Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(36th cycle)

Accelerating Quantized DNNs with Dedicated Hardware Accelerators and RISC-V Processors Using Precision-Scalable Multipliers

Luca Urbinati

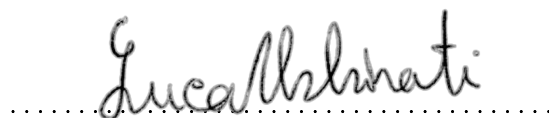
* * * * *

Supervisors
Prof. Casu, Mario R.
Prof. Lavagno, Luciano

Politecnico di Torino
July 1st, 2024

This thesis is licensed under a Creative Commons License, Attribution - Noncommercial-NoDerivative Works 4.0 International: see www.creativecommons.org. The text may be reproduced for non-commercial purposes, provided that credit is given to the original author.

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.



Luca Urbinati
Turin, July 1st, 2024

Summary

Mixed-Precision Quantization (MPQ) and Transprecision Computing (TC) represent two valuable techniques used to optimize Deep Neural Networks (DNNs) inference. They aim at minimizing the number of activation and weight bits for each DNN layer during training, and dynamically adjusting the numerical precision during runtime, respectively. Their goal is to find an optimal balance between accuracy, latency, and energy consumption. Implementing MPQ and TC in practice necessitates the use of Precision-Scalable (PS) and reconfigurable hardware. This aspect constitutes the primary topic of this thesis. Given that Deep Learning (DL) algorithms essentially involve scalar multiplications and dot products for executing convolutions and matrix multiplications, our attention is on PS multipliers. Specifically, we focus on two main categories of PS multiplier architectures, *Sum-Apart (SA)* and *Sum-Together (ST)*, and we integrate them into the Multiply-and-Accumulate (MAC) units of DNN accelerators and low-power extreme-edge RISC-V processors. In these multipliers, N multiplications are computed in parallel in a Single Instruction Multiple Data (SIMD) fashion, with operands on 16/N bits, where N = 1, 2, 4. While SA multipliers keep the results separate from each other, ST multipliers accumulate the results of low-precision multiplication internally, eliminating the need for an external adder. Consequently, they enable support for MPQ and TC and, at the same time, accelerate MAC operations by a factor of up to N compared to conventional full-precision 16-bit multipliers.

Our study provides a comprehensive comparison of the main ST multipliers in the literature. We begin with an overview of State-of-the-Art (SoA) ST multiplier architectures. Next, we introduce three new designs: one optimizing the critical path of a Baugh-Wooley (BW) ST multiplier, another derived from High-Level Synthesis (HLS), and the third based the Booth architecture. We evaluate their performance, power, and area (PPA) characteristics across a wide clock frequency range, after normalizing all the architectures to support 16, 8, and 4 bits of precision. The key finding reveals no single winner that satisfies all PPA scenarios, but rather a set of optimal ST multipliers depending on specific PPA constraints.

Our research also contributes to the advancement of ST-based DNN hardware accelerators by proposing implementations for 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv), and Fully-Connected (FC) layers. These

Application-Specific Integrated Circuit (ASIC) are PS and can be reconfigured at runtime to support operands at 16-, 8-, and 4-bit. We explain their working principles and architectures, and illustrate the design flow. Through extensive HLS-driven design space exploration (DSE), we analyze trade-offs between latency, area, and power, exploring various hardware parameters to identify Pareto-optimal accelerators. Furthermore, we demonstrate the benefits of our ST-based accelerators over those equipped with fixed-precision 16-bit multipliers, i.e., *standard* accelerators. The results of executing the four Machine Learning Performance (MLPerf) Tiny networks quantized in mixed-precision (MP), using SoCs integrating ST-based accelerators tailored to different PPA scenarios (i.e., low-area, low-power, and low-latency), show: an average inference latency speedup, across the four models, of 1.46x, 1.33x, and 1.29x, respectively; a reduced average energy reduction in most of the cases; and a marginal area overhead of 0.9%, 2.5%, and 8.0%, compared to SoCs equipped with standard accelerators. In conclusion, our study offers a complete analysis of ST-based accelerators within the context of SoCs, while highlighting future improvements to address identified inefficiencies.

Considering that SA and ST multipliers have typically been proposed separately in the literature, in this study we introduce a novel class of PS multipliers named *Sum-Together/Apart Reconfigurable (STAR)*, capable of working in both SA and ST modes within a single design. We develop four STAR multiplier architectures, including designs based on established *Divide-and-Conquer (D&C)* and *Sub-word Parallel (SWP)* families, as well as those based on three mutually exclusive datapaths (*3-way*) and separate SA and ST multipliers with multiplexed outputs. Comparative analysis in terms of PPA, conducted in a 28-nm technology, identifies STAR SWP as optimal for low-power and low-area requirements, STAR 3-way as the most suitable for high-performance scenarios, and STAR D&C as competitive for mid-range PPA requirements. These results offer valuable insights for designers aiming to implement efficient multipliers tailored to specific design targets.

Lastly, we integrate a STAR multiplier into the MAC unit of a low-power extreme-edge RISC-V processor for the first time, enabling support for MP-quantized DNNs. Specifically, we replace the default 16-bit multiplier inside the Multiplier/Divider unit of the *Ibex* processor with a 16-bit STAR SWP BW multiplier and introduce new MAC instructions, including standard 32-bit MAC and 16/8/4-bit MAC operations in ST/SA mode. The comparison between our modified Ibex processor and the original one unveils an acceleration up to 5.8× in FC, 3.7× in 2D-Conv, and 2.8× in DW-Conv quantized layers. Additionally, in a 28-nm technology with target clock frequencies of 200 and 600 MHz, the area and power consumption of the proposed solution are 0.015 and 0.017 mm^2, and 1.5 and 4.3 mW, respectively, with a limited overhead within 10% and 3% with respect to the original Ibex. In summary, with notable acceleration gains for typical quantized DNN layers and minimal overhead in terms of area and power consumption, our STAR MAC unit presents a viable option for efficient DL inference in resource-constrained devices.

Acknowledgements

Every doctoral journey is unique and unrepeatable. I am grateful for what I have experienced in my path, despite its ups and downs, as it has allowed me to delve into one of the most important fields of our time: electronics applied to artificial intelligence. In my own small way, I have been able to contribute to the advancement of human knowledge. In this regard, I express my gratitude to my supervisor for guiding my research direction, assisting in planning activities and meeting deadlines, enhancing research outcomes, and traveling to national and international conferences to present our work. Above all, I am thankful for his belief in me when he proposed me to pursue a doctoral degree. I also extend my thanks to all the students I had the pleasure of supervising, for their valuable contributions to my research field through their experimental theses. I especially want to express my gratitude to my friends and colleagues Bernardita Štitić and Edward Manca. Furthermore, I am grateful that this journey has shaped me not only as a researcher but also as a person, allowing me to live some of the most beautiful years of my life with serenity and to seek the path to my success. Now, I am ready to embrace new adventures, both professional and personal, with greater self-awareness and a clearer vision of the person I aspire to become.

