III. BACKGROUND

A. DEEP NEURAL NETWORKS' QUANTIZATION

The quantization of DNNs is now a common practice that decreases the numerical precision of weight parameters and activation values of neural networks layers. This process reduces the model size, lowering memory requirements to store weigths and activations, as multiple low-precision feature maps and weights can be efficiently packed into the same memory word [14]. For the same reason, it also reduces data transfers costs. Additionally, quantization can improve inference latency, throughput and energy by taking advantage of high-throughput integer instructions, such as SIMD instructions in microprocessors [18], or specialized hardware operators like subword-parallel ST multipliers [7].

In this paper we focus on UIQ, even though various other quantization techniques exist [1]. This choice is driven by the simple mathematical formulation, the availability in common ML frameworks (e.g., TensorFlow Lite), its efficient mapping on existing hardware (e.g., on 8-bit microcontrollers), and thus its widespread adoption on embedded devices for non-extreme quantization (> 2 bits) [1], [13], [27]. Moreover, when it comes to ASIC implementation, integer/fixed-point math pipelines are more efficient in terms of silicon area and power consumption when compared to floating-point (FP) ones [28], not to mention the faster execution times. In the following, we introduce the UIQ mathematical background in the context of DNNs, borrowing some definitions from [13] and [14]. Notice that, since we target ST-based accelerators only for the inference phase of DNNs, our focus is only on UIQ for inference, and not for training.

1) UNIFORM INTEGER QUANTIZATION

Given a set of real numbers in the *real range* $[\alpha, \beta]$ (e.g., a tensor with a high-precision FP format like FP32), UIQ maps each $x \in [\alpha, \beta]$ to an integer value $x_q \in [\alpha_q, \beta_q]$ represented uniformly on b bits, where $[\alpha_q, \beta_q]$ is the *quantized range*: for asymmetric or symmetric signed integers it is equal to $[-2^{b-1}, 2^{b-1}-1]$ or $[-2^{b-1}-1, 2^{b-1}-1]$, respectively; for unsigned integers it is $[0, 2^b - 1]$. The process of quantization is defined as:

$$x_q = \text{clip}\Big(\text{round}\big(\frac{1}{s}x + z\big), \alpha_q, \beta_q\Big) \tag{1}$$

where s is the *scaling factor*, z is the *zero-point* (i.e., the integer value to which the real value zero is exactly represented), *round* is the rounding function (e.g., round-to-nearest), and *clip* keeps the output range within the quantized range by saturating the outliers. In turn, s and z are defined from the chosen real and quantized ranges as:

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{2}$$

$$z = \text{round}\big(\frac{\beta\alpha_q - \alpha\beta_q}{\beta - \alpha}\big) \tag{3}$$

The opposite operation, which brings back x_q to the real range, is defined as:

$$\hat{x} = s(x_q - z) \tag{4}$$

where $\hat{x}$ is the closest real value (but not necessarily equal) to the original x, because rounding and clipping functions may introduce an irrecoverable error.

The quantization mapping discussed so far, with asymmetric ranges and $z \neq 0$, is known as *affine* quantization. Instead, when both ranges are symmetric, z becomes zero and (1) performs only the scale transformation. In this case, the quantization mapping is commonly known as *scale* or *symmetric* [29] quantization. Moreover, when s is a unique scalar value for all the channels of a tensor, quantization is referred to as *per-layer*; instead when s is a one-dimensional vector of scalars, each corresponding to a different channel of a tensor, quantization is called *per-channel*.

2) INTEGER-ONLY DNN KERNELS

Now consider the expression of an FC layer:

$$Y_k = b_k + \sum_{c=1}^{C} X_c W_{c,k} \qquad \forall\, k \in [1, K] \tag{5}$$

where $X \in \mathbb{R}^C$ is the input vector of neurons, $W \in \mathbb{R}^{K \times C}$ is the weight matrix, $b \in \mathbb{R}^K$ is the bias array, $Y \in \mathbb{R}^K$ is the output array, C and K are the number of input and output activations processed by the FC layer, respectively. By applying (4) to each of the four real variables in (5), setting their own quantized ranges a priori, and moving the quantized output array $Y_{q,k}$ to the left hand side, we obtain the quantized FC expression valid for the k-th output activation:

$$\begin{aligned} Y_{q,k} = \underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,k} - z_b)}_{\text{(b)}} + \frac{s_X s_W}{s_Y}\Bigg[\underbrace{\left(\sum_{c=1}^{C} X_{q,c} W_{q,c,k}\right)}_{\text{(c)}} \\ \underbrace{- \left(z_W \sum_{c=1}^{C} X_{q,c}\right)}_{\text{(d)}} \underbrace{- \left(z_X \sum_{c=1}^{C} W_{q,c,k}\right)}_{\text{(e)}} \underbrace{+ C z_X z_W}_{\text{(f)}} \Bigg] \\ \forall\, k \in [1, K] \end{aligned} \tag{6}$$

where X_q, W_q, b_q, Y_q are the integer values; s_X, s_W, s_b, s_Y are the scaling factors; and z_X, z_W, z_b, z_Y are the zero-points, associated with X, W, b, Y, respectively. Term (c) in (6) is the integer dot product, i.e., the core of the computation, instead term (d) introduces an overhead that causes a performance penalty. Both of them must be computed online because they depend on X_q, which is known only at runtime. On the contrary, terms (a), (b), (e), and (f) are constant, thus can be computed offline. Notice that in case of scale quantization for weights and affine quantization for activations, which is a common practice in the literature [13], [14], z_W and z_b become null, and so also terms (d) and (f), while (b) simplifies. This is also our assumption in this work. The result of (6), before being assigned to Y_q, is also rounded and clipped to fit the desired output quantized range of Y_q (not shown in the formula for better readability).

The mathematical derivations of the integer-only kernels for 2D- and DW-Conv closely follow that of FC. We report them in Appendix A. Hereafter, we will refer to (14), (16), and (6) as the *UIQ formulas*.

Now we focus on the integration of the rectified linear unit (ReLU) into the expressions of the integer-only kernels. In fact, to optimize inference on DNNs in embedded devices, some adjacent DNN layers can be typically combined into a single one. This operation, called *Layer Fusion*, is usually performed between convolutional/fully-connected layers and the Batch Normalization (BN) or activation layers (e.g., ReLU), and can be applied to both FP and quantized models. Since our ST-based accelerators support layer fusion with ReLU, as elaborated in Sec. V-B, we explain here the fusion process considering an FC layer with a subsequent ReLU layer. We choose ReLU because it stands out as the most common activation function when it comes to efficient hardware implementations of DNNs. By applying the ReLU non-linearity to the FP output Y_k of (5), we derive the expression of the FP FC-ReLU fused layer:

$$R_k = \begin{cases} 0 & \text{if } Y_k < 0 \\ Y_k & \text{if } Y_k \geq 0 \end{cases} \qquad \forall\, k \in [1, K] \tag{7}$$

where R_k is the k-th output of the ReLU layer. By repeating the same steps that brought to the derivation of (6) from (5)—applying (4) to each real variable of (7), setting their quantized ranges, and moving the quantized ReLU output $R_{q,k}$ to the left hand side—we obtain the quantized FC-ReLU fused layer valid for the k-th ReLU element:

$$R_{q,k} = \begin{cases} z_R & \text{if } T < 0 \\ z_R + s_R \cdot T & \text{if } T \geq 0 \end{cases} \qquad \forall\, k \in [1, K] \tag{8a}$$

where

$$\begin{aligned} T = s_b(b_{q,k} - z_b) + s_X s_W \Bigg[\left(\sum_{c=1}^{C} X_{q,c} W_{q,c,k}\right) \\ - \left(z_W \sum_{c=1}^{C} X_{q,c}\right) - \left(z_X \sum_{c=1}^{C} W_{q,c,k}\right) + C z_X z_W \Bigg]. \end{aligned} \tag{8b}$$

s_R and z_R are the scaling factor and the zero-point associated to $R_{q,k}$, whereas all the other variables are the same of those that appear in (6). Notice that $R_{q,k}$ undergoes a round-and-clip operation, not shown for clarity in (8), to fit into the desired quantized range of the ReLU layer.

The expressions for the quantized 2D-Conv-ReLU and DW-Conv-ReLU fused layers can be obtained through the same steps shown here for the quantized FC-ReLU.

B. MLPERF TINY BENCHMARK

The Machine Learning Performance Benchmark (MLPerf) is a widely recognized set of benchmarks in the field

of ML created by the collaborative effort of more than fifty organizations from both academia and industry [15]. In particular, the *Tiny* benchmark is a suite of four lightweight ML models representing real-world applications: Visual Wake Words (VWW), Image Classification (ImgClass), Keyword Spotting (KS), and Anomaly Detection (AD). MLPerf Tiny was designed to assess the performance of edge devices and ultra-low-power tiny ML systems with a limited energy, memory and/or computational power budget (such as mobile phones, microcontrollers, Internet of Things devices), by measuring accuracy, latency and energy during inference on those four ML models. In this respect, MLPerf Tiny is also a competition that encourages innovation in the field of Tiny ML [30]. For these reasons, each application not only comes with its own dataset for development and testing, but also with a dedicated performance evaluation dataset (*Perf test set*).

1) VISUAL WAKE WORDS

The VWW dataset [31] is a collection of 109619 96×96 RGB images which contain persons or not-persons, derived from the MSCOCO 2014 dataset [32]. The use-case of this dataset is for a device to wake up when a person is present, covering smart doorbell and occupancy applications. The model to use with this dataset is a smaller version of MobilenetV1, [33] that we define *MobileNetV1Tiny*.

2) IMAGE CLASSIFICATION

The ImgClass benchmark uses the CIFAR-10 dataset [34], which consists of 60000 32 × 32 RGB images belonging to 10 unique classes of 6000 images each. The use-case is for compact vision systems, including manufacturing, Internet-of-Thing sensor nodes, and autonomous agents and vehicles. The model to use is a custom ResNetV1 [35] that we define *ResNetV1Tiny*, which has no pooling layer after the first convolutional layer, fewer residual stacks, and lower dimension of filters and convolution strides than the original ResNetV1.

3) KEYWORD SPOTTING

The KS benchmark uses a large collection of English words, pronounced by persons with various accents, and derived from the Speech Commands v2 dataset [36]. It contains twelve classes: ten with keywords (down, go, left, no, off, on, right, stop, up, yes), one with background noises and one with silence. The use-case is for human-machine interaction, including wakeword detection and remote control of smart devices by voice. The benchmark's target network is the small Depth-wise Separable Convolutional Neural Network (*DS-CNN*) of [37].

4) ANOMALY DETECTION

This benchmark uses one of the six machine types present in the DCASE2020 competition's dataset [38], the toy-car machine type (ToyADMOS [39]), which contains single-channel 10-seconds length audio samples recorded from seven different toy cars (1000 each) mixed with environmental noise. The use-case is early detection of machine anomalies, a common industrial problem. The model of this benchmark is the reference implementation of DCASE2020 which is an FC-based autoencoder [38] (thus, we name it *FC-AutoEncoder*). Differently from the other MLPerf Tiny models, the main metric used in AD is not accuracy, but the Area Under The Receiver Operating Characteristics Curve (AUC).

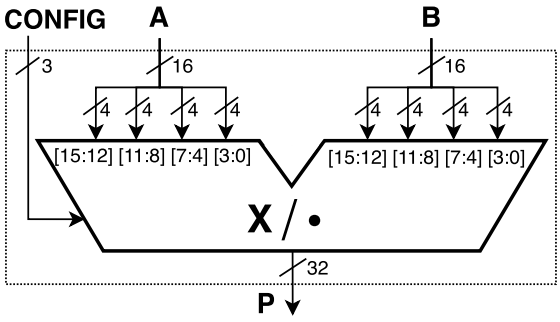

FIGURE 1. Reference ST multiplier, modified from [7]. The 16-bit inputs (A and B) are partitioned in 4-bit chunks to enable multiple operations, as defined by the 3-bit configuration input (*CONFIG*) and shown in Table 1. The *X* / · symbol indicates that the multiplier is capable of multiplication and *dot-product* operations, depending on the configuration.

TABLE 1. Supported precision configurations and operations of the reference ST multiplier of Fig. 1. The last three configurations correspond to dot-product operations at low precision.

CONFIG	ST multiplier's output
16×16 (000b)	$P_{[31:0]} = A_{[15:0]} \times B_{[15:0]}$
16×8 (100b)	$P_{[31:0]} = A_{[15:0]} \times B_{[7:0]}$
8×8 (010b)	$P_{[16:0]} = A_{[15:8]} \times B_{[7:0]} + A_{[7:0]} \times B_{[15:8]}$
8×4 (011b)	$P_{[16:0]} = A_{[15:8]} \times B_{[3:0]} + A_{[7:0]} \times B_{[11:8]}$
4×4 (001b)	$P_{[9:0]} = A_{[15:12]} \times B_{[3:0]} + A_{[11:8]} \times B_{[7:4]} + A_{[7:4]} \times B_{[11:8]} + A_{[3:0]} \times B_{[15:12]}$

IV. SUM-TOGETHER MULTIPLIERS

The new ST multipliers that we introduce, as well as all the others that we analyze in this work, have I/O signals and behave as the reference component described in Fig. 1 and Table 1. Depending on the *CONFIG* configuration signal, this can perform one 16 × 16.16×8 multiplication, or two 8 × 8.8×4 or four 4 × 4 dot-products in parallel, using the signed operands packed in the 16-bit inputs *A* and *B*. Depending on the configuration, a subset or the entire 32 bits of the multiplier's output *P* contain the operation result.

We focus on these precisions for the following reasons. In applications that require utmost accuracy, a common

choice is to use 16 bits to quantize activations and weights. Some examples are safety-critical applications, such as image segmentation in foggy environments for autonomous driving [5]; others are image processing applications that work with high-resolution satellite images, or high dynamic range (HDR) images and super-resolution [27]. 8 bits is the default precision to quantize DNNs while avoiding performance degradation [27] and is therefore the most commonly used. When smaller bitwidths for inputs and weights are needed, quantization techniques targeting 4 bits already provide an acceptable tradeoff between model size reduction and retained performance for most applications [1], [40]. Instead, when dealing with extreme low-bit quantization (< 4 bits), existing methods incur a serious accuracy loss compared to the baseline, unless very extensive tuning and hyperparameter search is performed. Hence, this is still an active line of research [1]. In light of these motivations, we work with ST multipliers that support operands with precision between 16 and 4 bits.

Regarding the asymmetric configurations (i.e., 16×8 and 8×4), we support them because they enable efficient packing of lower-precision operands, such as DNN weights, without compromising the precision of other operands, like DNN activations. Thus, they contribute to reducing the memory footprint of ML models. These configurations are used in SoA ML accelerators and processors [9], [18], and can also be found in commercial ML frameworks such as TFLite Micro.[1]

In the following, we first describe the architectures of the SoA ST multipliers as proposed in the literature and emphasize the differences between these and our re-implemented versions. Indeed, since the original SoA ST multipliers support a broad range of bitwidths for input and weights, we introduce minor modifications to align their configurations with the reference ST mentioned in Table 1. This is important to guarantee fair comparisons in all our experiments of Sec. VII. Lastly, we present the two newly proposed ST multipliers.

A. SOA ST MULTIPLIERS

The original subword-parallel BW multiplier of [6] is composed of a reconfigurable partial product matrix (PPM) and a final RCA [41]. The PPM can be reconfigured to compute one 16×16 multiplications or $16/m$ dot-products at $m = 8, 4$, or 2 bit precision in parallel. Our re-implementation of [6] (made with a structural RTL description) is reported on the left side of Fig. 2(a). From top to bottom, it has the same architecture of the original version. We also draw, for clarity, the output concatenation block (*&*), which merges the least significant output bits coming from the PPM with the most significant ones exiting from the final adder. However, in our version we introduce the following modifications. First, we remove the 2-bit support from the PPM, since we use precision between 4 and 16 bits as motivated before. This can be seen from the precision of the main building block of the PPM, which is a 4×4 BW multiplier. Second, in low-precision configurations, we right shift the final output to the least-significant bit (LSB) position, sign-extending it to 32 bits (*right shift & ext* block). On the right side of Fig. 2(a), we illustrate how the PPM is reconfigured in the five operating modes of Table 1. In the 16×16 and 16×8 modes, all the PPs of the PPM contribute to the multiplier's output P and the result is represented on 32 bits, making valid—i.e., yellow in Fig. 2(a)—the entire result P. At lower precision, only the yellow PPs on the left-to-right diagonal of the PPM remain active and behave like two 8×8 (8×8 / 8×4 in Fig. 2(a)) or four 4×4 (4×4 in Fig. 2(a)) BW multipliers, respectively. These PPs produce the valid (yellow) output bits of P, which are less than 32 in this case and require the alignment to the LSB position. The remaining grey PPs are gated, using AND gates, and generate the invalid (grey) output bits.

The multiplication unit of RI5CY [17], a RISC-V processor featuring a SIMD ISA extension, comprises a standard 32-bit integer multiplier, a 32-bit fixed-point multiplier and two subword-parallel dot-product units. These units accept two 16-bit or four 8-bit operands (packed in one 32-bit register) and accumulate the 32-bit result in one cycle, hence performing simultaneously up to four multiplications and accumulations. The architecture of the two dot-product units consists of either two 17-bit multipliers or four 9-bit multipliers, respectively, followed by a compression tree. In [18], the same authors extend the multiplication unit with other two subword-parallel dot-product units supporting 4- and 2-bit operands, respectively. They also added the ISA support for some asymmetric configurations (8×4, 4×8 and 16×2). In this work, we implement the high-precision fixed-point multiplier and the two low-precision dot-product units of [17] as three mutually exclusive datapaths in a single design, scaling their precision to 16, 8 and 4 bits, respectively. Our re-implementation, illustrated in Fig. 2(b), uses a behavioral RTL description, as the authors of [17] declared that it gives the synthesizer the maximum optimization freedom.

The Fusion Unit of Bit Fusion [19] dynamically composes and decomposes 2-bit multipliers (called BitBricks) through a shift-and-add logic. It supports one 8×8, two 4×8, four 4×4, four 2×8, eight 2×4, sixteen 2×2 input/weight multiplications in one clock cycle. Several optimizations to the original work of [19] are proposed in [12] and [20], which reduce complexity and reconfigurability overhead of the shift-and-add logic at the expense of a lower number of supported input/weight precisions (2×2, 4×4, 8×8). However, the ancestor of all these D&C architectures is the reconfigurable and parallel inner-product processor of [21]. This uses larger BitBricks on 4 or 8 bits and a higher input precision. In fact, each of the two input operands can accommodate one 64-bit, four 32-bit, sixteen 16-bit, or sixty-four 8-bit items. It also maintains a fixed bitwidth for the two input operands, ensuring a constant memory bandwidth

[1] An example of a TFLite Micro kernel for 2D-convolution supporting asymmetric configurations: https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/kernels/conv.cc. Accessed on: Jan 19, 2024.

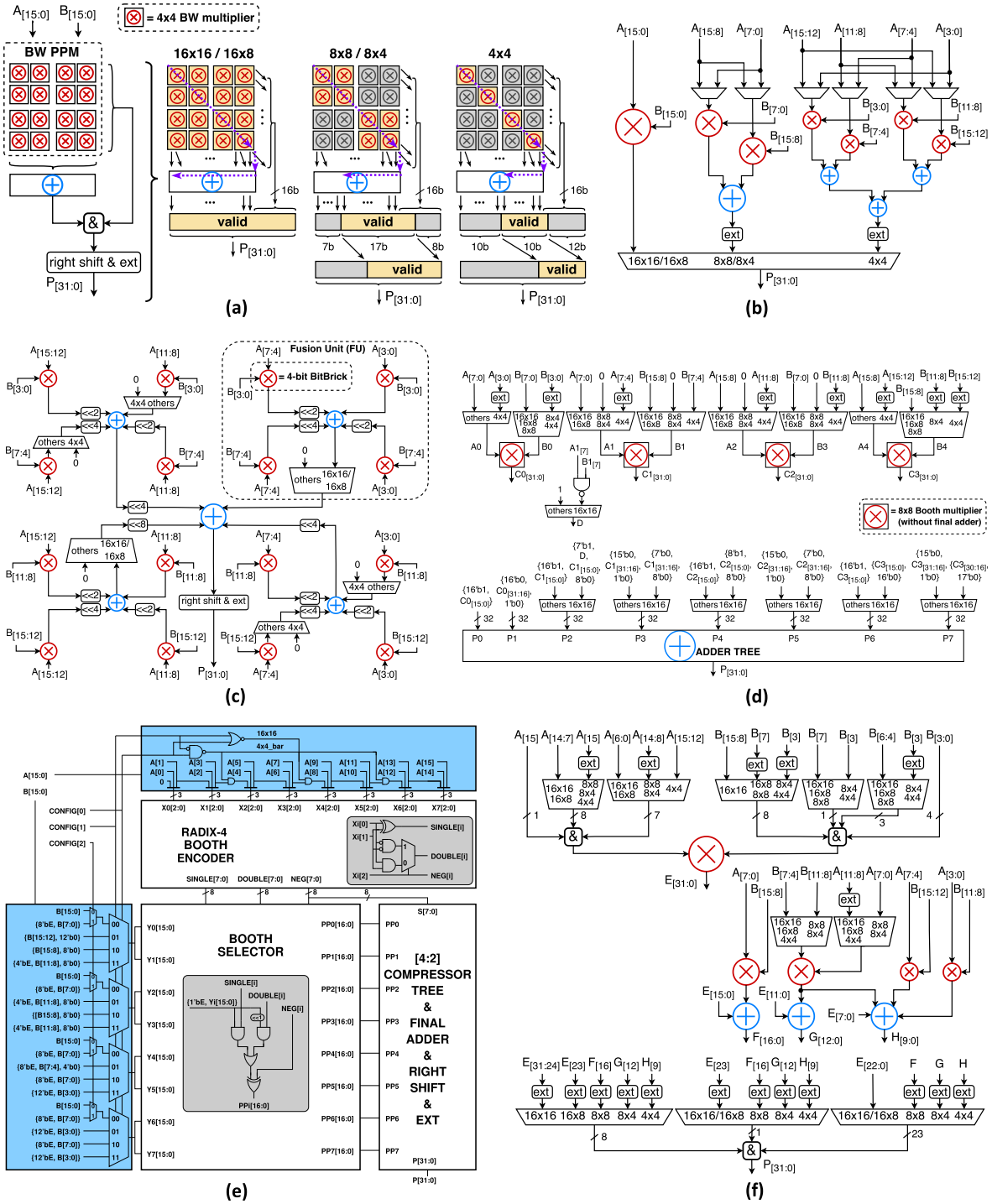

FIGURE 2. (a) Our version of BW ST [6]: architecture overview (left), PPM reconfiguration (right). (b) Our version of ST multiplier [17]: 16-bit high-precision multiplier (left), 8-bit and 4-bit low-precision dot-product units (middle, right). (c) Our version of D&C ST [21]: four FUs with four 4-bit BitBricks each, interconnected by shift-and-add logics. (d) Our version of ST multiplier [22]: four 8-bit Booth multipliers interconnected by muxes ending with an adder tree. (e) Radix-4 Booth ST (as in [7]): reconfiguration logic (blue), 16-bit Booth multiplier (white and gray). (f) HLS ST derived from HLS (proposed in this paper): four multipliers and three adders interconnected by a network of muxes and concatenations.

across different configurations. This contrasts with the D&C architectures in [12], [19], and [20], which suffer from memory bandwidth explosion at reduced precision, as noted in [9]. Among these D&C ST multipliers we implement the one from [21] for a fair comparison with the other SoA ST multipliers on an equal memory bandwidth basis, avoiding the problem of bandwidth explosion. In particular, we re-implement it with 4-bit BitBricks to support 16, 8 and 4-bit precision, as shown in Fig. 2(c): four FUs based on four 4-bit BitBricks each, interconnected by shift-and-add

logic. We also right-shift its output to the LSB position and sign-extend it to 32 bits in low-precision modes (*right shift & ext* block as in [6]).

The reconfigurable fixed-point multiplier of [22] targets DSP applications and consists of four 16-bit Booth multipliers (without final adder), a configurable partial-products compression array and three configurable 33-bit adders. It supports symmetric (one 32 × 32, two 16 × 16 or four 8 × 8) and asymmetric (two 16 × 32) signed/unsigned multiplication operations, and dot product/double dot product operations (one or two 16 × 16±16 × 16 with saturation, one or two 16 × 16±16 × 16.16 without saturation, and one 8 × 8.8×8+8×8.8×8). In our version of [22], we remove the extra logic that is not strictly necessary to implement the reference ST multiplier behavior, such as the saturation logic or the subtraction in the dot-products. Next, we change the way the dot product is computed for all precisions. For example, in configuration 8 × 8 we swap the lower part with the upper part of operand *B*: A[15:8]×B[15:8] ± aL[7:0]×bL[7:0] of [22] becomes A[15:8]×B[7:0] + A[7:0]×B[15:8], as reported in Table 1. We also scale down maximum and minimum precision to 16 and 4 bits, respectively. The resulting architecture, shown in Fig. 2(d), features four 8-bit Booth multipliers connected by a network of multiplexers ending with an adder tree.

The subword-parallel radix-4 Booth architecture of [7] already supports operands at 16, 8, and 4-bit precision, as the reference ST multiplier in Table 1. As illustrated in Fig. 2(e), it is composed of a lightweight reconfiguration logic (in blue) placed between the two input operands and a standard 16-bit Booth multiplier. The latter, drawn in white and gray, features a Wallace's reduction tree with 4:2 compressors and a Carry Propagate Adder with Prefix Network [41]. We implement the architecture of this Booth ST multiplier with a structural description as in our original paper [7].

At last, within all SoA ST multipliers that do not natively support the asymmetric configurations 16 × 8 and 8 × 4 (i.e., [6], [17], [21]), we add a sign-extension logic (not shown in Fig. 2 for better readability) that extends the lower precision operand *B* to either 16 or 8 bits before the actual multiplication operation. For this reason, zero-padding of the low-precision operands is not necessary in any configuration, as these operands always fully utilize all the parallelism of the multipliers' inputs *A* and *B*.

As a final note, we implement all of these SoA ST multipliers as signed multipliers.

B. BW-ADD: A BAUGH-WOOLEY ST MULTIPLIER WITH AN IMPROVED FINAL ADDER

In light of the results of our previous work [7], [8], we observe that the BW ST multiplier [6] is particularly area-efficient at clock frequencies lower than 600 MHz. At higher frequencies, the long diagonal critical path of the BW array and the carry chain of the final 16-bit RCA, highlighted by the purple dotted line in Fig. 2(a), are responsible for a significant area degradation [8], since the logic synthesizer must infer large logic gates to meet the stricter timing constraints. Thus, in this paper we address this problem by letting the logic synthesizer select the most suitable final adder implementation that meets the specified timing constraints with the minimum area, rather than forcing it to use an RCA. We name this multiplier *BW-ADD*. With this change, we expect a lower multiplier's area at high frequency, while remaining unaltered at low frequency, compared to [6].

LISTING 1. The C/C++ source code of the HLS ST multiplier.

```
1 #include <ac_int.h>
2
3 int32 st_multiplier_function(uint3 CONFIG,
4                              int16 A, B) {
5  int32 P;
6  if (CONFIG == 4) { // 16x8
7   P = A*B.slc<8>(0);
8  } else if (CONFIG == 2) { // 8x8
9   P = A.slc<8>(8)*B.slc<8>(0) + A.slc<8>(0)*B.slc<8>(8);
10 } else if (CONFIG == 3) { // 8x4
11  P = A.slc<8>(8)*B.slc<4>(0) + A.slc<8>(0)*B.slc<4>(8);
12 } else if (CONFIG == 1) { // 4x4
13  P = A.slc<4>(12)*B.slc<4>(0) + A.slc<4>(8)*B.slc<4>(4) +
14      A.slc<4>(4) *B.slc<4>(8) + A.slc<4>(0)*B.slc<4>(12);
15 } else { // 16x16
16  P = A*B;
17 }
18 }
```

C. HLS ST: AN ST MULTIPLIER DERIVED FROM HLS

As we present in Sec. V, we use HLS to generate the RTL of our PS DNN accelerators based on ST multipliers starting from a high-level description. To infer a specific implementation of an ST multiplier in the accelerators' MAC units, we force the HLS tool to import its RTL implementation. Usually, this RTL is described manually, as in the case of the SoA ST multipliers of Sec. IV-A. As an alternative, we decide to describe the ST functionality at a high-level and let the HLS tool, which in this work is Siemens Catapult, create automatically its RTL. The source code of this new ST multiplier, which we name *HLS ST*, is listed in Lst. 1. To easily access bit fields from integer data types, we use the method *slc* available in the Catapult C++ library *ac_int.h* (line 1): for example, *A.slc<4>(12)* is a 4-bit subfield from bit 15 down to bit 12 of the int16 signal *A* (line 12).

By inspecting the RTL generated by the HLS tool, which corresponds to the schematic in Fig. 2(f), we notice that it contains one 16-bit, two 8-bit and two 4-bit multipliers, three adders with 8/12/16-bit bitwidth precision, and a network of multiplexers and concatenation blocks (&) that unpacks the 16-bit input operands, distributes them to the multipliers and merges their low-precision results into the final 32-bit output. Moreover, the result of low-precision configurations is already aligned to the rightmost LSB position.

V. ST-BASED HARDWARE ACCELERATORS

A. WORKING PRINCIPLE

We now illustrate the working principle of our three DNN accelerators integrating ST multipliers in their MAC units. Fig. 3 shows the different access patterns (red) that the 2D-Conv, DW-Conv and FC accelerators use to read data

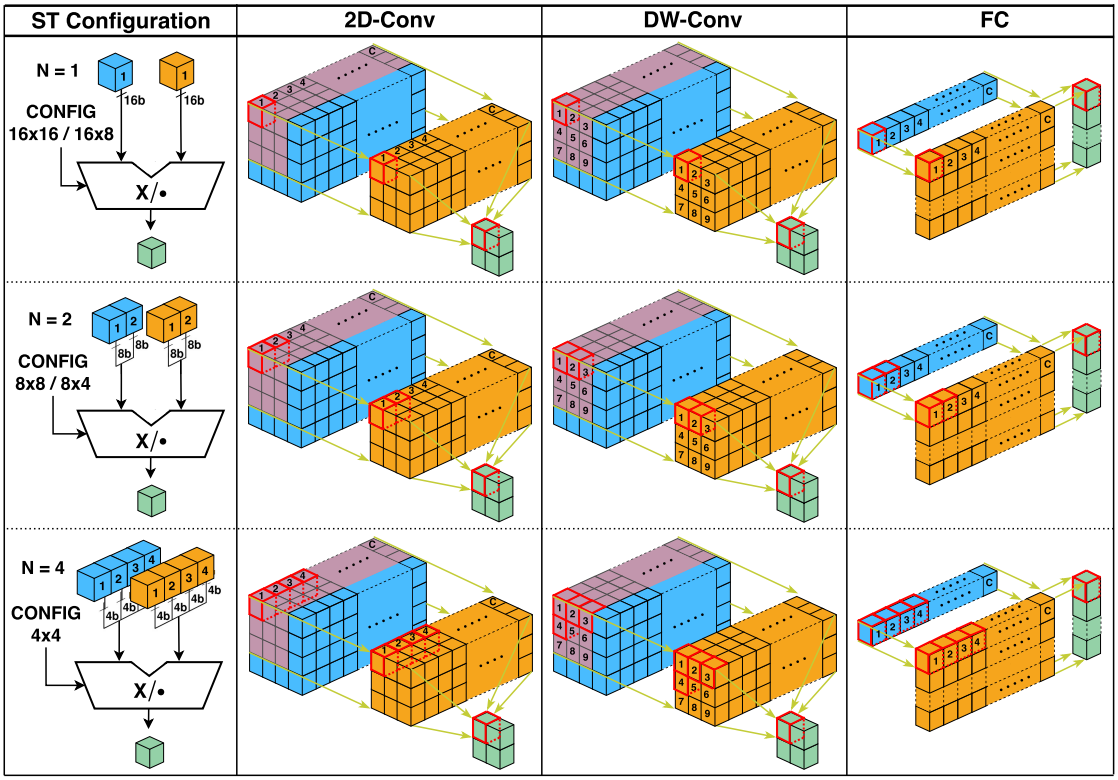

FIGURE 3. Working principle of our ST-based DNN accelerators: 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv) and Fully-Connected (FC).

from the activation (blue) and weight (orange) tensors, and how these data are packed in the 16-bit inputs of the ST multipliers.

1) 2D-CONV ACCELERATOR

For every orange filter with C kernels, a MAC unit of the 2D-Conv accelerator performs the multiplication of the C channels of the blue input tensor with the corresponding weight kernels, and the channel-wise accumulation of these multiplications. At full precision ($N = 1$), the ST multiplier within the MAC unit processes activations and weights from one input channel at a time. Instead, at lower precision the ST multiplier is fed with pairs of activation/weight data from two ($N = 2$) or four ($N = 4$) input channels at a time. This process is highlighted in red in the second column of Fig. 3 and allows to exploit the dot-product feature of the ST multiplier resulting in ideally fewer MAC cycles, which scale as C/N, and lower latency, which scales as $1/N$.

2) DW-CONV ACCELERATOR

In DW-Conv, every output channel is the result of the convolution between the corresponding blue input channel and orange weight kernel, with no accumulation along the channel dimension, as it happens instead in the 2D-Conv case. Therefore, we need to use the ST multiplier in a different way than for 2D-Conv: we can accumulate the partial products between the $N = 1$, 2, or 4 input/weight pairs from the receptive field of the input tensor and the corresponding weight kernel. This new dataflow is reported in red in the third column of Fig. 3.