To everyone who has been part of my PhD journey

Contents

List of Tables 10

List of Figures 12

1 Introduction 15
1.1 Motivations . . . 15
1.2 Thesis Contributions and Organization . . . 17

2 Related Work 21
2.1 Precision-Scalable Multipliers and MAC Units . . . 21
2.2 Precision-Scalable DNN Hardware Accelerators . . . 23
2.3 RISC-V Processors with Precision-Scalable Hardware Support . . . 25

3 Precision-Scalable Multipliers: Sum-Together (ST) Multipliers 29
3.1 ST Multipliers . . . 29
3.1.1 SoA ST multipliers . . . 31
3.1.2 Booth ST: a radix-4 Booth ST multiplier . . . 35
3.1.3 BW-ADD: a Baugh-Wooley ST multiplier with an improved final adder . . . 38
3.1.4 HLS ST: an ST multiplier derived from HLS . . . 39
3.2 Experimental Results . . . 40
3.2.1 PPA Comparison of ST Multipliers . . . 40

4 High-Level Design of ST-Based DNN Hardware Accelerators 43
4.1 Background . . . 43
4.1.1 Deep Neural Networks' Quantization . . . 43
4.1.2 MLPerf Tiny Benchmark . . . 46
4.2 ST-based Hardware Accelerators . . . 48
4.2.1 Working Principle . . . 48
4.2.2 Accelerators Architecture . . . 50
4.3 Accelerators Design Flow . . . 62
4.3.1 MP Quantization and Fine Tuning . . . 64
4.3.2 Minimization of UIQ Variables Bitwidth . . . 69

4.3.3 Generation of hardware accelerators 71
4.4 Experimental Results 72
4.4.1 DSE of ST-based Accelerators 72
4.4.2 Performance on MP-quantized MLPerf Tiny Models 78

5 Precision-Scalable Multipliers: Sum-Together/Apart Reconfigurable (STAR) Multipliers 83
5.1 STAR Architectures 84
5.2 STAR Sub-word Parallel Baugh-Wooley Design 86
5.3 Experimental Results 91
5.3.1 Power, Performance and Area Comparison of STAR Multipliers 91

6 Accelerating Quantized DNN Layers on RISC-V with a STAR MAC Unit 95
6.1 Ibex: The Baseline RISC-V Processor 95
6.2 The *Fast* MULT/DIV unit of the *small* Ibex 97
6.3 The novel STAR MAC unit integrated in the *small* Ibex 98
6.3.1 STAR BW Multiplier 100
6.3.2 STAR MAC unit 103
6.4 Experimental Results 109
6.4.1 Implementation Results 109
6.4.2 Performance on Quantized DNN layers 109

7 Conclusion and Future Work 113
7.1 Conclusion 113
7.2 Future Work 116

A Integer-only DNN kernels for 2D- and DW-Conv 119

B Mixed-precision results of MLPerf Tiny models 123

C Published Papers and Awards 129

Acronyms 133

Bibliography 135

List of Tables

3.1 Supported precision configurations and operations of the reference ST multiplier of Fig. 3.1. The last three configurations correspond to dot-product operations at low precision. 30
4.1 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. 4.4.1, denoted by the *DSE* entry, are listed in Table 4.2. 52
4.2 Hardware configuration knobs explored in the DSE of Sec. 4.4.1, including maximum tiles size, HLS directives, and implementation constraints. 53
4.3 Performance of MLPerf Tiny models (column 1) on the corresponding Perf test sets (Sec. 4.1.2), using AUC for FC-AutoEncoder and accuracy for the other three models, for their FP (column 3), MP (column 4) and MP with optimal C/C++ bitwidths (column 5) versions. 68
4.4 Minimum bitwidths (row 2) of the C/C++ variables (row 1) resulting from the ablation study. The notation follows the format <integer bits>.<fractional bits>. 69
4.5 *For* loops of the high-level C/C++ descriptions Lsts. 4.1–4.2 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. 73
4.6 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. 80
5.1 Operating modes of STAR. 85
6.1 New MAC instructions and number of required clock cycles. 99
6.2 Logic synthesis results of *Orig.*, *Orig. + MAC* and *STAR-based*. . . 109
6.3 Average speedup (i.e., ratio between clock cycles) of *STAR-based* vs *Orig.* (column 2) and of *STAR-based* vs *Orig. + MAC* (column 3), for three DNN layers for different features and weights bitwidths. . 111

B.1 MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization). 123
B.1 MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization) (continued). 124
B.1 MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization) (continued). 125
B.2 MP-quantized model of FC-AutoEncoder (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization). 125
B.3 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. 126
B.4 MP-quantized model of DS-CNN (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization).127

List of Figures

1.1 Mixed-Precision Quantization (MPQ) (a) and Transprecision Computing (TC) paradigm (b). Images taken from [18] and [10], respectively. . . . 16

1.2 Computing schema of a 4-bit SA multiplier (first row) and a 4-bit ST multiplier (second row). Images derived from [25]. . . . 16

3.1 Reference ST multiplier. 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 3.1. The $X/\cdot$ symbol indicates that the multiplier is capable of multiplication and *dot-product* operations, depending on the configuration. Image taken from [28]. . . . 30

3.2 Our version of BW ST [25]: architecture overview (left), PPM reconfiguration (right). Image taken from [28]. . . . 31

3.3 Our version of ST multiplier [41]: 16-bit high-precision multiplier (left), 8-bit and 4-bit low-precision dot-product units (middle, right). Image taken from [28]. . . . 32

3.4 Our version of D&C ST [49]: four FUs with four 4-bit BitBricks each, interconnected by shift-and-add logics. . . . 33

3.5 (a) 16-bit, (b) 8-bit, and (c) 4-bit operating modes of our version of [49]. Images modified from [33]. . . . 34

3.6 Our version of ST multiplier [48]: four 8-bit Booth multipliers interconnected by muxes ending with an adder tree. Image taken from [28]. . . . 35

3.7 Radix-4 Booth ST (as in [26, 28]): reconfiguration logic (blue), 16-bit Booth multiplier (white and gray). Image taken from [28]. . . . 36

3.8 Alignment of PPi partial products for $CONFIG$ 16×16/16×8 (a), 8×8/8×4 (b) and 4×4 (c). Images taken from [26]. . . . 37

3.9 HLS ST derived from HLS (as in [28]): four multipliers and three adders interconnected by a network of muxes and concatenations. Image taken from [28]. . . . 39

3.10 DSE of the SoA and newly proposed ST multipliers. Image taken from [28]. . . . 41

4.1 Working principle of our ST-based DNN accelerators: 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv) and Fully-Connected (FC). Image modified from [28]. 49
4.2 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. Image taken from [28]. 51
4.3 Memory addressing and concatenating logic acting on the four 4-bit memory banks of the internal input buffer (IBUF_A/B/C/D) of DW-Conv, already filled by an external DMA. Grey banks are unused in the corresponding configuration. Image derived from [40]. 61
4.4 Accelerators design flow. Image taken from [28]. 63
4.5 AutoQKeras search results. Image taken from [23]. 65
4.5 AutoQKeras search results. Image taken from [23]. 66
4.5 AutoQKeras search results. Image taken from [23]. 66
4.5 AutoQKeras search results. Image taken from [23]. 67
4.6 Results of DSE in Latency vs Area (LA). Points with black and red labels are Pareto points in LA and Latency vs Power, respectively. Image taken from [28]. 74
4.7 Results of DSE in Latency vs Power (LP). Points with black and red labels are Pareto points in Latency vs Area and LP, respectively. Image taken from [28]. 75
4.8 Example of a complete DSE for 2D-Conv. Image taken from [28]. . 76
5.1 STAR-enabled reconfigurable MAC for 2D/DW Convolution. Image from [27]. 84
5.2 The proposed STAR (a) Naive and (b) SWP (BW) architectures. Images taken from [27]. 86
5.3 The proposed STAR 3-way architecture. Image taken from [27]. . . 87
5.4 The proposed STAR D&C architecture. Image taken from [27]. . . . 88
5.5 STAR SWP (BW) operating modes. Partial product matrix (top square) and output (bottom rectangle). Right shift for ST modes only (d)-(e). Images taken from [27]. 88