Compared to 2D-Conv, this accelerator has an overhead that affects the reduction of both MAC cycles and latency, as we show later in Sec. VII-C. This is because the number of accumulations is given by the square of the kernel size (K^2), which is not a multiple of N at lower precision (i.e., $N = 2$ or 4). Let us consider the 3×3 kernel of Fig. 3 as an example. With $N = 2$ or 4, we need five or three iterations, respectively, to accumulate the products of input activations and kernel weights. In the last iteration, however, only one input pair is within the receptive field of the kernel. As a result, we need to feed the ST multiplier with zeros in place of the missing low-precision operands, but this clearly results in under-utilization of the ST hardware. The number of MAC cycles for DW-Conv is $\lceil K^2/N \rceil$ and the latency reduction scales as $\lceil K^2/N/K^2 \rceil$, which typically is greater than $1/N$, with this overhead decreasing as K increases [10].

3) FC ACCELERATOR

The working principle of this accelerator is shown in the last column of Fig. 3. To compute each element of the green output activation array (e.g., the one highligted in red), a MAC unit computes the dot product between the blue array of C input activations and one row of the orange weight matrix. The ST multiplier in the MAC unit takes N pairs

```
void st_based_accelerator(IBUF[], WBUF[], OBUF[], CONFIG) {

  L1: outermost loops (pipelined) { // (2D-Conv, DW-Conv)

    L2: OBUF[] reset loop, executed when neeeded (1st tile)

    L3: outer loop on input channels (2D-Conv), kernel \
    (DW-Conv), or input activations (FC) (pipelined) {

      read A from IBUF[] // (2D-Conv, FC)

      L4: innermost loop (unrolled) {
        read A from IBUF[] // (DW-Conv)
        read B from WBUF[]
        P = st_multiplier_function(CONFIG, A, B);
        OBUF[output_index] += P;
      }

      if (L3 index == (L3 max iterations / N)) {
        break;
      }

    }

    L5: quantization and ReLU loop (pipelined), executed \
    when needed (last tile) {
      Eq. (14) for 2DConv, (16) for DWConv, (6) for FC
    }

  }
}
```

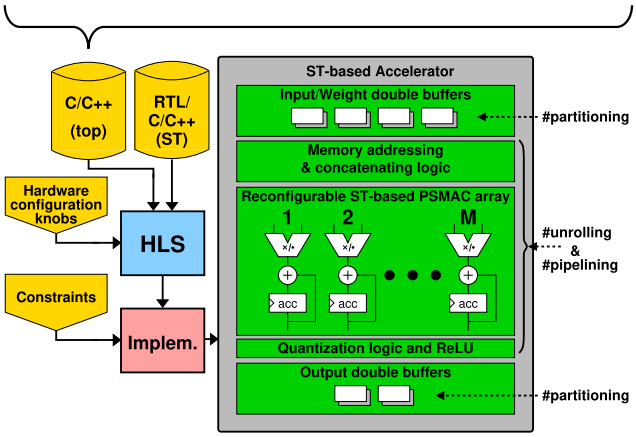

FIGURE 4. General architecture of the ST-based accelerators (bottom right), HLS flow (bottom left), and pseudo-code of the high-level C/C++ description (top) that produces the general architecture.

at a time from the two arrays and either multiplies them in high-precision mode ($N = 1$), or performs a dot-product in low-precision mode ($N = 2$, or 4). Similarly to 2D-Conv, C/N subsequent accumulations are needed to complete the calculation. The process is repeated for every row of the weight matrix, until the green output activation array is complete. As a result, the number of MAC cycles and the corresponding latency scale as C/N and $1/N$, respectively, like in the 2D-Conv case.

B. ACCELERATORS ARCHITECTURE

Our ST-based DNN accelerators share the same general architecture, outlined in the grey rectangle of Fig. 4. It consists of four parts as illustrated later in Secs. V-B1–V-B4: internal buffers (for input, weight, and output data), memory addressing and concatenating logics, reconfigurable ST-based PSMAC array, quantization logic and ReLU. We obtain this architecture using the flow on the left side of Fig. 4, starting from a high-level C/C++ description of the ST-based accelerator (*C/C++ (top)* block) and using HLS techniques to generate the final RTL implementation. We provide a full description of this flow in Sec. VI-C.

Even though it is not the focus of our paper, we assume that the three accelerators share an on-chip global buffer (not shown in Fig. 4), as shown for example in [42] and other more recent papers on ML SoCs [43], [44]. In particular, we assume that the global buffer is large enough to store at least two *tiles* (two is for double buffering) of each of the three relevant tensors involved in the execution of a single accelerator: input activation, weight, and output activation tensor. Indeed, we assume that the complete tensors have been fragmented in tiles [45] to exploit data locality in this on-chip global buffer. Moreover, we assume that an on-chip embedded processor invokes each accelerator to process those tiles one at a time.

Table 2 shows the maximum tiles dimensions (Part I) and the maximum tiles sizes (Part II) that would be stored in the global buffer. We determined these dimensions through a statistical analysis of the layer shapes of the most common DNNs for edge devices [10], [11] that are available in public Model Zoos for computer vision applications, such as TensorFlow [46], [47], Intel [48], [49], Xilinx [50], and Nvidia [51], [52]. These networks include the well-known families of ResNet, MobileNet, and EfficientNet. Based on our survey, we select 18 and 22 as the input tile height/width (IH / IW) and output tile height/width (OH / OW) dimensions for 2D- and DW-Conv, respectively, because these values represent a reasonable trade-off between area of the global buffer and number of iterations over the tiles required by the accelerators to complete the DNN layers [10], [11]. Height and width of weight tiles (KH / KW) are instead 7 and 5 for 2D- and DW-Conv, respectively, to ensure that the accelerators support the majority of DNNs (e.g., ResNetV1 uses 7×7 kernels).

Regarding the input tile channels/activations (IC / IA) and output tile channels/activations (OC / OA), we vary their size during the DSE of ST-based accelerators as discussed in Sec. VII-B. The values explored are in the first two rows of Table 3, which also contains HLS directives and implementation constraints that we let vary during the DSE. We call these variables *hardware configuration knobs* because they affect how the RTL is synthesized by the HLS tool. We use 32 as maximum value for IC and OC because we found that the number of input and output channels of activations and weight tensors of common DNNs are often divisible by this value. For the FC accelerator, we select values of IA and OA starting from those used in [6], which were 256 and 8, respectively. Then, we add values in a power-of-two fashion to expand the spectrum of solutions for our design space and to ensure that the area covered by all three accelerators ranges approximately from a minimum to a maximum in the same manner. We will describe the remaining hardware configuration knobs later in this section.

TABLE 2. Description of accelerators' parameters related to the tiles (Part I and Part II) and accelerators' internal buffers sizes (Part III). The values of the parameters explored during the DSE of Sec. VII-B, denoted by the *DSE* entry, are listed in Table 3.

Param. name	Description	2D-Conv	DW-Conv	FC
Part I: Maximum tiles dimensions				
IH / IW	Input tile height / width	18	22	–
KH / KW	Weight tile height / width	7	5	–
OH / OW	Output tile height / width	18	22	–
IC / IA	Input tile channels / activations	DSE	–	DSE
OC / OA	Output tile channels / activations	DSE	DSE	DSE
Part II: Maximum tiles sizes				
IS	Input tile size	$IH \times IW \times IC$	$IH \times IW \times IC$	IA
WS	Weight tile size	$KH \times KW \times IC \times OC$	$KH \times KW \times OC$	$OA \times IA$
OS	Output tile size	$OH \times OW \times OC$	$OH \times OW \times OC$	OA
Part III: Accelerators' internal buffers sizes				
IBS	Input buffer ($IBUF$) size	$KH \times KW \times 4$	$KH \times KW \times 1$	128
WBS	Weight buffer ($WBUF$) size	$KH \times KW \times 4 \times OC$	$KH \times KW \times 1 \times OC$	$OA \times 128$
OBS	Output buffer ($OBUF$) size	$1 \times 1 \times OC$	$1 \times 1 \times OC$	OA

TABLE 3. Hardware configuration knobs explored in the DSE of Sec. VII-B, including maximum tiles size, HLS directives, and implementation constraints.

Hardware configuration knob	HLS directive	2D-Conv	DW-Conv	FC
IC / IA		$\{4, 8, 16, 32\}$	–	$\{256, 512, 1024\}$
OC / OA		$\{4, 8, 16, 32\}$	$\{2, 4, 8, 16, 32\}$	$\{8, 16, 32\}$
PSMAC array parallelism (M) = = Unroll loop *oc* / *oa*	*unroll <unrolling factor (UF)>*	*OC*	*OC*	*OA*
Partition *IBUF*	*interleave <# of interleaved memories>*	no	*OC*	no
Partition *WBUF*	*interleave <# of interleaved memories>*	*OC*	*OC*	*OA*
Partition *OBUF*	*interleave <# of interleaved memories>*	*OC*	*OC*	*OA*
Map *st_multiplier_function* to ST multiplier IP	*map_to_operator <component name (X)>*	yes (*IP mode*, X = { [6], [7], [17], [21], [22], BW-ADD, HLS ST }) / / no (*Inline mode*, i.e., the *st_multiplier_function* is inlined)		
Target clock frequency		[100 ÷ 1000] MHz, 10 steps		

Let us now comment on the pseudo-code at the top of Fig. 4. It is a concise version of the high-level C/C++ description that produces the general architecture of the ST-based accelerators using HLS techniques. We first refer to this simplified code to highlight the commonalities between the high-level descriptions of the various accelerators. Then, we provide specific details on how the key parts of this code translate into the high-level C/C++ pseudo-codes of the three accelerators, reported in Lsts. 2, 3, and 4, for 2D-Conv, DW-Conv, and FC, respectively.

After a series of pipelined outermost loops L1–L3, the accelerator reads activations from the internal input buffer ($IBUF$) and prepares the first operand A for the ST multiplier through the memory addressing and concatenating logic. For 2D-Conv and FC, this operation takes place before the innermost loop L4; however, in the case of DW-Conv, it occurs within L4 because there is no input channels loop in the DW-Conv algorithm. Then, in L4 the accelerator reads weights from the internal weight buffer ($WBUF$) and fills the second operand B. Subsequently, it performs the multiplication/dot-product operation using the ST multiplier configured via *CONFIG* and accumulates the result in the internal output buffer ($OBUF$). The latter keeps stored the result of the previous tile iteration, or is reset in L2 when the accelerator processes the initial input-weight pair of tiles of a layer execution (see the *RESET* signal in Lsts. 2–4).

Since L4 is unrolled, the HLS tool synthesizes it by generating the array of M parallel reconfigurable ST-based PSMAC units shown in Fig. 4. To comply with the working principle presented in Sec. V-A, loop L3 needs to terminate earlier in low-precision configurations: this happens when the index of L3 reaches its maximum number of iterations ($L3_{max}$) divided by N, where $L3_{max}$ corresponds to the number of input channels for 2D-Conv, the product of the kernel dimensions for DW-Conv, or the number of input activations for FC, of the current tile execution. This is implemented by variables *ic_lim*, *k_lim*, and *ia_lim* in Lsts. 2, 3, and 4, respectively. As the number of iterations of loop L3 decreases at reduced precision, the remaining readings from $IBUF$ and $WBUF$ are not performed. Thus, there is no need to fill with zeros the unused parts of these buffers. Finally, only when the accelerator creates the last output tile, $OBUF$ undergoes quantization using the corresponding UIQ formula (i.e, Eq. (14) for 2D-Conv, (16) for DW-Conv, (6) for FC), followed by ReLU (when

LISTING 2. Pseudo-code of our ST-based 2D-Conv accelerator.

```
#include <ac_int.h>

#pragma map_to_operator "X"
int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}

void conv2d(
        int4 IBUF_A[IS], IBUF_B[IS], IBUF_C[IS], IBUF_D[IS],
        int4 WBUF_A[WS], WBUF_B[WS], WBUF_C[WS], WBUF_D[WS],
        ac_int<28, true> OBUF[OS],
        uint3 CONFIG, uint1 RESET) {
 int ic_lim;
 if (CONFIG==(8x8 || 8x4)) { ic_lim = IC/2-1; }
 else if (CONFIG==4x4) { ic_lim = IC/4-1; }
 else { ic_lim = IC-1; }

 #pragma pipeline_init_interval 1
 for (int oh=0; oh<OH; oh++) {
  #pragma pipeline_init_interval 1
  for (int ow=0; ow<OW; ow++) {
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
    if (RESET==1) { OBUF[OC*(OH*oh+ow)+oc] = 0; }
   }
   #pragma pipeline_init_interval 1
   for (int ic=0; ic<IC; ic++) {
    #pragma pipeline_init_interval 1
    for (int kh=0; kh<KH; kh++) {
     #pragma pipeline_init_interval 1
     for (int kw=0; kw<KW; kw++) {
      // Memory addressing and concatenating logics for A
      int A_idx = IC*(IW*(oh+kh)+(ow+kw))+ic;
      int4 A_HH = IBUF_A[A_idx];
      int4 A_HL = IBUF_B[A_idx];
      int4 A_LH = IBUF_C[A_idx];
      int4 A_LL = IBUF_D[A_idx];
      int16 A = ((A_HH<<12)&0xF000) | ((A_HL<<8)&0x0F00) |
                ((A_LH<< 4)&0x00F0) | ( A_LL    &0x000F);
      #pragma unroll OC
      for (int oc=0; oc<OC; oc++) {
       // Memory addressing and concatenating logics for B
       int B_idx = OC*(IC*(KH*kh+kw)+ic)+oc;
       int4 B_HH = WBUF_A[B_idx];
       int4 B_HL = WBUF_B[B_idx];
       int4 B_LH = WBUF_C[B_idx];
       int4 B_LL = WBUF_D[B_idx];
       int16 B = ((B_HH<<12)&0xF000) | ((B_HL<<8)&0x0F00) |
                 ((B_LH<< 4)&0x00F0) | ( B_LL    &0x000F);
       // Reconfigurable ST-based PSMAC array
       int28 P = st_multiplier_function(CONFIG,A,B);
       OBUF[OC*(OH*oh+ow)+oc] += P;
      } //oc
     } //kw
    } //kh
    if (ic==ic_lim) { break; }
   } //ic
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
    // Quantization logic (Eq. 14) and ReLU (when needed)
    ...
   }
  } //ow
 } //oh
}
```

LISTING 3. Pseudo-code of our ST-based DW-Conv accelerator.

```
#include <ac_int.h>

#pragma map_to_operator "X"
int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}

void dwconv(
        int4 IBUF_A[IS], IBUF_B[IS], IBUF_C[IS], IBUF_D[IS],
        int4 WBUF_A[WS], WBUF_B[WS], WBUF_C[WS], WBUF_D[WS],
        ac_int<28, true> OBUF[OS],
        uint3 CONFIG, uint1 RESET) {
 int k_lim,i0,i1,i2,i3,j0,j1,j2,j3;
 if (CONFIG==(8x8 || 8x4)) { k_lim = (KH*KW)/2; }
 else if (CONFIG==4x4) { k_lim = (KH*KW)/4; }
 else { k_lim = (KH*KW)-1; }

 #pragma pipeline_init_interval 1
 for (int oh=0; oh<OH; oh++) {
  #pragma pipeline_init_interval 1
  for (int ow=0; ow<OW; ow++) {
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
    if (RESET==1) { OBUF[OC*(OH*oh+ow)+oc] = 0; }
   }
   #pragma pipeline_init_interval 1
   for (int k=0; k<KH*KW; k++) {
    #pragma unroll OC
    for (int oc=0; oc<OC; oc++) {
     // Memory addressing and concatenating logics for A
     i0,i1,i2,i3 = LUT_i(CONFIG,k);
     j0,j1,j2,j3 = LUT_j(CONFIG,k);
     int4 A_HH = IBUF_A[OC*(IN_W*(oh+i0)+(ow+j0))+oc];
     int4 A_HL = IBUF_B[OC*(IN_W*(oh+i1)+(ow+j1))+oc];
     int4 A_LH = IBUF_C[OC*(IN_W*(oh+i2)+(ow+j2))+oc];
     int4 A_LL = IBUF_D[OC*(IN_W*(oh+i3)+(ow+j3))+oc];
     int16 A = ((A_HH<<12)&0xF000) | ((A_HL<<8)&0x0F00) |
               ((A_LH<< 4)&0x00F0) | ( A_LL    &0x000F);
     // Memory addressing and concatenating logics for B
     int4 B_HH = WBUF_A[OC*(KH*i3+j3)+oc];
     int4 B_HL = WBUF_B[OC*(KH*i2+j2)+oc];
     int4 B_LH = WBUF_C[OC*(KH*i1+j1)+oc];
     int4 B_LL = WBUF_D[OC*(KH*i0+j0)+oc];
     int16 B = ((B_HH<<12)&0xF000) | ((B_HL<<8)&0x0F00) |
               ((B_LH<< 4)&0x00F0) | ( B_LL    &0x000F);
     if ((k==k_lim)&&(CONFIG==(8x8 || 8x4)||CONFIG==4x4)) {
       // Force two or three 4-bit chunks of a and b
       // to zero in place of the missing elements
       ...
     }
     // Reconfigurable ST-based PSMAC array
     int28 P = st_multiplier_function(CONFIG,A,B);
     OBUF[OC*(OH*oh+ow)+oc] += P;
    } //oc
    if (k==k_lim) { break; }
   } //k
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
     // Quantization logic (Eq. 16) and ReLU (when needed)
     ...
   }
  } //ow
 } //oh
}
```

needed), preparing the output for the computation of the next layer. Otherwise, *OBUF* keeps accumulating the partial result/output inside the accelerator to avoid data transfers in the external memory, thus following an *output-stationary* dataflow [42].

Below we delve into the details of each architectural block of the ST-based accelerator illustrated in Fig. 4, highlighting the key differences between the three accelerators.

1) INTERNAL BUFFERS

Part III of Table 2 reports the accelerators' internal buffers sizes. These follow the same ordering of the parameters used by the tile sizes in Part II. For *IBUF* and *WBUF* of 2D- and DW-Conv we choose the minimum sizes that allow to compute $1 \times 1 \times OC$ output elements. In particular, we size *IBUF* of 2D-Conv to store 4 input channels, to allow ST multipliers to operate in all precision configurations. For *WBUF* we choose the kernel dimensions of 7 and 5, following the weight tile dimensions of Part I. For FC we size *IBUF* to store 128 activations and *OBUF* to store *OA* output elements, to have a buffer area comparable with that of the other two accelerators. The internal buffers use double buffering to ensure uninterrupted operations by the accelerators while fetching new data from the global buffer. Thus, from the accelerators' point of view, the whole memory hierarchy composed of global buffer and internal buffers behaves as a unified virtual memory that they can access transparently.

LISTING 4. Pseudo-code of our ST-based FC accelerator (inspired by [6]).

```
#include <ac_int.h>

#pragma map_to_operator "X"
int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}

void fc(int4 IBUF1_A[IS/2], IBUF2_A[IS/2],
        int4 IBUF1_B[IS/2], IBUF2_B[IS/2],
        int4 IBUF1_C[IS/2], IBUF2_C[IS/2],
        int4 IBUF1_D[IS/2], IBUF2_D[IS/2],
        int4 WBUF1_A[WS/2], WBUF2_A[WS/2],
        int4 WBUF1_B[WS/2], WBUF2_B[WS/2],
        int4 WBUF1_C[WS/2], WBUF2_C[WS/2],
        int4 WBUF1_D[WS/2], WBUF2_D[WS/2],
        ac_int<28, true> OBUF[OS],
        uint3 CONFIG, uint1 RESET) {
 int ia_lim;
 if (CONFIG==(8x8 || 8x4)) { ia_lim = IA/4-1; }
 else if (CONFIG==4x4) { ia_lim = IA/8-1; }
 else { ia_lim = IA/2-1; }

 #pragma pipeline_init_interval 1
 for (int oa=0; oa<OA; oa++) {
  if (RESET==1) { OBUF[oa] = 0; }
 }
 #pragma pipeline_init_interval 1
 for (int ia=0; ia<IA/2; ia++) {
  // Memory addressing and concatenating logics for A1, A2
  int a_idx = ia;
  int4 A1_HH = IBUF1_A[a_idx], A2_HH = IBUF2_A[a_idx];
  int4 A1_HL = IBUF1_B[a_idx], A2_HL = IBUF2_B[a_idx];
  int4 A1_LH = IBUF1_C[a_idx], A2_LH = IBUF2_C[a_idx];
  int4 A1_LL = IBUF1_D[a_idx], A2_LL = IBUF2_D[a_idx];
  int16 A1 = ((A1_HH<<12)&0xF000) | ((A1_HL<<8)&0x0F00) |
             ((A1_LH<< 4)&0x00F0) | ( A1_LL    &0x000F);
  int16 A2 = ((A2_HH<<12)&0xF000) | ((A2_HL<<8)&0x0F00) |
             ((A2_LH<< 4)&0x00F0) | ( A2_LL    &0x000F);
  #pragma unroll OA
  for (int oa=0; oa<OA; oa++) {
   // Memory addressing and concatenating logics for B1, B2
   int b_idx = OA*ia+oa;
   int4 B1_HH = WBUF1_A[b_idx], B2_HH = WBUF2_A[b_idx];
   int4 B1_HL = WBUF1_B[b_idx], B2_HL = WBUF2_B[b_idx];
   int4 B1_LH = WBUF1_C[b_idx], B2_LH = WBUF2_C[b_idx];
   int4 B1_LL = WBUF1_D[b_idx], B2_LL = WBUF2_D[b_idx];
   int16 B1 = ((B1_HH<<12)&0xF000) | ((B1_HL<<8)&0x0F00) |
              ((B1_LH<< 4)&0x00F0) | ( B1_LL    &0x000F);
   int16 B2 = ((B2_HH<<12)&0xF000) | ((B2_HL<<8)&0x0F00) |
              ((B2_LH<< 4)&0x00F0) | ( B2_LL    &0x000F);
   // Reconfigurable ST-based PSMAC array
   int28 P1 = st_multiplier_function(CONFIG,A1,B1);
   int28 P2 = st_multiplier_function(CONFIG,A2,B2);
   int28 P1_plus_P2 = P1+P2;
   OBUF[oa] += P1_plus_P2;
  } //oa
  if (ia==ia_lim) { break; }
 } //ia
 #pragma pipeline_init_interval 1
 for (int oa=0; oa<OA; oa++) {
   // Quantization logic (Eq. 6) and ReLU (when needed)
   ...
 }
}
```

The internal input and weight buffers are organized in four 4-bit memory banks, named *IBUF_A/B/C/D* and *WBUF_A/B/C/D*, respectively, to enable reading low-precision data according to the memory access patterns shown in Fig. 3. This is visible from the *int4* datatype in the function signatures of Lsts. 2–4. The output buffer is organized in 28-bit banks to match the bitwidth of the accumulators in the PSMAC array, as we will see in Sec. V-B3.

To guarantee the proper accelerators' execution, the internal buffers are filled by a Direct Memory Access (DMA) engine following the working principle illustrated in Fig. 3. For 2D-Conv, in configurations 16 × 16 and 16 × 8, *one* element of the input and weight tiles, once read from the global buffer, is extended to 16-bit (if needed) and split into four 4-bit chunks. Each input and weight chunk is then stored, from the most to the least significant, into IBUF_A-D and WBUF_A-D, respectively. In configurations 8 × 8 and 8 × 4, *two* input and *two* weight elements from the channels dimension of the corresponding tiles are extended to 8-bit (if needed) and split into 4-bit chunks. The chunks of the first input and the first weight are stored in IBUF_A-B and WBUF_C-D, respectively; the chunks of the second input and the second weight are stored in IBUF_C-D and WBUF_A-B, respectively. The 4-bit chunks are always stored from most to least significant. In the 4 × 4 case, *four* input and *four* weight elements from the channels dimension of the corresponding tiles are all extended to 4-bit (if needed), and then packed in IBUF_A/WBUF_D, IBUF_B/WBUF_C, IBUF_C/WBUF_B, and IBUF_D/WBUF_A, respectively.

For the FC accelerator, the process to fill the memory banks is similar to that of 2D-Conv. However, the number of the 4-bit memory banks is twice that of 2D-Conv (see lines 6–13 in Lst. 4) for a reason clarified in Sec. V-B3. In configurations 16 × 16 and 16 × 8, *two* consecutive activations from the input tile and *two* consecutive weights from the same row of the weight tile are read from the global buffer, extended to 16-bit (if needed) and split into four 4-bit chunks. From the most to the least significant, the four chunks of the two inputs are stored into IBUF1_A-D and IBUF2_A-D, while those of the two weights are stored into WBUF1_A-D and WBUF2_A-D, respectively. In configurations 8 × 8 and 8 × 4, *four* consecutive activations and weights are read along the input array and the same weight matrix row, respectively. Then, they are all extended to 8-bit (if needed) and each is split into two 4-bit chunks. The two chunks of the four inputs are stored in this order: IBUF1_A-B, IBUF1_C-D, IBUF2_A-B, and IBUF2_C-D, whereas those of the four weights are stored in this order: WBUF1_C-D, WBUF1_A-B, WBUF2_C-D, and WBUF2_A-B. 4-bit chunks are always stored from most to least significant. In the 4 × 4 case, *eight* pairs of consecutive inputs and weights are read, extended to 4-bit (if needed), and stored in the internal memory banks as follows: 1st in IBUF1_A/WBUF1_D, 2nd in IBUF1_B/WBUF1_C, 3rd in IBUF1_C/WBUF1_B, 4th in IBUF1_D/WBUF1_A, 5th in IBUF2_A/WBUF2_D, 6th in IBUF2_B/WBUF2_C, 7th in IBUF2_C/WBUF2_B, 8th in IBUF2_D/WBUF2_A.

Filling the memory banks of DW-Conv for configurations 16 × 16 and 16 × 8 follows the same steps of 2D-Conv. However, for low-precision operating modes the filling process is different. For configurations 8 × 8 and 8 × 4, *two* consecutive input elements from the receptive field of the activation tile and *two* consecutive weights from the corresponding kernel of the weight tile are extended to 8-bit (if needed) and split into 4-bit chunks. The chunks of the first and second inputs are stored in IBUF_A-B, whereas those of the first and second weights are stored in WBUF_C-D, leaving IBUF_C-D and WBUF_A-B unused.

For configuration 4×4, *four* consecutive input elements from the receptive field of the activation tile and *four* consecutive weights from the corresponding kernel of the weight tile are extended to 4-bit (if needed) and then stored in IBUF_A and WBUF_D only, leaving the other banks unused.

The data organization discussed above for the three accelerators is important as it enables the partitioning of the internal buffers into smaller memory banks (through the HLS directive *interleave*). This ensures that each bank contains all the data required by a single PSMAC unit to compute its own channel/activation output elements independently. In this way, the PSMAC array can compute M output channels/activations in parallel, as we show in detail in Sec. V-B3. However, to provide the input operands of ST multipliers in the PSMAC array in one clock cycle for all configurations, as it happens for 2D-Conv and FC, the memory organization of DW-Conv requires that IBUF_B and WBUF_C have two reading ports, and IBUF_A and WBUF_D have four reading ports, whereas all the other banks still have one reading port. As implementing four ports in SRAM ASIC technology would be critical, we decide to use latch-based memories for IBUF_A and WBUF_D.

2) MEMORY ADDRESSING AND CONCATENATING LOGIC

These two logic circuits are designed to implement the working principles outlined in Sec. V-A. Depending on the type of accelerator and selected configuration (as depicted in Fig. 3), the first is responsible for preparing the addresses to properly access *IBUF* and *WBUF* and retrieve the four 4-bit data from each memory bank (e.g., lines 31–35 and 41–45 of Lst. 2 for operand A and B, respectively). The second organizes these data into the 16-bit input operands of the ST multipliers through shift-and-mask operations (e.g., lines 36–37 and 46–47 of Lst. 2). For DW-Conv, these logic circuits are a bit more complex. Indeed, a pair of Look-Up Tables (LUTs) is required to retrieve the proper indexes, pre-computed offline, based on the values of *CONFIG* and k, where k is the iteration counter of the loop over the kernel (line 25 of Lst. 3). Moreover, as already discussed in Sec. V-A2, DW-Conv requires that two or three 4-bit chunks of A and B are filled with zeros in place of the missing low-precision operands, in the last kernel iteration for $N = 2$ or $N = 4$, respectively (lines 44–48 of Lst. 3).

3) RECONFIGURABLE ST-BASED PSMAC ARRAY

The PSMAC array of our ST-based accelerators contains M MAC units, as shown in Fig. 4. Each MAC unit works on a distinct output channel/activation, processing a different filter for 2D-Conv, kernel for DW-Conv, or row of the weights matrix for FC.

The *PSMAC array parallelism (M)*, as listed in Table 3, corresponds to the unrolling factor applied to the innermost loops of the accelerators' high-level code through the HLS directive *unroll*. Specifically, M is equal to *OC* for 2D- and DW-Conv, and to *OA* for FC. This causes the HLS tool to fully unroll the innermost loops (line 39 for Lst. 2, 27 for Lst. 3 and 38 for Lst. 4), because the unrolling factor matches their upper bound, thus replicating M times the ST-multiplier and the accumulation adder. As introduced in Sec. V-B1, to fully leverage this parallelism, we partition the internal buffers into M memory banks, enabling the PSMAC units to access their required data concurrently. For this purpose, we use the *interleave* directive with *OC* (for 2D- and DW-Conv) or *OA* (for FC) as argument. Table 3 also shows that the partitioning is not required for *IBUF* of 2D-Conv and FC since operand A is read outside the innermost unrolled loop (lines 31–37 in Lst. 2, lines 28–36 in Lst. 4).

For 2D- and DW-Conv, each MAC unit consists of one 16-bit ST multiplier (see the function call *st_multiplier_function*), one 28-bit adder and one 28-bit accumulation register (P) (lines 49–50 and lines 50–51 in Lsts. 2 and 3). For FC, we got inspired from [6], thus each MAC unit comprises two 16-bit ST multipliers (to process two activation/weight pairs in parallel), two 28-bit adders (to sum the outputs of the two multipliers and accumulate this result, respectively), and one 28-bit accumulation register ($P1\_plus\_P2$) (lines 50–53 in Lst. 4). This is also the reason why we have twice the input and weight buffers at the interface (lines 6–13 in Lst. 4).

The bitwidth of adders and accumulation registers is the result of the ablation study discussed in Sec. VI-B.

4) QUANTIZATION AND RELU BLOCK

This block implements the UIQ formulas (14), (16), and (6) (with $z_W = 0$ and $z_b = 0$ [13], [14]) into 2D-Conv, DW-Conv and FC, respectively. For an efficient hardware implementation, we convert the division by the output scaling factor s_Y into a multiplication by its inverse. Additionally, we minimize the bitwidth of the C/C++ variables of the UIQ formulas through the ablation study described in Sec. VI-B. When the accelerator has processed the last pair of input/weight tiles needed to complete a specific output tile, the accumulated results in the PSMAC array are ready to be quantized using the UIQ formulas. In fact, the accumulated results correspond to term (c) in all the UIQ formulas (14), (16), and (6). The remaining variables of the UIQ formula are passed to the accelerator as inputs because they can be computed offline.

Furthermore, this block implements layer fusion between UIQ formulas and ReLU as described in Sec. III-A1. Thus, when ReLU is needed, the accelerators can be configured to execute it in hardware. The related pseudo-code, omitted for simplicity, would be at lines 58, 57, and 59 of Lst. 2, Lst. 3, and Lst. 4, respectively.

Finally, all accelerators support per-layer quantization for activations, and per-layer or per-channel quantization for weights, as the latter offers superior performance for DNN quantization, as shown in [14] and [27].

VI. ACCELERATORS DESIGN FLOW

To obtain our ST-based hardware accelerators, we use the design flow outlined in Fig. 5. It consists of the following three steps, which are analyzed in detail in Secs. VI-A–VI-C:

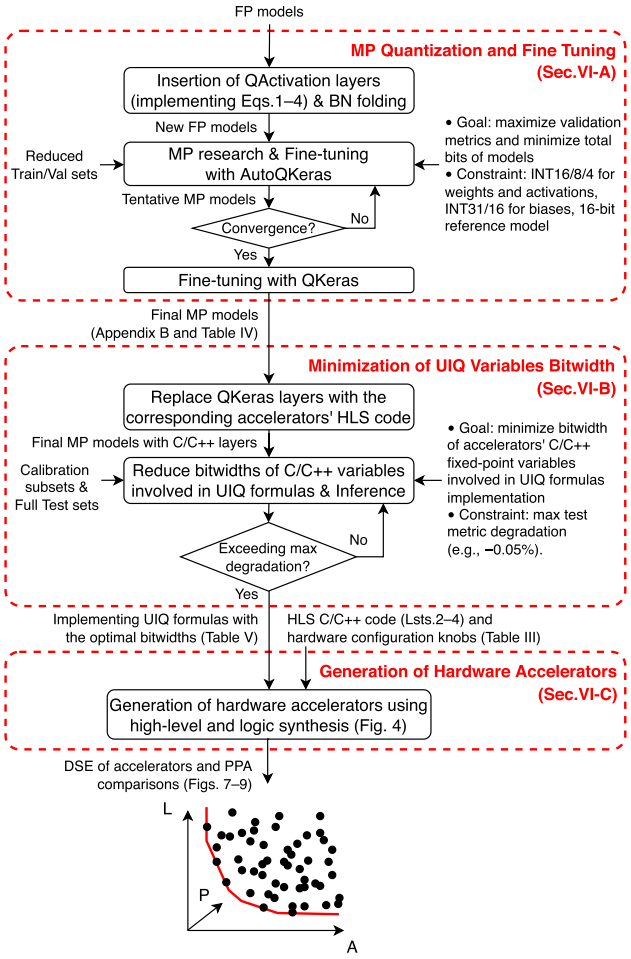

FIGURE 5. Accelerators design flow.