5.6 STAR SWP (BW): PPM with two 8-bit RCAs (top), PPM blocks (bottom), carry propagation “stopper” (bottom right). Each block receives specific binary values (marked with letters from a to i in the block itself) in P and I: a single letter represents logic values for P, while two letters for I and P, respectively (e.g., c/d means $I=$c, $P=$d). White and grey blocks, similar to a standard BW, have an extra AND gate to control the propagation of PPs. Red blocks guarantee the PP inversion and, together with blue and green ones, the generation of logic 1. The “stoppers” (marked with “X”) halt the carry propagation of the low-precision results in SA mode, based on the binary values m and d received in M. All binary values depend on *CONFIG*. Image from [27]. 93
5.7 PPA comparison of STAR architectures. Image taken from [27]. . . 94
6.1 The *small* version of the Ibex core: two-stage pipeline, low-power, RV32IMC RISC-V ISA. Image taken from [54]. 96
6.2 Overview of the original *Fast* MULT/DIV unit of the Ibex core. Image taken from [117]. 97
6.3 PPM of the STAR BW multiplier (a), three versions of PPM blocks (b-d), and carry propagation blocking strategy (e). Images taken from [31]. 101
6.4 The five operating modes of the STAR BW multiplier, where the top square of each configuration is the BW PPM and the bottom rectangle is the multiplier’s output. Images taken from [31]. 102
6.5 The STAR MAC unit implemented in the Ibex core. The *STAR BW Multiplier* is the one reported in Fig. 6.3. Image taken from [31]. . . 103
6.6 Schematics describing: (a) the input multiplexers (of Fig. 6.5) and (b) the *ALU REG UPDATE* block. Images taken from [31]. 104
6.7 The operations performed by STAR MAC at each ICC for (a) MAC, (b) MACH, MACHU, and MACHSU, (c) MACyST, (d) MACySA, and (e) MACySAH instructions ($y \in \{4,8,16\}$). 105
6.7 The operations performed by STAR MAC at each ICC for (a) MAC, (b) MACH, MACHU, and MACHSU, (c) MACyST, (d) MACySA, and (e) MACySAH instructions ($y \in \{4,8,16\}$). 106
6.8 Schematics of: (a) *MAC REG UPDATE* and (b) *C GEN* blocks. Images taken from [31]. 108
6.9 Two examples of STAR MAC used in 2D-Conv and DW-Conv layers: MAC16{ST/SA} in the upper part, MAC8{ST/SA} in the lower part. Image taken from [31]. 110

Chapter 1

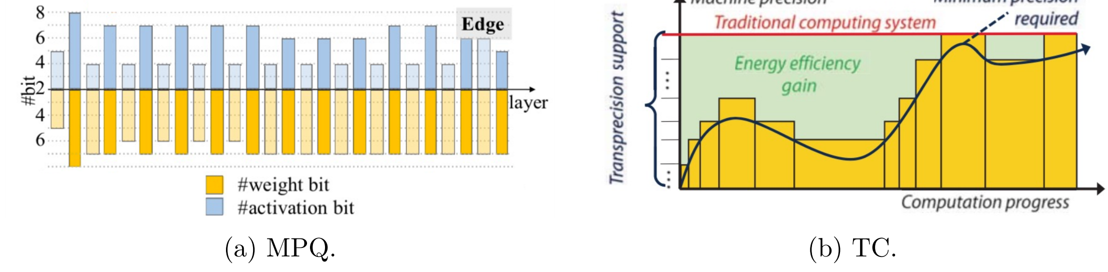
Introduction

1.1 Motivations

In recent years, Deep Learning (DL) at the edge has gained significant momentum due to the promising prospect of deploying intelligent systems on resource-constrained devices [1, 2]. In this context, quantization emerged as a pivotal technique for reducing memory footprint and bandwidth, saving energy, and performing faster Deep Neural Networks (DNNs) inference, by reducing the precision of the numerical representation of weights and feature maps [1, 3].

Recently, there has been a surge of interest in academia and industry towards *Mixed-Precision Quantization (MPQ)* [4]. By leveraging the distinct sensitivities to quantization exhibited by each DNN layer [5, 6], MPQ searches for the minimum number of activation and weight bits for each layer to strike an optimal trade-off between accuracy, latency, and energy consumption [7–9]. Strictly related to MPQ, there is also the so-called *Transprecision Computing (TC)* paradigm, which aims at reconfiguring the numerical format (precision) and the accuracy of computation at run time to boost energy efficiency [10].

The implementation of MPQ and TC requires solving two problems. On one hand, the number of possible activation-weight bitwidth combinations for implementing MPQ grows exponentially in the number of layers and bitwidth precision considered, resulting in a huge search space [11]. Even though there are already several solutions addressing this problem [9, 11–19], and new methodologies that try to solve loss and gradient problems when training networks quantized at low-precision [2, 20–22], this is still an open research problem since the current methods are either slow and computationally demanding [11], or relatively faster but sub-optimal [14]. On the other hand, since the optimal combination of bits for weights and activations may vary across applications and even within the same application across different phases, such as DNNs quantized in mixed-precision (MP) [23], the implementation of MPQ and TC requires *Precision-Scalable (PS)* and reconfigurable hardware. Moreover, since the operations at the core of DL algorithms, such as



(a) MPQ.

(b) TC.

Figure 1.1: Mixed-Precision Quantization (MPQ) (a) and Transprecision Computing (TC) paradigm (b). Images taken from [18] and [10], respectively.

convolutions and matrix multiplications, are essentially scalar multiplications and dot products, recent research efforts have focused on developing PS multipliers [24–28], Precision-Scalable Multiply-and-Accumulate (PSMAC) units [25, 29–31], specialized DNN accelerators [24, 25, 28, 29, 32–40], and microprocessors (MCUs) [31, 41–45], tailored to quantized Machine Learning (ML) workloads.

This second problem is the one addressed in this thesis. In particular, we focus on a) two of the main categories of PS multiplier architectures, *Sum-Apart (SA)* (also known as *Sum-Separate* [25]) and *Sum-Together (ST)*; and b) their integration within the PSMAC units of DNN accelerators and low-power extreme-edge RISC-V processors. Bit-serial architectures, like those proposed in [8, 36, 45], represent another interesting active area of research [29]; however, they will not be discussed.

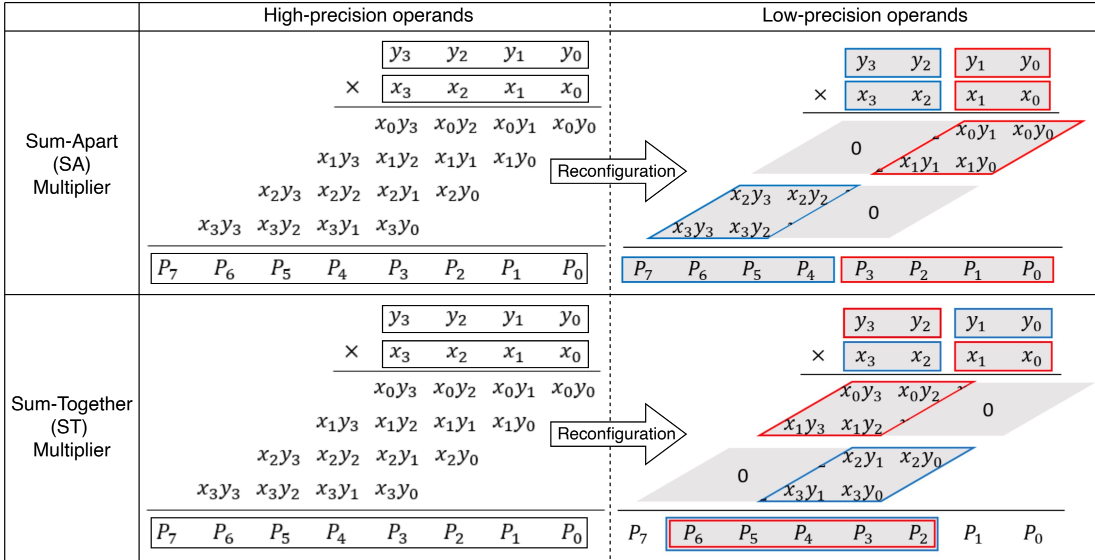


Figure 1.2: Computing schema of a 4-bit SA multiplier (first row) and a 4-bit ST multiplier (second row). Images derived from [25].