A) **MP Quantization and Fine Tuning**. Quantizing a set of DNN models in MP is the first step of the proposed flow. For this paper, we choose as case study the MLPerf Tiny benchmark [15] because its four networks are well-suited for edge devices, which are the main target for our accelerators. Specifically, we quantize activations and weights of its models on 16-, 8- or 4-bit integers, the same precisions supported by our ST-based accelerators.

B) **Minimization of UIQ Variables Bitwidth**. The second step is an ablation study aimed at optimizing the hardware accelerators using an iterative approach. In this process, we gradually reduce the bitwidth of the C/C++ fixed-point variables of the UIQ formulas, and, for every bitwidth selection, we evaluate the performance of the MP-quantized models, obtained from step A) on their test sets. This process ends by reporting the minimum bitwidths for which the models do not exceed a user-defined degradation threshold.

C) **Generation of Hardware Accelerators**. Using the optimal bitwidth precision determined in step B), we perform a DSE in both latency vs area and latency vs power for each accelerator. In the exploration we vary many hardware configuration knobs, as listed in Table 3, including HLS directives (e.g., pipelining and unrolling) and type of ST multiplier for the PSMAC array.

A. MP QUANTIZATION AND FINE TUNING

We build the first step of our design flow on top of QKeras [16], a Keras extension tailored for quantization tasks. It provides drop-in replacement for some layers to transform a FP Keras model into a quantized one. It supports quantization-aware training by implementing fake-quantized layers and straight through estimator for back propagation. Since QKeras supports affine uniform quantization for weights but not for activations, we create a new activations layer class to implement Eqns. (1)–(4), resulting in a new version of QKeras for integer-arithmetic-only inference. This new version behaves similarly to TFLite [13], but, differently from TFLite, it also supports precisions lower than 8 bits for activations and weights. We release this modified version of QKeras on GitHub as open-source code.[2] As we show in Tables 8–11 in Appendix B for the four MLPerf Tiny models, we insert the new activation layer (called *QActivation*) before and after each *Conv2D*, *DepthwiseConv2D*, and *Dense* layer.

For the bit-width exploration, we use AutoQKeras [16], an extension of QKeras that employs Bayesian Optimization to determine the optimal number of bits for each DNN layer. We constrain weights and activations to INT16, INT8, or INT4 bits, and biases to INT31[3] or INT16, since it is well known that quantizing biases to lower precisions significantly hurts model performance [13], [27]. We configure AutoQKeras to maximize a score function that is the product of the validation metric of the quantized model (bounded between 0 and 1) and the total bit reduction with respect to a 16-bit flat reference model (i.e., a model with all activations and weights quantized to INT16 and biases quantized to INT31). The total number of bits of a model is the sum of the products between the number of activations/weights of each layer and the number of bits used to represent them. In our case study, we use the validation accuracy as validation metric for all MLPerf Tiny models except for FC-AutoEncoder, whose output metric is the mean squared error loss between input and output predictions (MSE_{loss}). To map the +inf–0 range of the MSE_{loss} to the 0–1 range of the other validation metrics (as required by AutoQKeras), we create the following custom validation metric for the AutoEncoder: $1/(1 + MSE_{loss}/10)$.

For each network, we use AutoQKeras to iteratively sample from the search space a different combination of feature map, weight, and bias bitwidths for each layer. Then, we let AutoQKeras fine-tune the resulting MP network for a few epochs starting from pre-trained FP weights, when available,

[2]The modified QKeras supporting affine uniform quantization for activations is available at: https://github.com/LucaUrbinati44/qkeras-mod.git

[3]INT31 (31 bits) is the maximum precision supported by QKeras.

to shorten the bitwidth exploration; otherwise, we let it train the model from scratch. The training is performed using QKeras' quantization-aware training engine. In our case study, since we can rely on the pre-trained weights provided by the MLPerf Tiny repository,[4] we follow the first approach. To further speed up the exploration, we use subsets of the full training and validation sets, together with early stopping. We interrupt AutoQKeras' search when the validation score reaches convergence, i.e., stabilizes around a fixed value. This happens approximately after 200, 400, 100, and 200 search iterations for MobileNetV1Tiny, ResNetV1Tiny, DS-CNN, and FC-AutoEncoder, respectively. Finally, for each model we select the bitwidth combination that gives the best validation score and we conclude by fine-tuning it. In our case study, we use the default settings of the training scripts included in the MLPerf Tiny GitHub repository.

In Sec. V-B, we did not discuss the hardware implementation of BN arithmetic, which we decide not to support in our accelerators to keep lightweight designs. This is not a limitation because BN parameters can be efficiently folded offline with the weights of adjacent convolutional layers using a technique known as BN folding [14]. BN folding is a standard procedure for accelerating DNN inference in embedded devices, as BN parameters remain constant after training. To ensure that applying BN folding to the final MP models obtained from AutoQKeras' exploration would not result in folded weights exceeding the supported bitwidths of our accelerators (16, 8, 4 bits), we proactively provide the FP models to AutoQKeras with pre-folded weights since the beginning of the exploration. Thus, we replace *QConv2D+BatchNormalization* with *QConv2DBatchnorm*, and *QDepthwiseConv2D+BatchNormalization* with *QDepthwiseConv2DBatchnorm*. At the time of our experiments QKeras did not support BN-fused layers for FC layers (i.e., *QDenseBatchnorm* was not yet available). Thus, in our case study we do not apply BN folding to FC-AutoEncoder, as shown in the architecture of the final MP FC-AutoEncoder model (Table 11, Appendix B).

The final MP-quantized MLPerf Tiny models are reported in Appendix B. Their FP and MP performance on the corresponding Perf test sets, using AUC (for FC-AutoEncoder) and accuracy (for the other three models), are provided in columns 4 and 5 of Table 4, respectively. To ensure a solid FP baseline for our comparison with MP models, we re-evaluate the performance of the FP models in our software environment, rather than blindly relying on the values reported in [15] (86, 86.5, 92.2, 88.0, for MobileNetV1Tiny, ResNetV1Tiny, DS-CNN, FC-AutoEncoder, respectively). For this task we use the pre-trained weights and test scripts provided by the MLPerf Tiny repository. In Table 4 we also report the total bits reduction of MP models against their 16-bit flat quantized counterparts (column 7), which are the reference models used by AutoQKeras for guiding the minimization of its objective function, as discussed earlier.

The results show that the MP models exhibit approximately a 1% decrease in accuracy compared to their FP counterparts while still meeting the MLPerf Tiny Quality Targets (column 3), which correspond to the performance that the models should retain after quantization and other optimizations [15]. Moreover, the total bits reduction (column 7) is greater than 50% for all models, confirming the effectiveness of the MP optimization performed by AutoQKeras.

B. MINIMIZATION OF UIQ VARIABLES BITWIDTH

Meeting the hypothetical constraint of zero computational errors in UIQ formulas would require mathematical operators (i.e., multipliers and adders) with excessively large bitwidths, due to the propagation of the bit precision through the involved mathematical operations. This would result in an impractically large accelerator area or could even prevent the HLS tool from generating feasible solutions. Therefore, in this second step of the design flow, we perform an ablation study to optimize the hardware accelerators by reducing the bitwidth of the C/C++ variables used in the UIQ formulas.

Let us consider the UIQ formula (6) of FC, with $z_W = 0$ and $z_b = 0$, as our reference. The same reasoning holds for the UIQ formulas of the other accelerators. The variables that we consider for the ablation study are listed in the column header of Table 5. The first six are the actual variables shown in the UIQ formula, whereas the last three are the intermediate results $v1_{q,k}$, $v2_{q,k}$, $v3_{q,k}$ obtained from the decomposition of (6) in (9)–(12):

$$v1_{q,k} = \left[\sum_{c=1}^{C} X_{q,c} W_{q,c,k} - z_X \sum_{c=1}^{C} W_{q,c,k} \right] \tag{9}$$

$$v2_{q,k} = s_X s_W \cdot v1_{q,k} \tag{10}$$

$$v3_{q,k} = s_b b_{q,k} + v2_{q,k} \tag{11}$$

$$Y_{q,k} = clip(round(z_Y + s_Y^{-1}\, v3_{q,k}), \alpha_q, \beta_q) \tag{12}$$

where $Y_{q,k}$ is the k-th output element, with $k \in [1, K]$, quantized on INTy bits ($y = 16$, 8, or 4) on the integer quantized range $[\alpha_q, \beta_q] = [-2^{b_y-1} + 1, 2^{b_y-1} - 1]$, and all other variables are those introduced alongside (6) in Sec. III-A1. In our accelerators we implement each of these variables as a fixed-point or as an integer number.

Our ablation study aims at optimizing the hardware accelerators using an iterative hardware-software co-design approach. As a preliminary step, we replace the invocations of the low-level TensorFlow routines inside the QKeras *QConv2DBatchnorm*, *QDepthwiseConv2DBatchnorm*, and *QDense* layer classes, with the invocations of the HLS C/C++ code that describes the corresponding accelerator. Then, we start by performing a statistical analysis of the maximum and minimum values taken by each variable. This analysis involves running inference on the MP-quantized models obtained in the previous step of the flow. The inference is performed on small calibration subsets extracted from

[4]GitHub repository of the MLPerf Tiny benchmark: https://github.com/mlcommons/tiny/tree/master/benchmark. Accessed on: Jan 19, 2024.

TABLE 4. Performance of MLPerf Tiny models (column 1) on the corresponding Perf test sets (column 2), using AUC for FC-AutoEncoder and accuracy for the other three models, for their FP (column 4), MP (column 5) and MP with optimal C/C++ bitwidths (column 6) versions.

MLPerf Tiny Model	Perf Test Set (# samples)	MLPerf Tiny Quality Target [15] [Top-1 Acc./AUC]	FP Model [Top-1 Acc./AUC]	MP Model (vs. FP) [Top-1 Acc./AUC]	MP Model with optimal C/C++ bitwidth (vs. FP) [Top-1 Acc./AUC]	Tot. Bits Reduc. (vs. 16-bit flat) [%]
MobileNetV1Tiny	VWW (1000)	80.00	86.00	85.00 (−1.00)	85.00 (−1.00)	−52.25
ResNetV1Tiny	CIFAR-10 (200)	85.00	88.50	87.50 (−1.00)	87.50 (−1.00)	−52.91
DS-CNN	Speech Comm. V2 (1000)	90.00	92.10	90.00 (−2.10)	90.40 (−1.70)	−62.56
FC-AutoEncoder	ToyADMOS (248)	85.00	88.71	87.51 (−1.20)	87.13 (−1.58)	−70.85

TABLE 5. Minimum bitwidths resulting from the ablation study.

C/C++ variable	$\sum_{c=1}^{C} X_{q,c} W_{q,c,k}$	$z_X \sum_{c=1}^{C} W_{q,c,k}$	$s_X s_W$	$s_b b_{q,k}$	$1/s_Y$	z_Y	$v1_{q,k}$	$v2_{q,k}$	$v3_{q,k}$
<integer bits>.<fractional bits>	28.0	28.0	4.24	3.6	18.0	18.0	29.0	33.6	34.6

the corresponding test sets. In this way, we determine the least number of bits of the integer part of each fixed-point variable that retains the maximum MP performance (i.e., AUC for FC-AutoEncoder, or accuracy for the remaining MLPerf Tiny models). Afterwards, with these numbers of integer bits as starting point, we perform a bitwidth exploration of the C/C++ variables of the UIQ formulas (including intermediate variables $v1$, $v2$ and $v3$): we iteratively decrease the number of fractional and/or integer bits, considering one C/C++ variable at a time, and evaluate the effect on the test metric of the considered models by performing inference on their full test sets (the Perf test sets for MLPerf Tiny models). We stop the exploration when it is no longer possible to reduce precision without a reduction greater than a certain threshold in the performance metric of at least one of the analyzed DNNs. In our case study we set a threshold of 0.5% with respect to the MP-quantized test metrics in column 5 of Table 4. In the future, we plan to find these optimal bitwidths automatically through hardware-aware training [53].

The so-obtained optimal bitwidths for the MLPerf Tiny benchmark are in Table 5, whereas the inference results on the MP-quantized MLPerf Tiny models, obtained by invoking the accelerators in software with these bitwidths, are reported in column 6 of Table 4. From these results we observe that: FC-AutoEncoder has an additional penalty of 0.38% against the FP model; MobileNetV1Tiny and ResNetTiny show no further accuracy loss; for DS-CNN, there is even a slight improvement of 0.4%, which is a positive side effect of the quantization process that may occasionally occur [14]. We use these optimal values to synthesize the accelerators in the third step of the accelerators design flow (Sec. VI-C).

C. GENERATION OF HARDWARE ACCELERATORS

In the last step of our design flow, we generate the ST-based accelerators using HLS as shown in the left part of Fig. 4. The procedure consists of two steps. The first performs the actual HLS process by invoking Siemens Catapult (*HLS* block) with the following three inputs:

1) The top C/C++ high-level description of the ST-based accelerator to generate (*C/C++ (top)* block). It reflects the pseudo-codes of Lsts. 2-4;
2) The description of the ST multiplier type to use in the PSMAC array (*RTL/C/C++ (ST)* block): an RTL Intellectual Property (IP) block (*IP mode*) or an inlined C/C++ function (*Inline mode*). The distinction between the two modes will be explained later in this subsection.
3) A set of hardware configuration knobs, sampled from Table 3, and a set of HLS constraints and directives, e.g., clock frequency, unrolling, pipelining, partitioning (*hardware configuration knobs* block).

The second step (*Implem.* block) involves the logic synthesizer, in our case Synopsys Design Compiler (DC), which receives two inputs:

1) The RTL of the accelerator generated by the HLS tool;
2) A set of implementation constraints and usual logic synthesis directives, e.g., clock frequency and clock uncertainty, input/output ports delays, driving/load cells, compilation strategy.

We use the HLS directives to perform several optimizations. As mentioned in Secs. V-B1 and V-B3, we fully unroll the innermost loops in Lsts. 2–4 with the *unroll* directive and partition in banks the accelerator's memories with the *interleave* directive. This combination infers the M parallel MAC units in the PSMAC array and ensures parallel data accesses. For all the other loops we set the *Initiation Interval* to 1 to pipeline their execution and increase the accelerator's throughput. When the HLS tool is not able to find a suitable schedule of the operations that satisfies the timing constraint, we remove pipelining from the outer-most loops (more details in Sec. VII-B). The clock frequency constraint is common to both high-level and logic synthesis. However, in the HLS tool we also set an additional constraint: a clock uncertainty of 50% through the *Clock Overhead* directive, which divides the target clock period in half to take into account the next steps of the flow that might increase the delay, such as routing [54]. This technique helps reduce the critical paths in the generated

RTL by pushing Catapult to insert additional control steps. As a consequence, the logic synthesizer can achieve the desired timing with smaller logic gates.

Concerning the kind of ST multiplier used in the MAC array (*RTL/C/C++ (ST)* block), we have two options. The first is to let the HLS tool map the C/C++ function of the ST multiplier in the high-level description (*st_multiplier_function*) to one of the seven RTL descriptions reported in Table 3. For this we use the directive *map_to_operator* (e.g., line 3 of Lst. 2), followed by the name of the multiplier's RTL top-level entity that we want to use $X = \{$ [6], [7], [17], [21], [22], BW-ADD, HLS ST $\}$. In other words, each ST multiplier type is treated as an IP block called *Catapult C Optimized Reusable Entity* (CCORE) that the tool uses in place of the *st_multiplier_function* function call. In this case, the ST multiplier code is not synthesized along with the accelerator during the HLS process, but is rather instantiated as a component in the generated accelerator's RTL code. We call this first option *IP mode* in Table 3. The second option, not explored in [8], is to let the tool inline the C/C++ function of the ST multiplier in the top high-level description of the accelerator, so that it gets synthesized along with the rest of the accelerator. We call this second option *Inline mode* in Table 3. In this case, we just have to comment out the *map_to_operator* directive from the accelerator's C/C++ top function. Based on Catapult's documentation [54], implementing a function that is called multiple times as a CCORE (in our case the *st_multiplier_function* function subject to the *unroll* directive) is expected to improve design regularity and reduce the shared logic of multiplexers, leading to a better area efficiency. However, we experiment also with function inlining because the advantages of using CCOREs are not always guaranteed and are design-dependent. For example, the operators inside of the CCORE (e.g., multipliers) will not be available for sharing with any other operator of the same type outside the CCORE's boundaries.

VII. EXPERIMENTAL RESULTS

A. PPA COMPARISON OF ST MULTIPLIERS

To compare all the ST multipliers considered in this paper and identify the best in PPA, we follow the same methodology of our previous work [8], which constists of synthesizing their RTL descriptions using DC, on a 28-nm CMOS technology at 0.9 V, after adding I/O registers.

Fig. 6 reports the results of area and power vs clock period obtained by varying the target clock frequency from 0.5 to 1.5 GHz in ten steps. The solutions with the lowest area or power for a given target clock period represent Pareto-optimal points and are connected by a solid black line representing the Pareto front. In both plots, we exclude the right-most outliers to prevent the compression of the left and most significant solutions. Power is calculated using random input bits evenly distributed between zero and one. Although this approach may not faithfully represent realistic ML workloads, it still allows for a valid comparative analysis.

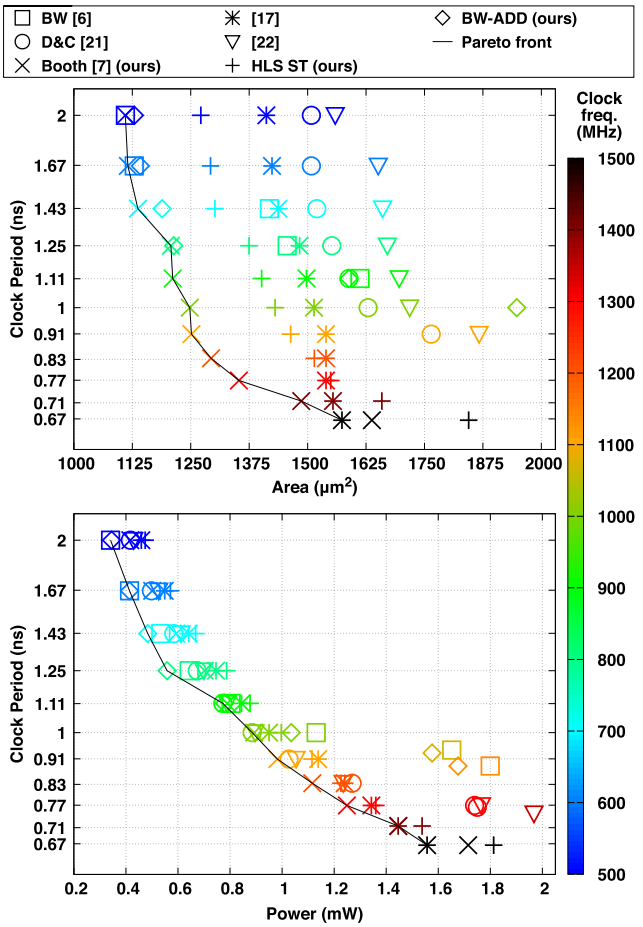

FIGURE 6. DSE of the SoA and newly proposed ST multipliers.

In the area vs clock period graph, the Booth design [7] shares the primacy with [6] at 500 MHz (2 ns), then outperforms the other designs from 600 (1.67 ns) to 1400 MHz (0.71 ns) thanks to its low reconfigurability overhead compared to a standard Booth multiplier, as discussed in [7].

The design of [17] is instead Pareto-optimal in area only at 1500 MHz (0.67 ns). The reason lies in the heuristics of the logic synthesizer. Due to the behavioral description of this ST multiplier, the tool has greater freedom in selecting the best implementation for the internal multipliers and adders in terms of area and timing. As the clock constraint tightens, the tool progressively discovers more area-efficient solutions. Conversely, when the constraint is less stringent, the optimization process halts earlier upon finding solutions that satisfy the desired clock period.

Our new BW-ADD is among the best in area in the low-frequency range, being second best from 700 (1.43 ns) to 800 MHz (1.25 ns), closer to the Pareto front than the original BW [6]. Our results confirm that the BW architecture, although very efficient at low frequencies, is not suitable for higher frequencies [41], even with a faster adder, due to the inherently long critical paths of its BW PPM.

Solutions based on dedicated multipliers for each configuration (like [17], [22], HLS ST) are inefficient in area because of the redundant logic gates not shared among different operating modes. In other words, their internal multipliers operate in a mutually-exclusive manner based on the specific operating mode. Instead, single high-precision multipliers working in a subword-parallel manner (like [6], BW-ADD and [7]) have a higher utilization ratio of their logic gates, which is reflected in a lower area, especially when the timing constraint is not too strict.

The D&C [21] is the second to last in terms of area, which is most likely due to the shift-and-add logic that connects the low-precision multipliers.

In the power vs clock period graph, all solutions are in general very close. The most relevant results are the following: from 400 to 800 MHz, the optimal ST multipliers are those with a BW architecture (e.g., [6] and BW-ADD); from 1000 to 1300 MHz, [7] progressively dominates over [21] and [22]; at high frequencies [17] turns out to be the most power efficient.

To sum up this comparison of ST multipliers, the optimal solutions depend on the PPA constraints: [7] offers the best trade-off in area vs clock period for most of the frequencies, [6] and BW-ADD prove to be Pareto-optimal in power at low frequencies, whereas [7] and [17] are the best in both area and power at high frequencies.

B. DSE OF ST-BASED ACCELERATORS

Similarly to what we did in [8], we perform a DSE in area, power and latency on a 28-nm CMOS technology for the three ST-based accelerators. We use the HLS flow described in Sec. VI-C and vary hardware configuration knobs, implementation constraints, accelerators' internal buffers, and maximum tile sizes, according to the values in Table 3. We also vary the target clock frequency (last row of Table 3) in ten steps from 100 to 1000 MHz, which we verified being the maximum clock frequency reachable by all the accelerators, and the kind of ST multiplier used in the MAC array, for which we have the IP mode or the Inline mode (*HLS ST Inline* in the keys of Figs. 7–8), as explained in Sec. VI-C.

Despite the suboptimal performance of certain ST multipliers, as indicated by our findings in Sec. VII-A, we still incorporate all types of ST multipliers into the DSE of ST-based accelerators to verify whether the ranking observed at the multiplier level remains consistent at the accelerator level.

As introduced in Sec. 4, when the HLS tool fails to meet the target clock frequency, we disable pipelining from some of the outer-most loops of the accelerators. In Table 6 we report the combinations of accelerator type, clock frequency value, OC value, ST multiplier type, and loop name for which we disable pipelining.

Area and power of the accelerators are measured through DC, with the same methodology of Sec. VII-A. The latency of each accelerator point is determined by multiplying the execution time required by the accelerator to process one tile by the total number of tiles into which a reference DNN layer is divided. Such reference layer depends on the accelerator type and is represented by the following (input tensor, weight tensor) pair: (16 × 16×256, 3 × 3×256 × 256) for 2D-Conv; (112 × 112×32, 3 × 3×32) for DW-Conv; (1024, 1000 × 1024) for FC. The first is the most frequent layer among the selected DNNs for edge devices (Sec. V-B); the second is the first depth-wise layer of MobileNetV1; the third is rather arbitrary because FC layers vary significantly from one network to another. In any case, by experimenting with many other tensors sizes, we obtain very similar DSE trends as those reported in Figs. 7–8, which can be therefore extended to any DNN layer. Furthermore, we plot the results normalized so as to make them layer-independent.

TABLE 6. *For* loops of the high-level C/C++ descriptions Lsts. 2–3 for which we disable pipelining in order to allow Catapult HLS to find a schedulable design. We use the loop index as a reference to the loop.

Hw. Acc.	Loop Index	OC	ST mul. type	Clock Freq. [MHz]
2D-Conv	oh, ow	4, 8, 16, 32	all	≥ 200
DW-Conv	oh, ow, k	8	HLS ST Inline	≥ 1000
DW-Conv	oh, ow, k	16	HLS ST Inline	≥ 500
DW-Conv	oh, ow, k	16	not HLS ST Inline	≥ 600
DW-Conv	oh, ow, k	32	all	≥ 500

Figs. 7-8 do not report the results of the entire DSE, but only the Pareto-optimal points. To obtain the two figures, we project these points from the tri-dimensional PPA space to two bi-dimensional spaces of Latency vs Area (LA) and Latency vs Power (LP), respectively. An illustrative example of a complete DSE for the 2D-Conv accelerator is instead reported in Fig. 9, which shows the extensive range of design variations explored. The points in Fig. 7 connected by the solid line (the Pareto front) and labeled in black are LA-optimal (Pareto-optimal in the LA space), whereas those labeled in red are LP-optimal, that is, they belong to the Pareto front in the Latency vs Power plot in Fig. 8. These labels denote the number of input/output channels for 2D-Conv (m, n = IC, OC), output channels for DW-Conv (m = OC), or input/output activations for FC (m, n = IA, OA) used to generate the corresponding accelerator point, according to the notation introduced in Table 2. In a dual manner, Fig. 8 reports the LA and LP projections on the same Latency vs Power graph: this time the black labels identify the LP-optimal points, whereas the red labels mark the LA-optimal ones.

We observe that the majority of optimal points in the LA space are suboptimal in the LP space, and vice versa. Consider an SoC designer aiming to allocate an area of 0.06 mm^2 for a 2D-Conv accelerator. The designer might select solution (B) with (32, 8) input/output channels pair optimized at 400 MHz, achieving a normalized latency of

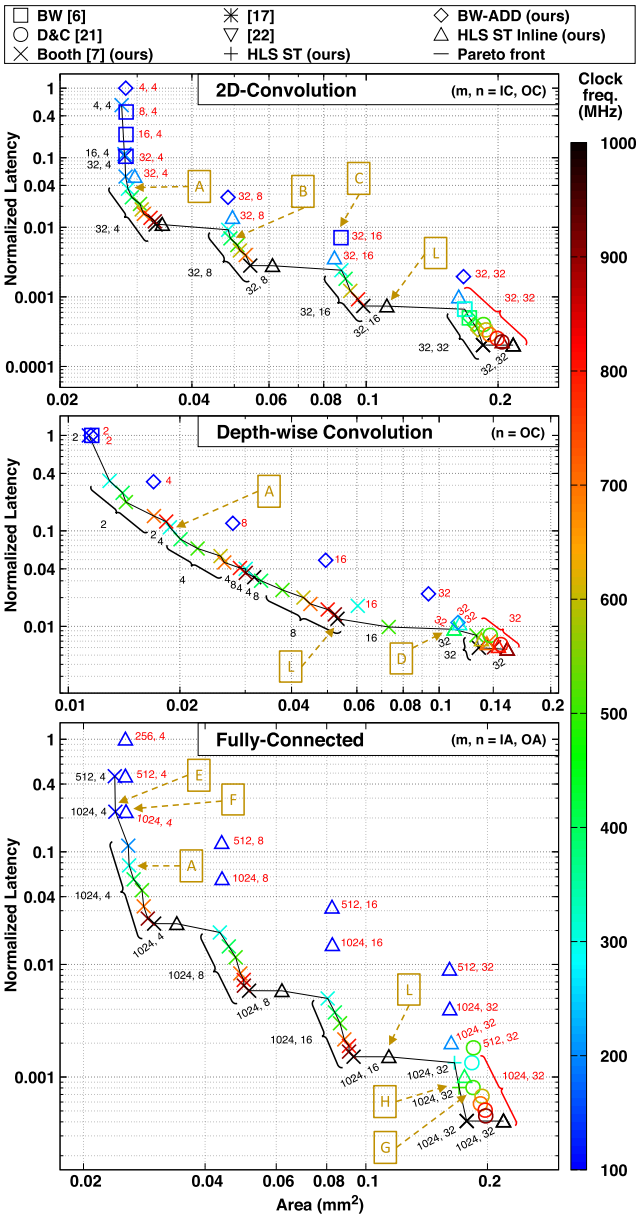

FIGURE 7. Latency vs Area: results of DSE for 2D-Convolution (top), Depth-wise convolution (middle), and Fully-Connected (bottom) accelerators. Points with black and red labels are Pareto points in Latency vs Area and Latency vs Power, respectively.

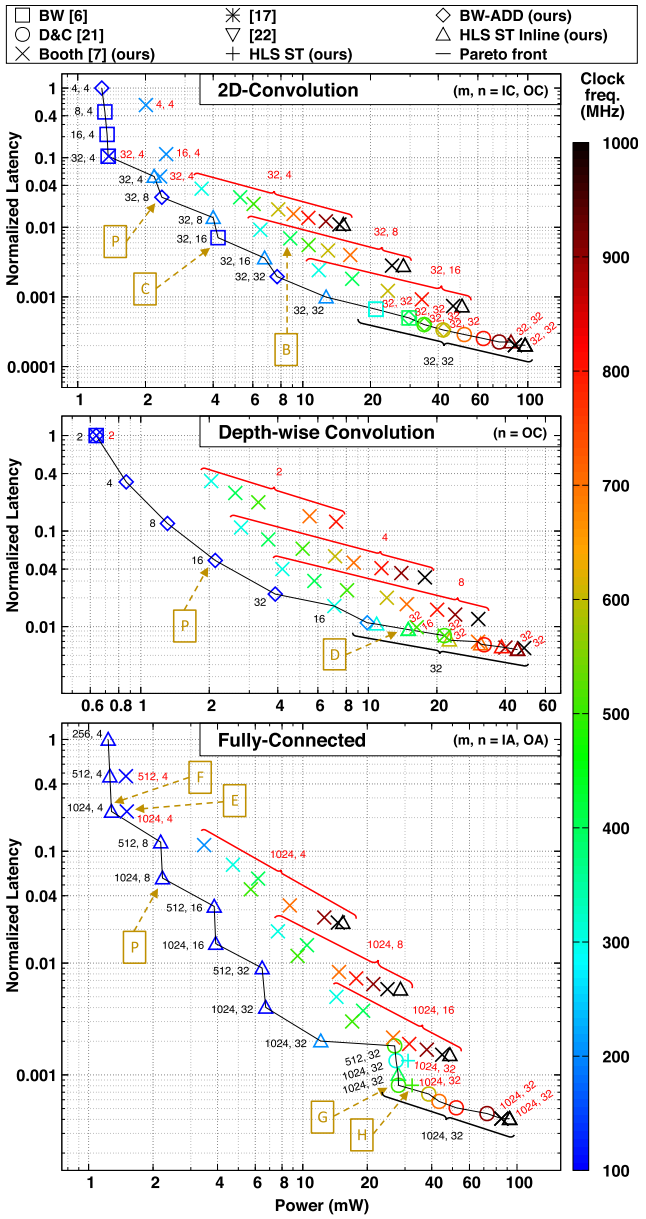

FIGURE 8. Latency vs Power: results of DSE for 2D-Convolution (top), Depth-wise convolution (middle), and Fully-Connected (bottom) accelerators. Points with black and red labels are Pareto points in Latency vs Area and Latency vs Power, respectively.

0.007, and using IP [7] as ST multiplier. However, for the same latency, the power-optimal choice becomes solution (C) with (32, 16) input/output channels pair optimized at 100 MHz, having IP [6]. Note that (C) uses 1.6x more area than (B), whereas (B) consumes around 2.5x more power than (C).

There are also a few points that are optimal in both LA and LP projections: for instance, the DW-Conv accelerator (D), designed for low latency, featuring a 32 channels and operating at 400 MHz with HLS ST Inline.

The designer can also optimize the trade-off in the PPA space by choosing solutions that are LA(LP)-optimal and sit close to the LP(LA) Pareto front. For example, solution (E) with (1024, 4) activations pair, with normalized latency 0.23 and IP [7] at 100 MHz is LA-optimal, but is also very close to the LP Pareto point marked with (F) and using HLS ST Inline, with 18% of power overhead. Conversely, LP-optimal solution (G), with (1024, 32) activations pair and IP [21], is also a valid solution in the LA space with only 8% area overhead with respect to the nearest LA Pareto point (H), which uses HLS ST as IP.

More in general, from Figs. 7–8 we observe that:

- The DSE and the PPA results are especially sensitive to two main variables that control the PSMAC array

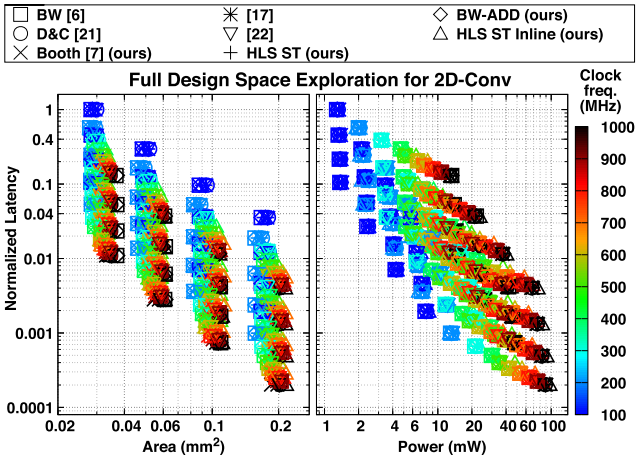

FIGURE 9. Example of a complete DSE for 2D-Conv.

parallelism: OC and OA. As such parallelism increases, the number of MAC units and the size of weight and output memories increase. This leads to an increase of area and power, but more output channels/activations can be simultaneously computed, thanks to the higher number of MAC units, thus reducing latency considerably.