Now, we offer a concise overview of the SA and ST multiplier architectures [25]. Fig. 1.2 shows the computing schema of a 4-bit SA multiplier (first row) and a 4-bit ST multiplier (second row). Contrary to a standard multiplier, which merely multiplies the two high-precision operands together, SA and ST multipliers can also be reconfigured to split the two high-precision inputs into sub-words, for computing N low-precision multiplications in parallel between the pairs of sub-words having the same color. In other words, SA and ST multipliers behave like standard multipliers, when processing high-precision operands ("High-precision operands" column in Fig. 1.2), or like sub-word parallel multipliers, when processing low-precision operands ("Low-precision operands" column in Fig. 1.2).ST multipliers perform the product of the two red and two blue sub-words in parallel and then,internally, accumulate the results of these low-precision multiplications within the multiplier itself, thereby eliminating the need for an external adder, but requiring a right shift operation to align the final result to the least-significant bit (LSB) position.Conversely, SA multipliers keep these low-precision multiplication results separate from each other, i.e., *apart*, and may need an external adder in case the specific computation requires the accumulation of these partial results. In general, they perform $N=1$ multiplication when operating at full precision (e.g., on 16 bits); whereas they carry out $N=2$ or 4 parallel dot-products (for ST multipliers) or parallel multiplications (for SA multipliers) using low-precision operands (e.g., on 8 or 4 bits) at reduced precision. In other words, the bitwidth precision of the operands is inversely proportional to N (e.g., $16/N$ bits). Since these multipliers work in a Single Instruction Multiple Data (SIMD) fashion, they exhibit an higher throughput compared to conventional full-precision multipliers, leading to faster DNN inference when used in PSMAC units [29]. Moreover, ST multipliers contribute to further speeding up Multiply-and-Accumulate (MAC) operations thanks to their *sum together* feature. Indeed, a MAC unit that relies on the ST multiplier saves $N-1$ MAC additions when compared to conventional MAC units that rely on conventional non-ST multipliers. As a result, the overall latency is reduced by a factor up to $1/N$.

To sum up, due to their runtime reconfigurability and parallel processing, SA and ST multipliers enable support for MPQ and TC and, at the same time, accelerate MAC operations. Therefore, they have recently found application in MAC units [25, 27, 30, 37, 41, 46], hardware accelerators [24, 25, 28, 33, 35, 37, 39, 40, 46, 47] and microprocessors [31, 41–44].

1.2 Thesis Contributions and Organization

This thesis presents a collection of works in the field of PS multipliers and their implementation within dedicated hardware accelerators and RISC-V processors for quantized DNN acceleration. The contributions of this thesis encompass five topics, each discussed in a dedicated chapter ranging from Chapter 3 to Chapter 6, whereas

Chapter 2 covers the related work and Chapter 7 the conclusion and future work.

- Chapter 3. For the first time, our study undertakes a comprehensive comparison of all the main ST multipliers documented in the literature [25, 41, 48, 49]. Initially, we outline the architectures of the State-of-the-Art (SoA) ST multipliers. Next, we evaluate their performance, power, and area (PPA) characteristics [24, 28]. While our research shares similarities with the most comprehensive work on PSMAC architectures [29], we expand upon that by introducing five additional ST multipliers to the comparison, three of which proposed by us [26, 28]. The first, named *BW-ADD*, is an enhanced version of the Baugh-Wooley (BW) [50] ST multiplier of [25]. It utilizes a modified final adder to shorten the critical path of the original Ripple Carry Adder (RCA). The second design, which we call *HLS ST*, is derived from High-Level Synthesis (HLS). Indeed, one of our objectives is to assess the capability of HLS in generating a competitive ST multiplier in terms of PPA when compared to manually-designed Register-Transfer Level (RTL) implementations. Lastly, we present *Booth ST*, a novel ST multiplier [26] based on the Booth architecture [50], the first of its kind in the SoA, which features a lightweight reconfiguration logic. This comparison provides a broader perspective and valuable insights into the landscape of ST-based architectures.

- Chapter 4. We enrich the collection of accelerators that rely on ST multipliers by proposing three different implementations for the most common DNN layers: 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv), and Fully-Connected (FC). Furthermore, we illustrate their working principle, hardware architecture, and how to obtain them using HLS. Specifically, we present a complete overview of our accelerators design flow, from C/C++ to final hardware implementation. Differently from the DNN accelerators of the SoA, we also incorporate the hardware support for uniform integer quantization (UIQ) [1, 51, 52] to quantize the output activations. Then, we conduct an extensive design space exploration (DSE) of our ST-based accelerators using HLS, examining the Latency vs Area and Power vs Area trade-offs. We explore various hardware parameters, including MAC parallelism, clock frequency, and the specific implementation type of ST multiplier to be employed in the accelerators' MAC units. The results of this DSE serve System-on-Chip (SoC) designers to identify the best combination of hardware configuration knobs for a given PPA target. Lastly, we showcase the benefits achieved by our ST-based accelerators in terms of reduced latency and energy consumption, by comparing them to accelerators equipped with non-ST fixed-precision 16-bit multipliers, which we refer to as *standard* accelerators and *standard* multipliers, respectively. To conduct this assessment, we integrate the accelerators into SoCs under three distinct scenarios (low-area, low-power, and low-latency). As a case study, we execute the models

of the MLPerf Tiny benchmark [53], pre-emptively quantized in MP using a customized version of QKeras [14] for which we open-source the code [28].

- Chapter 5. We propose a new class of multipliers, named *Sum-Together/Apart Reconfigurable (STAR)*, which can be reconfigured to operate either in SA mode or ST mode [27, 31]. This represents a novelty in the literature, as SA and ST multipliers have traditionally been proposed as alternative implementations [25]. We believe that having PS multipliers supporting both SA and ST modes in a single design could be advantageous in various scenarios, such as within the MAC units of RISC-V cores [31] or PS hardware accelerators [27]. In fact, they enable a more efficient utilization of the hardware resources, which can be dynamically shared to perform different tasks. For instance, we show how a STAR-based MAC unit could be used in a single hardware accelerator for executing either 2D or Depth-wise (DW) Convolutions depending on the operating mode of STAR. Moreover, we develop four STAR multiplier architectures, each accommodating $N = 1$ full-precision multiplication, or $N = 2, 4$ parallel multiplications in SA or ST mode, with operands at $16/N$ bits [27]. Among these architectures, two are based on the established *Divide-and-Conquer (D&C)* and *Sub-word Parallel (SWP)* families [29]. In particular, for the SWP approach we consider a BW architecture for which we provide a detailed explanation. The third solution, inspired from [41], adopts a 3-way approach: it comprises three mutually exclusive datapaths, each featuring one 16-bit, two 8-bit, and four 4-bit multipliers, respectively. The fourth architecture consists of two separate multipliers, one SA and one ST, with multiplexed outputs. Finally, we compare them in terms of PPA by varying the clock frequency target over a wide range to identify the best solutions for low-power and low-area, high-performance, and mid-range PPA requirements.

- Chapter 6. We are the first to integrate STAR in the MAC unit of a low-power extreme-edge RISC-V processor to support quantized neural networks with variable reduced precision [31]. Specifically, we replace the default 16-bit multiplier inside the MULT/DIV unit of the Ibex processor [54] with a 16-bit STAR BW multiplier and add new MAC instructions which are not available in the original *MULT/DIV* unit: standard 32-bit MAC and 16/8/4-bit MAC operations in ST/SA mode. Next, we compare our new Ibex processor with the original one in terms of area and estimated power. We also evaluate the achieved performance speedup when running a set of quantized 2D-Conv, DW-Conv and FC layers.

Chapter 2

Related Work

Some of the work described in this chapter was also previously published in [24, 27, 28, 31, 40].

2.1 Precision-Scalable Multipliers and MAC Units

In the literature, various proposals for the implementation of **ST multipliers** can be found. Although the definition of ST mode was initially introduced with the SWP BW ST multiplier by [25], earlier works had already proposed reconfigurable multipliers capable of supporting both single high-precision multiplications and parallel low-precision dot-products [37, 41, 42, 48, 49].

In their work [41, 42], the authors introduced SIMD extensions to the Instruction Set Architecture (ISA) of a RISC-V processor, incorporating sub-word parallel multipliers and low-precision dot-product units to accelerate quantized DNNs.

In [37], the authors proposed a general-purpose systolic array for DL, consisting of reconfigurable Fusion Units (FUs) that utilize low-precision multipliers, called BitBricks, by dynamically combining their outputs or maintaining them separate through a network of combinational logic. A similar approach is used by [30, 32, 34], as well as their precursor [49]. On the other hand, [33] introduced a novel MAC unit termed bit-split-and-combination (BSC) that addresses the issue of memory bandwidth explosion [29] of the D&C MAC units of [37] when operating at the lowest supported precision. This is achieved by gating some of the BitBricks as in the SWP approach of [25].

In [48], the authors introduced a complex reconfigurable fixed-point multiplier, originally intended for digital signal processing (DSP) applications, which has many functionalities including ST multiplications.

The most comprehensive study on PSMACs is documented in [29]. It offers a thorough review and benchmarking of the SoA PSMAC architectures and categorizes them into SWP [25, 35], D&C [37, 46], and bit-serial [36, 55, 56]. The

first category includes architectures capable of operating in either full-precision or reduced-precision mode by selectively gating their arithmetic logic cells. The second consists of architectures composed of many low-precision multipliers (e.g., 4-bit) that, when properly combined using shift-add logic, function as higher precision multipliers (e.g., 16-bit). The architectures of the third type process input operands serially and perform add-shift operations repeatedly at every clock cycle, resulting in a number of cycles for a complete multiplication operation that depends on the bit-width of the operands [36, 56].

Moreover, in [55] the authors extended the concept of bit-serial to multi-bit serial, meaning that the inputs of the PSMAC are fed to the multiplier in multi-bit chunks rather than one-by-one. For example, for a 4-bit serial MAC the weights are fed 4 bits at a time, requiring two clock cycles forn an 8-bit computation. As per the taxonomy outlined in [29], it is important to note that ST architectures can be either SWP or D&C.

In contrast, our study not only evaluates the ST multipliers analyzed in [29], namely the 16-bit Baugh-Wooley ST multiplier from [25] and the FU from [37], but also includes the SWP ST multipliers from [48] and the D&C ST multiplier from [49]. Additionally, we examine the SWP dot-product unit integrated into the RISC-V core described in [41], which functions as a full-fledged ST multiplier. Moreover, we add to the comparison our three newly proposed ST multipliers (BW-ADD [28], HLS ST [28], Booth ST [26]). Another distinction from [29] is our methodology of evaluation: we compare ST multipliers as independent components rather than as PSMAC unit subcomponents. Detailed analyses of the SoA ST multiplier architectures and their PPA comparison are presented in Sec. 3.1 and Sec. 3.2, respectively.

Concerning **STAR multipliers**, we are pioneers in introducing this concept, with only a few related works available.

In their study presented in [25], the authors examined the individual advantages of SA and ST approaches. They conducted a comparison between two FC accelerators, one utilizing SA multipliers and the other employing ST multipliers, both implemented with a BW architecture. The evaluation encompassed factors such as energy consumption, speed, and area utilization. The results highlighted the pros and cons of both approaches without determining a clear winner for every objective.

A non-recent paper [48] introduced a fixed-point SWP DSP unit capable of functioning in both SA and ST modes, although these terms did not exist yet at the time. However, this unit is considerably more complex than our design, as it comprises not only a single SWP and reconfigurable multiplier, but also incorporates four standard 16×16 Booth multipliers, three configurable 33-bit adders, and two saturation logics. Moreover, it supports a range of precisions from 32 down to 8 bits, indicating its broader applicability to regular DSP tasks rather than being

specifically tailored for ML workloads which usually require 8 or less bits. Consequently, we opt to exclude it from our analysis of STAR architectures (Chapter 5).

Building on the proven effectiveness and benefits of SA and ST multipliers demonstrated in previous research [25], we describe the different types of STAR architectures (Secs. 5.1–5.2) and compare them in terms of PPA (Sec. 5.3) [27]. In comparison to [25], we also use a wider clock frequency range from 0.4 to 2.0 GHz.

2.2 Precision-Scalable DNN Hardware Accelerators

A decade ago, during the initial era of DNN accelerators [57], inference predominantly relied on MAC units equipped with fixed-precision multipliers. For example, renowned accelerators like DaDianNao [58], EIE [59], and Eyeriss [60] employed 16-bit multipliers in their MAC units.

Nowadays, propelled by advanced training strategies [20], a new generation of DNN accelerators has emerged, showcasing PS datapaths to support feature maps and weights at reduced precision.

Here, we provide a concise overview of PS accelerators based **exclusively on SA or ST multipliers** [25, 33, 35, 37, 46, 47].

The Envision accelerator, as described in [35], is a convolutional neural network processor equipped with a MAC unit based on a SWP SA BW multiplier which supports one 16-bit, two 8-bit, or four 4-bit multiplications.

In [25], the authors presented two engines for accelerating FC layers. The first implementation employs SA MAC units, while the second one integrates ST MAC units; both of them leverage SWP BW multipliers for their operations. Compared to [35], these engines support also eight 2-bit multiplications.

In their paper [37], the authors introduced a systolic array D&C ST architecture for, known as Bit Fusion, designed for general-purpose DL applications. This architecture is composed of FUs, each of which dynamically assembles and disassembles 2-bit BitBricks to accommodate various input/weight pair bit precisions, including 8/2, 4/4, 2/8, and 8/8 configurations.

DNPU, introduced by [46], is a RISC-based SoC with accelerators for convolution, dense and recurrent layers. The paper also proposed a D&C SA reconfigurable multiplier based on look-up tables (LUTs), which is capable of supporting 16, 8, and 4 bits.

[33, 47] introduced PS systolic accelerators inspired from [37] which are composed of BSC MAC units (Sec. 2.1) supporting operands on 8, 4, and 2 bits.

Other PS accelerators are documented in the literature. For instance, [61] proposed a pipelined PS SIMD MAC unit to accelerate mixed-precision General Matrix Multiply (GEMM) operations using binary segmentation [62] applied to a 16-bit

fixed-precision multiplier; the accelerator proposed by [38] targets primarily training and supports Floating-Point (FP) at 16- or 8-bit precision. However, they fall beyond the scope of this thesis since we deal with combinational PS multipliers, and inference accelerators supporting operands with fixed-point datatype representation, respectively. Interested readers about DNN hardware accelerators are encouraged to explore the following surveys and papers: [29] extensively compares PSMAC units within PS accelerators, including bit-serial accelerators like [36, 56, 63]; the authors of [64, 65] designed serial and partially serial accelerators for RISC-V based SoCs; [66, 67] offer insights into both academic and commercial accelerators, whereas [68, 69] provide an annual overview of commercial accelerators only. Furthermore, PS accelerators tailored for Field Programmable Gate Array (FPGA) applications, like the one discussed in [70], are beyond the scope of this thesis, as our focus is exclusively on Application-Specific Integrated Circuit (ASIC) solutions.

Our research on PS DNN accelerators stands out from the SoA in several key aspects:

- Firstly, to the best of our knowledge, we are pioneers in the design of 2D-Conv and DW-Conv accelerators utilizing ST multipliers [39, 40].
- Additionally, inspired from previous research [25], we revisit the ST-based FC accelerator, to offer a comprehensive suite of ST-based accelerators catering to diverse DNN layers [24, 28].
- We provide detailed insights into the working principles (Sec. 4.2.1) and hardware architecture (Sec. 4.2.2) of our ST-based accelerators. We also comment on the design methodology employed in their realization (Sec. 4.3) [24, 28]. By elucidating these aspects, we aim to contribute to the broader understanding of ST accelerator architectures, offering implementation aspects, and analyzing advantages and disadvantages of using them to accelerate quantized DNNs.
- An additional distinguishing feature of our work is the integration of support for UIQ [51, 52] within our accelerators, a novel contribution not addressed in any of the previously cited accelerators. In this regard, we propose an accelerator design flow that includes the minimization of fixed-point variable bitwidths required by UIQ formulas (Sec. 4.3.2).
- We leverage HLS techniques to derive our ST-based accelerators (Sec. 4.3.3), a method not used by any of the previously cited ASIC accelerators that support precision scalability. This strategic choice allows us to conduct rapid DSEs, identifying optimal solutions in terms of latency, area, and power consumption (Sec. 4.4.1).

- Furthermore, we compare latency and energy consumption of ST-based accelerators when running MP-quantized DNNs [28] against standard accelerators (Sec. 4.4.2). This comparative study represents a novel exploration within the existing literature, shedding light on the advantages offered by ST-based accelerators in real-world applications.

2.3 RISC-V Processors with Precision-Scalable Hardware Support

General-purpose processors offer unmatched flexibility compared to ASIC or FPGA-based accelerators for implementing DL algorithms. They are capable of accommodating a wide range of DNNs, including those yet to emerge in the literature [71]. However, their versatility comes with limitations, e.g., higher inference time, due to their general-purpose ISA, limited computational and memory resources, and constraint energy budget [72]. Additionally, many existing embedded MCUs do not support sub-byte computation, i.e., SIMD instructions, which prevent them to fully exploit the MPQ and TC paradigms [73, 74]. For example, to run a DNN model quantized with 4-bit integers (INT4) on a standard MCU with commodity hardware, such model is typically mapped to the nearest supported precision, such as 8-bit integers (INT8). This means extending the quantized data to match the supported data type and compute one data at a time, resulting in wasted time and suboptimal computational efficiency. Even though packing multiple low-precision operands in the same memory word may save some memory space, the intended benefit in terms of computation time is not achieved [45]. Therefore, new processor architectures supporting quantized ML workloads and TC on Internet-of-Things (IoT) and sensor-node devices are needed.

The RISC-V is an open-source and royalty-free ISA that emerged as a robust alternative to proprietary ISAs. Due to its modular and extensible design, and the explicit support of domain-specific custom extensions, it became soon a reference for both industrial and academic processors over the last decade [75]. RISC-V implementations range from simple and energy-efficient cores, e.g., [41], to those oriented to High-Performance Computing (HPC), e.g., [76–78]. This thesis specifically focuses on low-power extreme-edge RISC-V processors, with a particular emphasis on the Ibex core [54], for which we modify the internal multiplication unit to enable PS support. A comprehensive explanation of our work, along with a summary of the Ibex core, will be provided in Chapter 6.

Now, let us briefly summarize the main RISC-V processors that integrate novel PS hardware resources along with dedicated low-precision arithmetic instructions.

Ri5cy [41] served as the initial inspiration for our research endeavors. It is an

open-source 32-bit RISC-V core that implements the full RV32IMC ISA[1]. The core features a multiplication unit comprising various components, including a standard 32-bit integer multiplier, a 32-bit fixed-point multiplier, and two SWP dot-product units. These dot-product units are implemented with two 17-bit multipliers or four 9-bit multipliers, respectively, followed by a 32-bit adder that sums the intermediate products through a compression tree. They accept two 16-bit or four 8-bit operands, respectively, packed into one 32-bit register, and an optional third input register used for the accumulation. The dot-product units perform up to four multiplications with accumulations simultaneously in a single clock of latency. Unlike our approach, Ri5cy is designed and optimized to operate in a multi-core cluster environment, targeting high-performance DSP-oriented applications [79]. This is evident from the numerous parallel low-precision multipliers within its multiplication unit. In contrast, our approach utilizes a low-power extreme-edge core [54] (Secs. 6.1–6.2) modified to integrate a single PS STAR multiplier (Sec. 6.3) [31]. This multiplier can be reconfigured to perform multiple low-precision SWP multiplications, with or without accumulation. By leveraging the concept of hardware re-utilization, our approach is tailored for small embedded devices.

In [42], the same authors of [41] improved the multiplication unit of Ri5cy by adding the support for low-bitwidth SIMD arithmetic instructions at 4- and 2-bit precision. They followed the same principle to their previous work: integrating an additional set of multipliers and an adder tree for each new supported format. The resulting core can execute 8 or 16 operations per cycle at 4- and 2-bit precision, respectively. Although this enhancement incurs additional area overhead, it does not affect the critical path of the design. Furthermore, they introduce all possible permutations of asymmetric MP opeerations (e.g., 16×8, 16×4, 16×2, 8×16, etc.), along with a custom instruction that simultaneously executes the dot-product while loading an operand for the next operation.

Alternative approaches to SWP for low-precision MAC units in RISC-V cores include D&C [43, 44] and bit-serial [45].

In [43] the authors proposed a D&C MAC unit that works in SA mode and aims to be integreated into a 32-bit microprocessor such as [41]. The unit handles classical multiplications as well as MAC operations for operands in powers of 2 ranging from 2 to 32 bits. It leverages the D&C strategy using 256 independent 2-bit multipliers as BitBricks.

This recent work [44] presented a re-configurable tightly-coupled DNN co-processor, seamlessly integrated into a Parallel Ultra-Low-Power (PULP) cluster[2]. Structured

[1] $RV32IMC$ stands for 32-bit base RISC-V instruction set ($RV32$) (which is the base set for a 32-bit processor that includes only simple operations, such as jump/branch management, addition, interrupt callback, load/store, shift, logical, memory ordering), with integer (I) and multiplication/division instructions (M).

[2] PULP is an open-source RISC-V computing platform meticulously engineered to prioritize

as a systolic array consisting of sixteen PS processing elements (PEs), the proposed co-processor performs inference using various integer fixed-point data types, ranging from INT16 to INT2, and supports training with FP16 precision. Indeed, each PE is capable of performing four/eight/sixteen parallel MAC operations on 8/4/2 bits, respectively, or computing a single FP16/INT16 MAC. At the heart of each PE lies a PS multiplier which comprises a 16-bit multiplier, utilized for both inference and training, along with four 8-bit PS multiplier trees, solely employed for inference. A tree is a full-fledged D&C multiplier like the one discussed in [49]. Training is a distinguishing feature of this work compared to others, like [42], which instead utilize the Arithmetic Logic Unit (ALU) instead of a custom FP multiplier. There are also other solutions for training in low-precision on MCUs, e.g., [81, 82], but this thesis focuses only on inference.

Regarding bit-serial approaches, in their research outlined in [45], the authors introduced fine-grained PSMAC operations thanks to a bit-serial 16-bit multiplier and a vector extension to the RISC-V ISA. Their design allows to vary weights from 1 to 16 bits, while maintaining activations at a constant 16-bit precision.

While the D&C approach requires a larger and more intricate multiplier structure, resulting in suboptimal area utilization, and the serial approach inherently compromises performance for finer bitwidth granularity, our research opts for a balanced alternative: the SWP approach.

There are also SoCs integrating PS hardware accelerators and RISC-V processors in the same chip [61, 83–86]; however, we concentrate our efforts solely on individual processors rather than more complex systems.

For interested readers, we provide references to some surveys covering the latest research progress on RISC-V ISA extensions [87], efficient acceleration of DL inference on resource-constrained edge devices [72], and the field of TinyML [88].

high energy efficiency. It comprises eight RISC-V cores, a heterogeneous cluster interconnect for memory access, dataflow, and custom co-processors control [80].

Chapter 3

Precision-Scalable Multipliers: Sum-Together (ST) Multipliers

Some of the work described in this chapter was also previously published in [24, 26–28].

3.1 ST Multipliers

The new ST multipliers that we introduce, as well as all the others that we analyze in this manustript, have I/O signals and behave as the reference component described in Fig. 3.1 and Table 3.1. Depending on the $CONFIG$ configuration signal: a) this multiplier 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; b) 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 [7]; others are image processing applications that work with high-resolution satellite images, or high dynamic range (HDR) images and super-resolution [89]. 8 bits is the default precision to quantize DNNs while avoiding performance degradation [89] 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 retain performance for most applications [1, 90]. 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

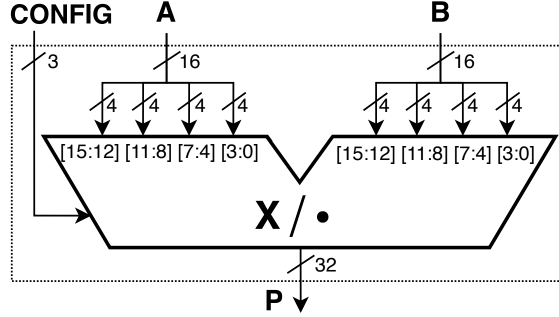


Figure 3.1: Reference ST multiplier. The 16-bit inputs (A and B) are partitioned in 4-bit chunks to enable multiple operations, as defined by the 3-bit configuration input ($CONFIG$) and shown in Table 3.1. The $X/\boldsymbol{\cdot}$ symbol indicates that the multiplier is capable of multiplication and *dot-product* operations, depending on the configuration. Image taken from [28].

Table 3.1: Supported precision configurations and operations of the reference ST multiplier of Fig. 3.1. 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)	$P_{[9:0]} = A_{[15:12]} \times B_{[3:0]} + A_{[11:8]} \times B_{[7:4]} + A_{[7:4]} \times B_{[11:8]} + A_{[3:0]} \times B_{[15:12]}$

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 [29, 42], and can also be found in commercial ML frameworks such as TensorFlow Lite Micro

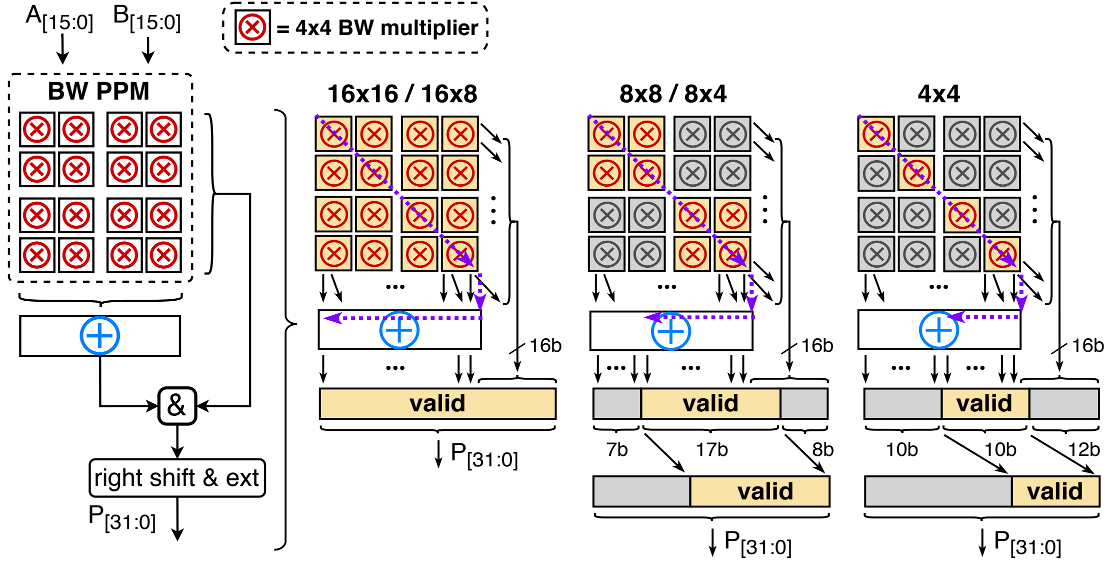


Figure 3.2: Our version of BW ST [25]: architecture overview (left), PPM reconfiguration (right). Image taken from [28].

(TFLM)[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 3.1. This is important to guarantee fair comparisons in all our experiments of Sec. 3.2. Lastly, we present the three newly proposed ST multipliers.

3.1.1 SoA ST multipliers

The original SWP **BW ST multiplier of [25]** is composed of a reconfigurable partial product matrix (PPM) and a final RCA [50]. 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 [25] (made with a structural RTL description) is reported on the left side of Fig. 3.2. 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

[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.

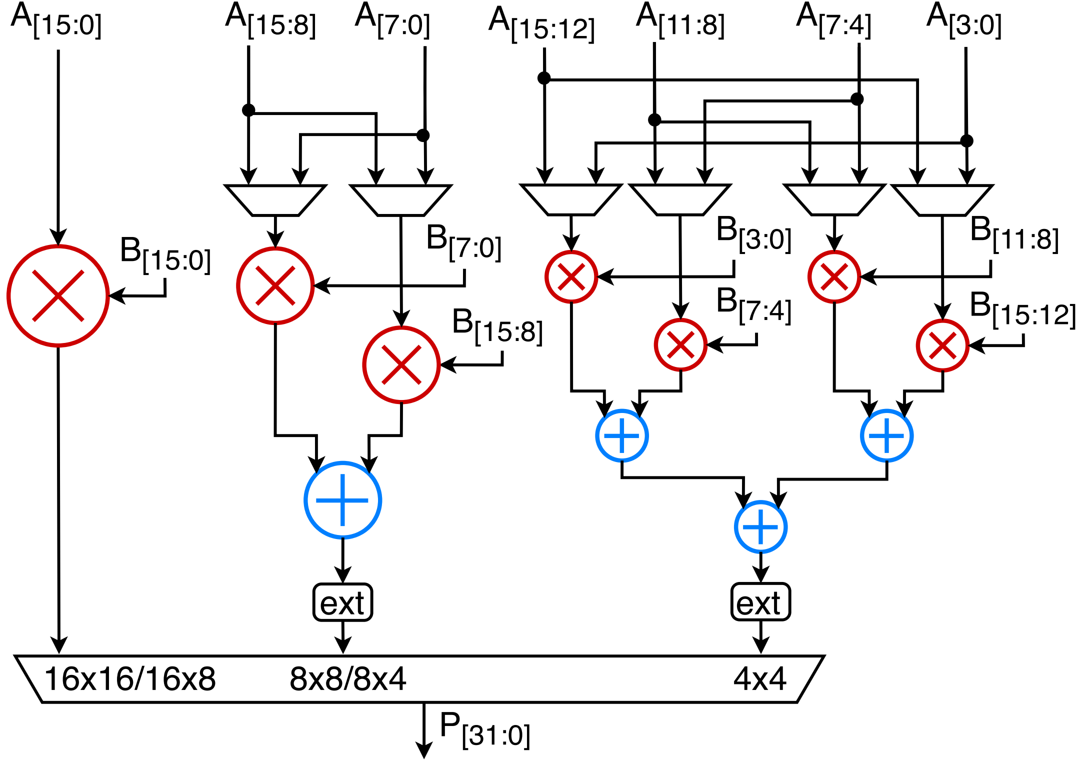


Figure 3.3: Our version of ST multiplier [41]: 16-bit high-precision multiplier (left), 8-bit and 4-bit low-precision dot-product units (middle, right). Image taken from [28].

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 LSB position, sign-extending it to 32 bits (*right shift & ext* block). On the right side of Fig. 3.2, we illustrate how the PPM is reconfigured in the five operating modes of Table 3.1. In the 16×16 and 16×8 modes, all the partial products (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. 3.2—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. 3.2) or four 4×4 (4×4 in Fig. 3.2) 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 [25], and generate the invalid (grey) output bits.

The **multiplication units of Ri5cy [41] and [42]** have been already discussed in Sec. 2.3. In our work, we implement the high-precision fixed-point multiplier

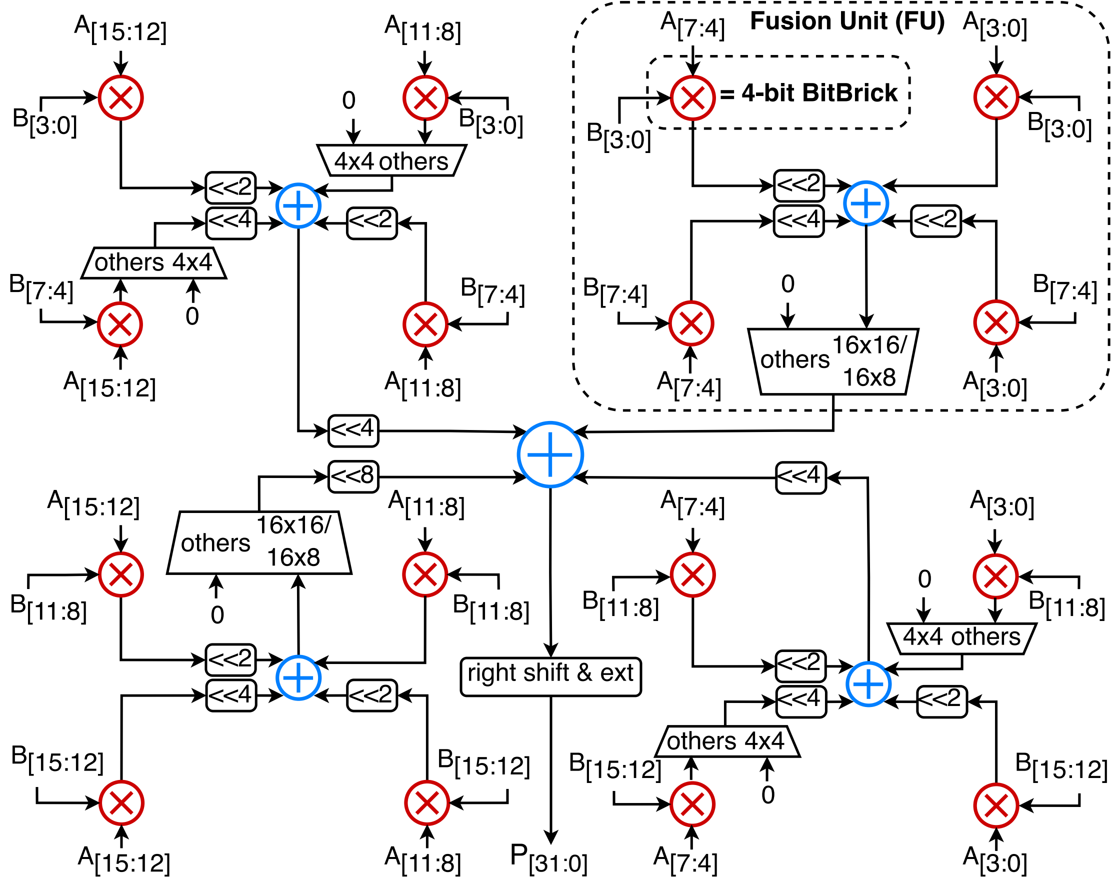


Figure 3.4: Our version of D&C ST [49]: four FUs with four 4-bit BitBricks each, interconnected by shift-and-add logics.

and the two low-precision dot-product units of [41] 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. 3.3, uses a behavioral RTL description, as the authors declared that it gives the synthesizer the maximum optimization freedom [41].

The FU of Bit Fusion [37] 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 [37] are proposed in [32] and [30], which reduce complexity and reconfigurability overhead of the shift-and-add logic at the expense of a lower number of supported input/weight precisions (2x2, 4x4, 8x8). However, the ancestor of all these D&C architectures is the reconfigurable and parallel inner-product processor of [49]. This uses larger BitBricks on 4 or 8 bits and a higher input precision. In fact, each of the two input operands

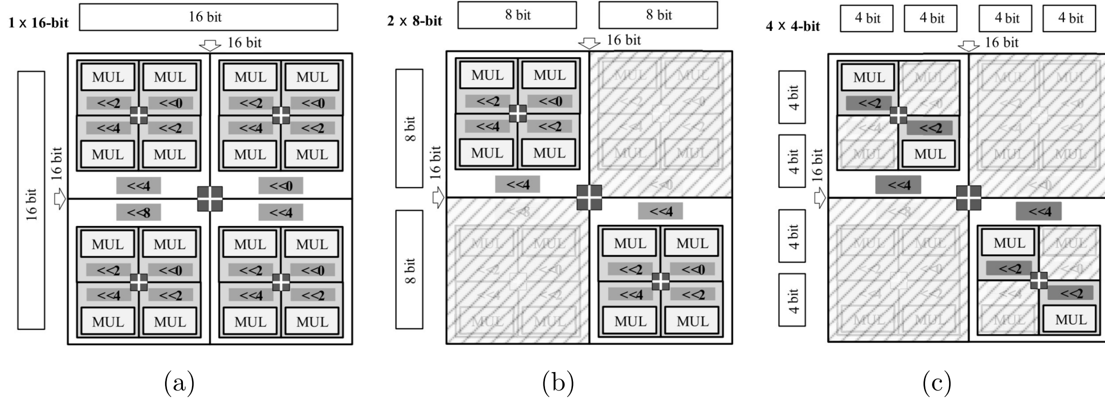


Figure 3.5: (a) 16-bit, (b) 8-bit, and (c) 4-bit operating modes of our version of [49]. Images modified from [33].

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 across different configurations. This contrasts with the D&C architectures in [30, 32, 37], which suffer from memory bandwidth explosion at reduced precision, as noted in [29]. Among these D&C ST multipliers we implement the **FU from [49]** 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. 3.4: 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 [25]). Fig. 3.5 illustrates how our re-implementation of the D&C ST multiplier inspired from reference [49] is reconfigured in the main operating modes outlined in Table 3.1. While at full-precision all the FUs are enabled and contribute to the multiplier's output (Fig. 3.5a), at low-precision the FUs and BitBricks outside the left-to-right diagonal are progressively disabled/gated (Fig. 3.5b–c).

The **reconfigurable fixed-point multiplier of [48]** 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 $16\times16 \pm 16\times16$ with saturation, one or two $16\times16 \pm 16\times16 + 16$ without saturation, and one $8\times8 + 8\times8 + 8\times8 + 8\times8$). In our version of [48], 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

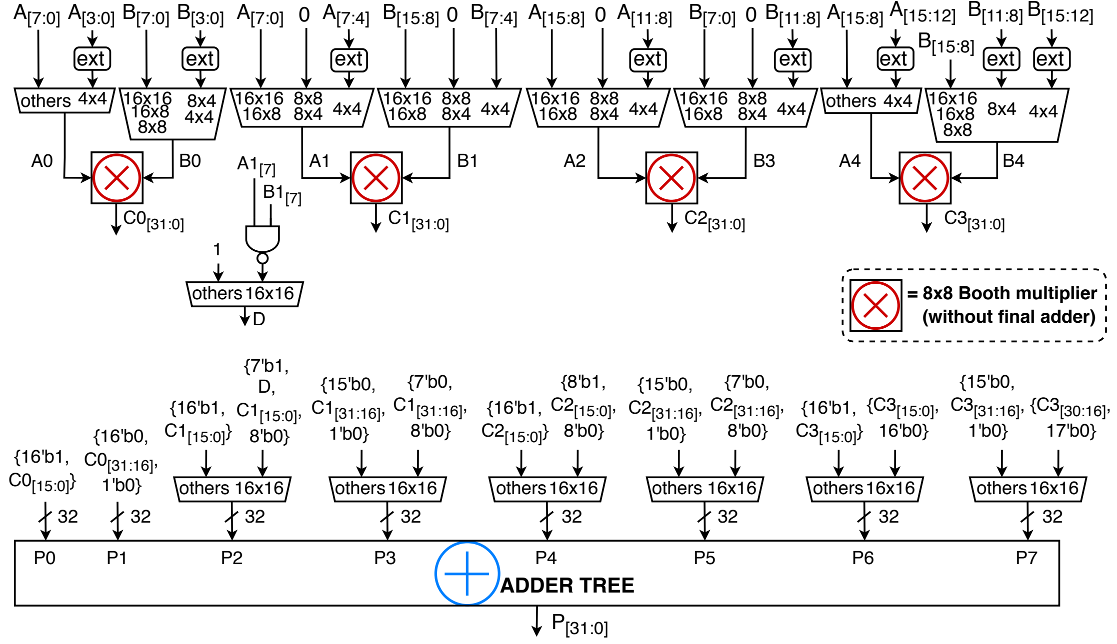


Figure 3.6: Our version of ST multiplier [48]: four 8-bit Booth multipliers interconnected by muxes ending with an adder tree. Image taken from [28].

operand B: A[15:8]×B[15:8] ± aL[7:0]×bL[7:0] of[48] becomes A[15:8]×B[7:0] + A[7:0]×B[15:8], as reported in Table 3.1. We also scale down maximum and minimum precision to 16 and 4 bits, respectively. The resulting architecture, shown in Fig. 3.6, features four 8-bit Booth multipliers connected by a network of multiplexers ending with an adder tree.

At last, within all SoA ST multipliers that do not natively support the asymmetric configurations 16×8 and 8×4 (i.e., [25, 41, 49]), we add a sign-extension logic (not shown in Fig. 3.2–3.6 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.

3.1.2 Booth ST: a radix-4 Booth ST multiplier

We propose a novel ST multiplier with a Booth architecture [50] that supports operands at 16-, 8-, and 4-bit precision as the reference ST multiplier in Table 3.1. Fig. 3.7 illustrates Booth ST: it is composed of a lightweight reconfiguration logic (drawn in blue) placed between the two input operands and a standard Radix-4 16-bit Booth multiplier (drawn in white and gray), featuring a Wallace's reduction