- As expected, very low clock frequencies ($\leq$ 200 MHz) have to be preferred when low area and/or low power are the goals. On the other hand, medium-high clock frequencies are necessary to achieve higher performance.
- The majority of Pareto points for 2D-Conv and FC have always large values of IC and IA, respectively. In fact, increasing these values reduces the overall latency by decreasing the number of tiles N_T and increasing the size of each tile S_T ($=IS$, WS, or OS according to Table 2). This is because, even though the product $N_T \times S_T$ is constant, as N_T decreases the overall latency decreases in the same proportion, since fewer tiles correspond to fewer times that the accelerator is executed; at the same time, each execution has a latency that increases less than proportionally as S_T increases. This is visible in Lst. 2 and Lst. 4: the latency contribution of the loops that do not depend on IC or IA is amortized by the increased latency of the loops that depend on those variables.
- There is a strong correlation between the Pareto-optimal ST multipliers of Fig. 6 and the types of ST multipliers used in the dominating accelerators in Figs. 7–8.
  This is evident especially in the LA space, where a large percentage of the accelerators that sit on the Pareto front (75% for 2D-Conv, 93% for DW-Conv, 80% for FC) have a PSMAC array based on the ST multiplier that we proposed in [7], the dominant point in the Clock Period vs Area subplot of Fig. 6. A few Pareto points, however, are based on BW-ADD and HLS ST, which are indeed sub-optimal in the top graph of Fig. 6 but sit close to the Pareto front. This is because sometimes the optimization heuristics of the logic synthesizer manage to obtain slightly better results with those ST multipliers. Notice that in Fig. 7 no Pareto-optimal accelerators are based on the ST multipliers that are largely sub-optimal in the top graph of Fig. 6 ([17], [21], [22]).
  Similarly, in the LP space, since in the Clock Period vs Power graph at the bottom of Fig. 6 the ST multipliers are all very close to the Pareto front, the dominant accelerators present a more heterogeneous distribution of ST multiplier's types and the choice of the best IP depends on the designer's actual PPA constraints.
- Accelerators with ST multipliers designed manually in RTL are not always the best choice. In fact, there are accelerators with HLS-based ST multipliers or fully-obtained from a C/C++ description (HLS ST Inline) that belong to the Pareto front. In particular, there are some design points using HLS ST in the LA space, and many more using HLS ST Inline in the LP space.

To conclude, the outcomes of the accelerators' DSE do not reveal a single winner, but rather a wide variety of Pareto-optimal solutions, offering SoC designers the flexibility to choose the most suitable implementation aligned with their target, being low area, low power, or high performance. We will see a practical example in the following subsection.

C. INFERENCE ON MP-QUANTIZED MLPERF TINY MODELS

In this subsection, we showcase the benefits in latency and energy achieved by ST-based accelerators when running inference on the four MLPerf Tiny models, quantized in MP as discussed in Sec. VI-A. This is achieved through a comparative analysis against standard accelerators. These accelerators use standard 16-bit multipliers and sign-extend to 16 bits both activations and weights when quantized with a lower precision.

We carry out this comparison in three different constrained PPA scenarios: low-area, low-power, and low-latency, the latter being defined with a significantly larger area constraint than the first.

For each scenario, from the DSE plots of Figs. 7–8 we select a set of ST-based accelerators to be integrated in a hypothetical SoC with the global buffer and an embedded processor. The set comprises one 2D-Conv, one DW-Conv and one FC accelerator, all having the lowest latency while satisfying the given area or power constraint. The processor orchestrates the sequential execution of each layer of the MP-quantized MLPerf Tiny models exploiting tensor tiling and the transparent memory transfers to/from the external memory due to the double buffering mechanism (as explained in Sec. V-B). In particular, for synchronization between embedded processor and accelerators, the double buffering mechanism ensures a smooth and synchronized execution of two subsequent tensor tiles. This method involves utilizing double buffers, enabling the immediate start of the next tile's execution without delay, as the required data for the subsequent tile is already available thanks to the DMA engine. The latter is initialized by the processor at the start

of a layer execution and operates concurrently with the accelerator to fill the double buffers with activation and weight data for the next tile. Upon completion of the last tile of a layer, the processor receives an interrupt from the accelerator and configures the DMA for the acceleration of the next layer. Additionally, at the start of a layer execution, the processor configures the accelerator to the required precision via the *CONFIG* signal (see Lsts. 2–4). We decide to let the processor compute the pooling layers, which, in the case of MLPerf Tiny networks, exclusively consist of average pooling. In general, other pooling methods, such as those developed by [55], could be used and potentially implemented in hardware to further enhance performance. To ensure a fair comparison, an equivalent SoC is created with three standard accelerators. These are synthesized using the same configuration knobs of the three selected ST-based accelerators (refer to Table 3), except that the ST-multipliers are replaced by standard ones, so that the accelerators have the same latency in terms of number of clock cycles of the ST-based accelerators when these are configured at the highest precision (16×16).

The execution latency of an MP-quantized MLPerf Tiny model is calculated as the sum the execution latency of the accelerated layers (2D-Conv, DW-Conv and FC layers), neglecting the execution time of the remaining layers, which are executed in software.

The execution latency of a layer is calculated by multiplying the number of tiles, into which a layer is decomposed, by the execution latency required by the corresponding accelerator to process one tile. Therefore, the *actual latency speedup* is the ratio of the execution latency of an MP-quantized MLPerf Tiny model accelerated using standard accelerators and the execution latency of the same model accelerated using our ST-based accelerators.

We are also interested in the *theoretical latency speedup*, calculated as the ratio of the execution latency of an MP-quantized MLPerf Tiny model accelerated using standard accelerators and the execution latency of the model accelerated using ideal ST-based accelerators, i.e., accelerators whose entire execution latency is accelerated by a factor N. In fact, in our ST-based accelerators only some *for* loops are accelerated by a factor N, as seen in Lsts. 2–4: the loop on the input channels for 2D-Conv, the loop on the kernels for DW-Conv and the loop on the input activations for FC. All the other *for* loops represent an overhead for the actual execution of the accelerator when compared with the ideal accelerator (as we discussed in Sec. VII-B). Thus, the theoretical speedup, compared with the actual one, allows us to seize the impact of this computational overhead. Moreover, the theoretical speedup of a DNN model depends on the model architecture and on how deeply its layers are quantized.

Regarding the energy consumption of an MLPerf Tiny model, we estimate it as the sum of the products between the execution latencies of each accelerated layer and the average power consumption of the corresponding accelerator.

TABLE 7. Latency speedup and energy reduction of the four MP-quantized MLPerf Tiny models executed using accelerators that satisfy different PPA constraints in low-area, low-power, or low-latency. We use the harmonic mean for the mean of the speedups and the arithmetic mean for the mean of the energies.

Scenario, PPA Constraint, Selected Accelerators (Label Figs. 7–8)	MLPerf Tiny Model	Theoretical Latency Speedup	Actual Latency Speedup	Actual Energy Reduction [%]
Low-area, $< 0.03\,mm^2$, *A*	MobileNetV1Tiny	1.51x	1.28x	-15.79
	ResNetV1Tiny	1.58x	1.51x	-25.59
	DS-CNN	2.42x	1.61x	-30.14
	FC-AutoEncoder	1.61x	1.48x	-21.25
	Mean	1.72x	1.46x	-23.19
Low-power, $< 3\,mW$, *P*	MobileNetV1Tiny	1.46x	1.17x	-9.27
	ResNetV1Tiny	1.57x	1.43x	-11.47
	DS-CNN	2.29x	1.34x	-11.35
	FC-AutoEncoder	1.59x	1.43x	-31.38
	Mean	1.68x	1.33x	-15.87
Low-latency, $< 0.12\,mm^2$ *L*	MobileNetV1Tiny	1.47x	1.16x	14.20
	ResNetV1Tiny	1.55x	1.39x	-2.33
	DS-CNN	2.29x	1.30x	7.32
	FC-AutoEncoder	1.55x	1.34x	-6.81
	Mean	1.66x	1.29x	3.10

In Table 7 we report latency speedup (theoretical and actual) and energy reduction for the MP-quantized MLPerf Tiny models executed by accelerators satisfying these PPA constraints: low-area ($< 0.03\,mm^2$), low-power ($< 3\,mW$), and low-latency ($< 0.12\,mm^2$). We mark the selected accelerators with letters *A*, *P*, *L* for the three scenarios, respectively, in both Table 7 and Figs. 7–8. For each scenario, we select three accelerators operating at the same clock frequency (but the frequency can vary across different scenarios).

The results of Table 7 show that our ST-based accelerators speed up inference on the four MLPerf Tiny models in all scenarios, with an actual latency speedup of 1.46x for low-area, 1.33x for low-power, and 1.29x for low-latency, calculated as the harmonic mean of the speedup of the four networks in each scenario.

As for the gap between theoretical and actual speedups, we notice that in every scenario this is more evident for MobileNetV1Tiny and DS-CNN. In fact, these are the only networks using the DW-Conv accelerator, whose speedup improves as the kernel size increases, as seen in Sec. V-A2. Since the kernel in these models is always 3×3, the contribution of the accelerated DW-Conv layers to the speedup is limited.

The average energy reduction across the four models in the low-area and low-power scenarios is -23% and -16%,

respectively. In the low-latency scenario the benefit in energy is less evident and sometimes even unfavourable for ST-based accelerators. This is because the selected ST-based accelerators for this scenario (marked with L in Fig. 7) process many output channels in parallel thanks to the unrolling directive. This implies that part of the reconfiguration logic of ST-based accelerators is replicated, increasing the area and power overhead of ST-based accelerators against standard ones, which do not have the reconfiguration logic. In particular, ST-based DW-Conv accelerator is the one with the most complex reconfiguration logic of the three ST-based accelerators. Not surprisingly, the two models for which the energy of ST-based accelerators actually increases compared to standard ones are MobileNetV1Tiny and DS-CNN.

These results suggest that ST multipliers are well-suited for 2D-Conv and FC, but not for DW-Conv. However, we are already tackling these inefficiencies by developing a new PS DW-Conv accelerator based on another kind of subword-parallel multiplier which works in a Sum-Apart (SA) mode [6]. This has the same configurations of the ST multiplier, but does not return the sum of the low-precision multiplications; instead, it keeps them *separate*, side-by-side, in the multiplier's output. The new working principle of the SA-based DW-Conv accelerator would allow to multiply one high-precision, or two/four low-precision elements from the input and weights channels in parallel, without summing them together, but maintaining the multiplication results separate to adhere to the DW-Conv algorithm.

Finally, we estimate the area overhead of SoCs equipped with ST-based accelerators against SoCs using standard accelerators, for the three scenarios. Other than the three accelerators (internal buffers included), we include a small processor (i.e., Zero-Riscy [56], cache included) and the global SRAM-based buffer. The results show that SoCs with ST-based accelerators exhibit a limited area overhead of 0.9% in the low-area scenario, 2.5% in the low-power one, and 8.0% in the low-latency one, compared to the standard counterparts.

VIII. CONCLUSION

In this paper, we presented our contribution in the area of DNN accelerators using precision-scalable Sum-Together (ST) multipliers. We started by introducing two new ST multipliers (a Baugh-Wooley with modified final adder and one derived from High-Level Synthesis (HLS)), and we made an exhaustive comparison of all the state-of-the-art ST multipliers in terms of power, performance and area (PPA). We then provided detailed insights into the working principles, hardware architectures and design flow of three layer-specific ST-based DNN accelerators for 2D-Convolution, Depth-wise Convolution and Fully-Connected layers, supporting uniform integer quantization. We showcased the Pareto-optimal accelerators resulting from the HLS-driven design space exploration (DSE) in Latency vs Area and Latency vs Power spaces. The results of this DSE allow designers to select the best type of ST multiplier in conjunction with the best configuration of hardware parameters for a given target in the PPA space. Lastly, we demonstrated pros and cons of our ST-based accelerators integrated into a System-on-Chip (SoC) with different design requirements: low-area, low-power, and low-latency. We reported the achieved latency speedup

TABLE 8. MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization).

#	QKeras Layer	Output Shape	Activation / Weight Bits (INT)	Bias Bits (INT)
0	InputLayer	96, 96, 3		
1	QActivation	96, 96, 3	4	
2	QConv2DBatchnorm + ReLU	48, 48, 8	4	31
3	QActivation	48, 48, 8	16	
4	QDepthwiseConv2DBatchnorm + ReLU	48, 48, 8	16	31
5	QActivation	48, 48, 8	16	
6	QConv2DBatchnorm + ReLU	48, 48, 16	16	31
7	QActivation	48, 48, 16	8	
8	QDepthwiseConv2DBatchnorm + ReLU	24, 24, 16	8	31
9	QActivation	24, 24, 16	16	
10	QConv2DBatchnorm + ReLU	24, 24, 32	16	16
11	QActivation	24, 24, 32	16	
12	QDepthwiseConv2DBatchnorm + ReLU	24, 24, 32	8	31
13	QActivation	24, 24, 32	4	
14	QConv2DBatchnorm + ReLU	24, 24, 32	4	31
15	QActivation	24, 24, 32	8	
16	QDepthwiseConv2DBatchnorm + ReLU	12, 12, 32	4	16
17	QActivation	12, 12, 32	8	
18	QConv2DBatchnorm + ReLU	12, 12, 64	8	31
19	QActivation	12, 12, 64	8	
20	QDepthwiseConv2DBatchnorm + ReLU	12, 12, 64	8	16
21	QActivation	12, 12, 64	16	
22	QConv2DBatchnorm + ReLU	12, 12, 64	8	16
23	QActivation	12, 12, 64	8	
24	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 64	8	31
25	QActivation	6, 6, 64	16	
26	QConv2DBatchnorm + ReLU	6, 6, 128	16	31
27	QActivation	6, 6, 128	8	
28	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 128	8	31
29	QActivation	6, 6, 128	4	
30	QConv2DBatchnorm + ReLU	6, 6, 128	4	31
31	QActivation	6, 6, 128	16	
32	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 128	8	31
33	QActivation	6, 6, 128	8	
34	QConv2DBatchnorm + ReLU	6, 6, 128	8	16
35	QActivation	6, 6, 128	16	
36	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 128	16	16
37	QActivation	6, 6, 128	8	
38	QConv2DBatchnorm + ReLU	6, 6, 128	4	16
39	QActivation	6, 6, 128	16	
40	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 128	16	31
41	QActivation	6, 6, 128	16	
42	QConv2DBatchnorm + ReLU	6, 6, 128	8	16
43	QActivation	6, 6, 128	8	
44	QDepthwiseConv2DBatchnorm + ReLU	6, 6, 128	8	16
45	QActivation	6, 6, 128	8	
46	QConv2DBatchnorm + ReLU	6, 6, 128	8	16
47	QActivation	6, 6, 128	8	
48	QDepthwiseConv2DBatchnorm + ReLU	3, 3, 128	8	16
49	QActivation	3, 3, 128	16	
50	QConv2DBatchnorm + ReLU	3, 3, 256	8	31
51	QActivation	3, 3, 256	16	
52	QDepthwiseConv2DBatchnorm + ReLU	3, 3, 256	8	16
53	QActivation	3, 3, 256	8	
54	QConv2DBatchnorm + ReLU	3, 3, 256	4	31
55	QActivation	3, 3, 256	4	
56	AveragePooling2D	1, 1, 256		
57	QActivation	1, 1, 256	4	
58	Flatten	256		
59	QDense	2	4	16
60	QActivation	2	4	
61	Softmax	2		

TABLE 9. MP-quantized model of ResNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization). L marks the left branches, R the right ones.

#	Layer type	Output Shape	Connected to	Activation / Weight Bits (INT)	Bias Bits (INT)
0	InputLayer	32, 32, 3			
1	QActivation	32, 32, 3	0	16	
2	QConv2DBatchnorm + ReLU	32, 32, 16	1	16	31
3	QActivation	32, 32, 16	2	8	
4	QConv2DBatchnorm + ReLU	32, 32, 16	3	8	31
5	QActivation	32, 32, 16	4	16	
6	QConv2DBatchnorm	32, 32, 16	5	16	31
7	QActivation	32, 32, 16	6	8	
8	Add	32, 32, 16	7, 3		
9	ReLU	32, 32, 16	8		
10 L	QActivation	32, 32, 16	9	8	
11 L	QConv2DBatchnorm + ReLU	16, 16, 32	10 L	8	16
12 L	QActivation	16, 16, 32	11 L	8	
13 L	QConv2DBatchnorm	16, 16, 32	12 L	8	16
14 L	QActivation	16, 16, 32	13 L	8	
10 R	QActivation	32, 32, 16	9	8	
11 R	QConv2D	16, 16, 32	10 R	8	16
12 R	QActivation	16, 16, 32	11 R	8	
15	Add	16, 16, 32	14 L, 12 R		
16	ReLU	16, 16, 32	15		
17 L	QActivation	16, 16, 32	16	8	
18 L	QConv2DBatchnorm + ReLU	8, 8, 64	17 L	8	31
19 L	QActivation	8, 8, 64	18 L	8	
20 L	QConv2DBatchnorm	8, 8, 64	19 L	4	31
21 L	QActivation	8, 8, 64	20 L	4	
17 R	QActivation	16, 16, 32	16	8	
18 R	QConv2D	8, 8, 64	17 R	8	16
19 R	QActivation	8, 8, 64	18 R	8	
22	Add	8, 8, 64	21 L, 19 R		
23	ReLU	8, 8, 64	22		
24	AveragePooling2D	1, 1, 64	23		
25	QActivation	1, 1, 64	24	16	
26	Flatten	64	25		
27	QDense	10	26	8	31
28	QActivation	10	27	16	
29	Softmax	10	28		

TABLE 10. MP-quantized model of DS-CNN (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization).

#	Layer type	Output Shape	Activation / Weight Bits (INT)	Bias Bits (INT)
0	InputLayer	49, 10, 1		
1	QActivation	49, 10, 1	16	
2	QConv2DBatchnorm + ReLU	25, 5, 64	16	31
3	Dropout	25, 5, 64		
4	QActivation	25, 5, 64	8	
5	QDepthwiseConv2DBatchnorm + ReLU	25, 5, 64	8	16
6	QActivation	25, 5, 64	8	
7	QConv2DBatchnorm + ReLU	25, 5, 64	4	31
8	QActivation	25, 5, 64	8	
9	QDepthwiseConv2DBatchnorm + ReLU	25, 5, 64	8	16
10	QActivation	25, 5, 64	8	
11	QConv2DBatchnorm + ReLU	25, 5, 64	4	16
12	QActivation	25, 5, 64	8	
13	QDepthwiseConv2DBatchnorm + ReLU	25, 5, 64	4	16
14	QActivation	25, 5, 64	4	
15	QConv2DBatchnorm + ReLU	25, 5, 64	4	16
16	QActivation	25, 5, 64	16	
17	QDepthwiseConv2DBatchnorm + ReLU	25, 5, 64	16	31
18	QActivation	25, 5, 64	4	
19	QConv2DBatchnorm + ReLU	25, 5, 64	4	16
20	QActivation	25, 5, 64	4	
21	Dropout	25, 5, 64		
22	AveragePooling2D	1, 1, 64		
23	QActivation	1, 1, 64	16	
24	Flatten	64		
25	QDense	12	8	31
26	QActivation	12	16	
27	Softmax	12		

TABLE 11. MP-quantized model of FC-AutoEncoder (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization).

#	Layer type	Output Shape	Activation / Weight Bits (INT)	Bias Bits (INT)
0	InputLayer	640		
1	QActivation	640	4	
2	QDense + BatchNormalization + ReLU	128	4	16
3	QActivation	128	16	
4	QDense + BatchNormalization + ReLU	128	8	16
5	QActivation	128	8	
6	QDense + BatchNormalization + ReLU	128	4	16
7	QActivation	128	4	
8	QDense + BatchNormalization + ReLU	128	4	31
9	QActivation	128	4	
10	QDense + BatchNormalization + ReLU	8	4	31
11	QActivation	8	16	
12	QDense + BatchNormalization + ReLU	128	16	16
13	QActivation	128	8	
14	QDense + BatchNormalization + ReLU	128	4	31
15	QActivation	128	8	
16	QDense + BatchNormalization + ReLU	128	8	31
17	QActivation	128	8	
18	QDense + BatchNormalization + ReLU	128	8	16
19	QActivation	128	16	
20	QDense	640	8	16
21	QActivation	640	16	

and energy reduction on the inference of MP-quantized MLPerf Tiny models, as a case study, and also the area overheads of ST-based accelerators, when comparing against SoCs with equivalent accelerators based on non-ST fixed-precision 16-bit multipliers. In the future, we plan to solve the inefficiencies of the ST-based Depth-wise Convolution accelerator discussed in this paper, by implementing a novel accelerator based on Sum-Apart (SA) multipliers.

APPENDIX A
INTEGER-ONLY DNN KERNELS FOR 2D- AND DW-CONV

In this appendix we use the notation of Table 2. As mentioned in Sec. III-A2, the quantized kernels of 2D- and DW-Conv are derived similarly to FC. Let us start from non-quantized 2D-Conv:

$$Y_{oh,\,ow,\,oc} = b_{oc} + \left(\sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{oh+i,\,ow+j,\,ic} \cdot W_{kh,\,kw,\,ic,\,oc} \right) \quad \forall\, oh \in [1, OH], ow \in [1, OW], oc \in [1, OC] \tag{13}$$

$X \in \mathbb{R}^{IH \times IW \times IC}$ is the tensor of input activations, $W \in \mathbb{R}^{KH \times KW \times IC}$ is the weight one, $b \in \mathbb{R}^{OC}$ is the bias array, $Y \in \mathbb{R}^{OH \times OW \times OC}$ is the output tensor; (IH, IW) and (OH, OW) are the dimensions of the input and output tensors, IC

$$
\begin{aligned}
Y_{q,\,oh,\,ow,\,oc} = \; & \underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,\,oc} - z_b)}_{\text{(b)}} \\
& + \frac{s_X s_W}{s_Y}\Bigg[\underbrace{\left(\sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,ic} \cdot W_{q,\,kh,\,kw,\,ic,\,oc} \right)}_{\text{(c)}} \\
& - \underbrace{\left(z_W \sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,ic} \right)}_{\text{(d)}} \\
& - \underbrace{\left(z_X \sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} W_{q,\,kh,\,kw,\,ic,\,oc} \right)}_{\text{(e)}} \\
& + \underbrace{(IC \cdot KH \cdot KW) \cdot z_X z_W}_{\text{(f)}} \Bigg] \\
& \forall\, oh \in [1, OH],\, ow \in [1, OW],\, oc \in [1, OC]
\end{aligned}
\tag{14}
$$

$$
\begin{aligned}
Y_{h,\,w,\,oc} = b_{oc} + \left(\sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{oh+kh,\,ow+kw,\,oc} \cdot W_{kh,\,kw,\,oc} \right) \\
\forall\, oh \in [1, OH],\, ow \in [1, OW],\, oc \in [1, OC]
\end{aligned}
\tag{15}
$$

$$
\begin{aligned}
Y_{q,\,oh,\,ow,\,oc} = \; & \underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,\,oc} - z_b)}_{\text{(b)}} \\
& + \frac{s_X s_W}{s_Y}\Bigg[\underbrace{\left(\sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,oc} \cdot W_{q,\,kh,\,kw,\,oc} \right)}_{\text{(c)}} \\
& - \underbrace{\left(z_W \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,oc} \right)}_{\text{(d)}} \\
& - \underbrace{\left(z_X \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} W_{q,\,kh,\,kw,\,oc} \right)}_{\text{(e)}} \\
& + \underbrace{KH \cdot KW \cdot z_X z_W}_{\text{(f)}} \Bigg] \\
& \forall\, oh \in [1, OH],\, ow \in [1, OW],\, oc \in [1, OC]
\end{aligned}
\tag{16}
$$

and OC the number of input and output channels, and KH and KW the kernel dimensions. To quantize it, we first apply (4) to each real variable in (13). Next, setting their own quantized ranges and moving the quantized output array $Y_{oh,\,ow,\,oc}$ to the left hand side, we obtain the quantized 2D-Conv expression in (14), as shown at the top of the previous page, valid for the $(oh,\ ow,\ oc)$-th output element:

X_q, W_q, b_q, Y_q are the integer values; s_X, s_W, s_b, s_Y are the scaling factors; and z_X, z_W, z_b, z_Y are the zero-points, associated with X, W, b, Y, respectively. The right-hand side of (14) is rounded and clipped to fit the desired output quantized range of Y_q before being assigned to Y_q (not shown in the formula for higher readability). The meaning of terms (a)–(f) is the same as those in the UIQ formula for FC, as discussed in Sec. III-A2. Compared to (6), to compute an output element, the single summation within terms (c), (d) and (e) is replaced with three summations: the indexes of two of them span from 1 to the kernel dimensions KH and KW, whereas the index of the third spans from 1 to IC. Moreover, the constant C in term (f) is replaced with the product of the three upper bounds of the summations.

The expressions of a non-quantized DW-Conv layer and its integer-quantized version are in (15) and (16), as shown at the top of the previous page, respectively. For (15), X, Y, and b are the same tensors of 2D-Conv, except that $IC = OC$; W, instead, is a weight tensor with shape $KH \times KW \times OC$. Eq. (16) is derived from (15) with the same steps used for FC and 2D-Conv. It differs from (14) in the number of summations (two instead of three) and in the absence of the IC constant in term (f).

As discussed in Sec. III-A2 for (6), in this paper we assume that $z_W = 0$ and $z_b = 0$ for (14) and (16) as well.

APPENDIX B
MIXED-PRECISION RESULTS OF MLPERF TINY MODELS

Tables 8–11 report the architecture of the MP-quantized MLPerf Tiny models obtained in the first step of our accelerators design-flow of Sec. VI-A. The last two columns show the number of bits for activation/weight and bias, respectively.

ACKNOWLEDGMENT

Computational resources were provided by HPC@POLITO, a project of Academic Computing within the Department of Control and Computer Engineering, Politecnico di Torino (http://hpc.polito.it). The authors thank their former master's thesis student Marco Alessio Terlizzi for the MP training of MLPerf Tiny models.

REFERENCES

[1] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence*. New York, NY, USA: Chapman & Hall, 2022, ch. 1.2.12, pp. 14–17.

[2] M. Rakka, M. E. Fouda, P. Khargonekar, and F. Kurdahi, "Mixed-precision neural networks: A survey," 2022, *arXiv:2208.06064*.

[3] S. Anwar, K. Hwang, and W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, South Brisbane, QLD, Australia, Apr. 2015, pp. 1131–1135.

[4] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient ConvNets through approximate computing," in *Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV)*, Lake Placid, NY, USA, Mar. 2016, pp. 1–8.

[5] T. Hotfilter, J. Hoefer, P. Merz, F. Kreß, F. Kempf, T. Harbaum, and J. Becker, "Leveraging mixed-precision CNN inference for increased robustness and energy efficiency," in *Proc. IEEE 36th Int. System-on-Chip Conf. (SOCC)*, Santa Clara, CA, USA, Sep. 2023, pp. 1–6.

[6] 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.

[7] L. Urbinati and M. R. Casu, "A reconfigurable multiplier/dot-product unit for precision-scalable deep learning applications," in *Proc. SIE*, Pizzo, Italy, 2022, pp. 9–14.

[8] L. Urbinati and M. R. Casu, "Design-space exploration of mixed-precision DNN accelerators based on sum-together multipliers," in *Proc. 18th Conf. Ph.D Res. Microelectron. Electron. (PRIME)*, Valencia, Spain, Jun. 2023, pp. 377–380.

[9] 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.

[10] L. Urbinati and M. R. Casu, "A reconfigurable depth-wise convolution module for heterogeneously quantized DNNs," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Austin, TX, USA, May 2022, pp. 128–132.

[11] L. Urbinati and M. R. Casu, "A reconfigurable 2D-convolution accelerator for DNNs quantized with mixed-precision," in *Proc. Appl. Electron. Pervading Ind., Environ. Soc. (ApplePies)*, Genoa, Italy, 2023, pp. 210–215.

[12] W. Liu, J. Lin, and Z. Wang, "A precision-scalable energy-efficient convolutional neural network accelerator," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 10, pp. 3484–3497, Oct. 2020.

[13] 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.*, Salt Lake City, UT, USA, Jun. 2018, pp. 2704–2713.

[14] 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*.

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

[16] 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.

[17] 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.

[18] 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.

[19] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, H. Esmaeilzadeh, and J. K. Kim, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Angeles, CA, USA, Jun. 2018, pp. 764–775.

[20] W. Li, A. Hu, G. Wang, N. Xu, and G. He, "Low-complexity precision-scalable multiply-accumulate unit architectures for deep neural network accelerators," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 70, no. 4, pp. 1610–1614, Apr. 2023.

[21] 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.

[22] X. Zhang, Z. Li, and Q. Zheng, "Design of a configurable fixed-point multiplier for digital signal processor," in *Proc. Asia–Pacific Conf. Postgraduate Res. Microelectron. Electron. (PrimeAsia)*, Shanghai, China, Jan. 2009, pp. 217–220.
[23] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, San Francisco, CA, USA, Feb. 2017, pp. 246–247.
[24] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, San Francisco, CA, USA, Feb. 2017, pp. 240–241.
[25] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.
[26] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2018, pp. 1–6.
[27] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*.
[28] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, San Francisco, CA, USA, Feb. 2014, pp. 10–14.
[29] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. Summers, H. Borras, J. Muhizi, M. Trahms, S.-C. Hsu, S. Hauck, and J. Duarte, "QONNX: Representing arbitrary-precision quantized neural networks," 2022, *arXiv:2206.07527*.
[30] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, Apr. 2022.
[31] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual wake words dataset," 2019, *arXiv:1906.05721*.
[32] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Proc. 13th Eur. Conf. Comput. Vis. (ECCV)*, Zurich, Switzerland, 2014, pp. 740–755.
[33] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[34] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Apr. 2009. [Online]. Available: https://www.cs.toronto.edu/
[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, Jun. 2016, pp. 770–778.
[36] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," 2018, *arXiv:1804.03209*.
[37] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*.
[38] Y. Koizumi, Y. Kawaguchi, K. Imoto, T. Nakamura, Y. Nikaido, R. Tanabe, H. Purohit, K. Suefusa, T. Endo, M. Yasuda, and N. Harada, "Description and discussion on DCASE2020 Challenge Task2: Unsupervised anomalous sound detection for machine condition monitoring," 2020, *arXiv:2006.05822*.
[39] Y. Koizumi, S. Saito, H. Uematsu, N. Harada, and K. Imoto, "ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection," in *Proc. IEEE Workshop Appl. Signal Process. Audio Acoust. (WASPAA)*, New Paltz, NY, USA, Oct. 2019, pp. 313–317.
[40] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," in *Advances in Neural Information Processing Systems*, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds. Red Hook, NY, USA: Curran Associates, 2020, pp. 1796–1807.
[41] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design*, 4th ed. Reading, MA, USA: Addison-Wesley, 2011, ch. 11.
[42] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 367–379.
[43] L. Zhang, Q. Lv, D. Gao, X. Zhou, W. Meng, Q. Yang, and C. Zhuo, "A fine-grained mixed precision DNN accelerator using a two-stage big–little core RISC-V MCU," *Integration*, vol. 88, pp. 241–248, Jan. 2023.
[44] 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 2 b-to-32 b fully flexible bit-precision and vector lockstep execution mode," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 6, pp. 2450–2463, Mar. 2023.
[45] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs," *IEEE Trans. Comput.*, vol. 70, no. 8, pp. 1253–1268, Aug. 2021.
[46] TensorFlow. *TensorFlow Hub*. Accessed: Jan. 19, 2024. [Online]. Available: https://tfhub.dev
[47] TensorFlow. *TensorFlow Lite Example Apps*. Accessed: Jan. 19, 2024. [Online]. Available: https://www.tensorflow.org/lite/examples
[48] Intel. *Neural Compute Application Zoo*. Accessed: Jan. 19, 2024. [Online]. Available: https://movidius.github.io/ncappzoo/
[49] Intel. (2023). *OpenVINO Model Zoo*. Accessed: Jan. 19, 2024. [Online]. Available: https://docs.openvino.ai/2023.2/modelzoo.html
[50] AMD Xilinx. *Vitis AI Model Zoo*. Accessed: Jan. 19, 2024. [Online]. Available: https://docs.xilinx.com/r/en-U.S./ug1414-vitis-ai/Vitis-AI-Model-Zoo
[51] NVIDIA. *NVIDIA NGC Catalog*. Accessed: Jan. 19, 2024. [Online]. Available: https://catalog.ngc.nvidia.com/models
[52] NVIDIA. *Jetson Model Zoo*. Accessed: Jan. 19, 2024. [Online]. Available: https://elinux.org/JetsonZoo
[53] M. A. Mansoori and M. R. Casu, "Multi-objective framework for training and hardware co-optimization in FPGAs," in *Proc. Applications Electron. Pervading Ind., Environ. Soc. (ApplePies)*, Genoa, Italy, 2023, pp. 273–278.
[54] *Catapult Synthesis User and Reference Manual*, Mentor Graph. Corp., Wilsonville, OR, USA, 2018.
[55] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi, "Designing novel AAD pooling in hardware for a convolutional neural network accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 3, pp. 303–314, Mar. 2022.
[56] 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.

LUCA URBINATI (Graduate Student Member, IEEE) received the B.Sc. degree in electronics and telecommunications from Università di Bologna, Bologna, Italy, in 2017, and the M.S. degree in electronic engineering from Politecnico di Torino, Italy, in 2019, where he is currently pursuing the Ph.D. degree.

He has also collaborated with researchers in other disciplines, such as food safety, embedded systems, and high-level synthesis. His research interests include integrated architectures for artificial intelligence (AI) and machine learning, focusing particularly on the implementation of AI algorithms in embedded edge devices through a HW–SW co-design approach.

MARIO R. CASU (Senior Member, IEEE) received the Ph.D. degree in electronics and communications engineering from Politecnico di Torino, Turin, Italy, in 2001.

He is currently an Associate Professor with Politecnico di Torino. His research interests include systems-on-chip (SoC) 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 SoC and networks-on-chip. He regularly serves in the Technical Program Committee for international conferences, such as DAC, ICCAD, and DATE.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement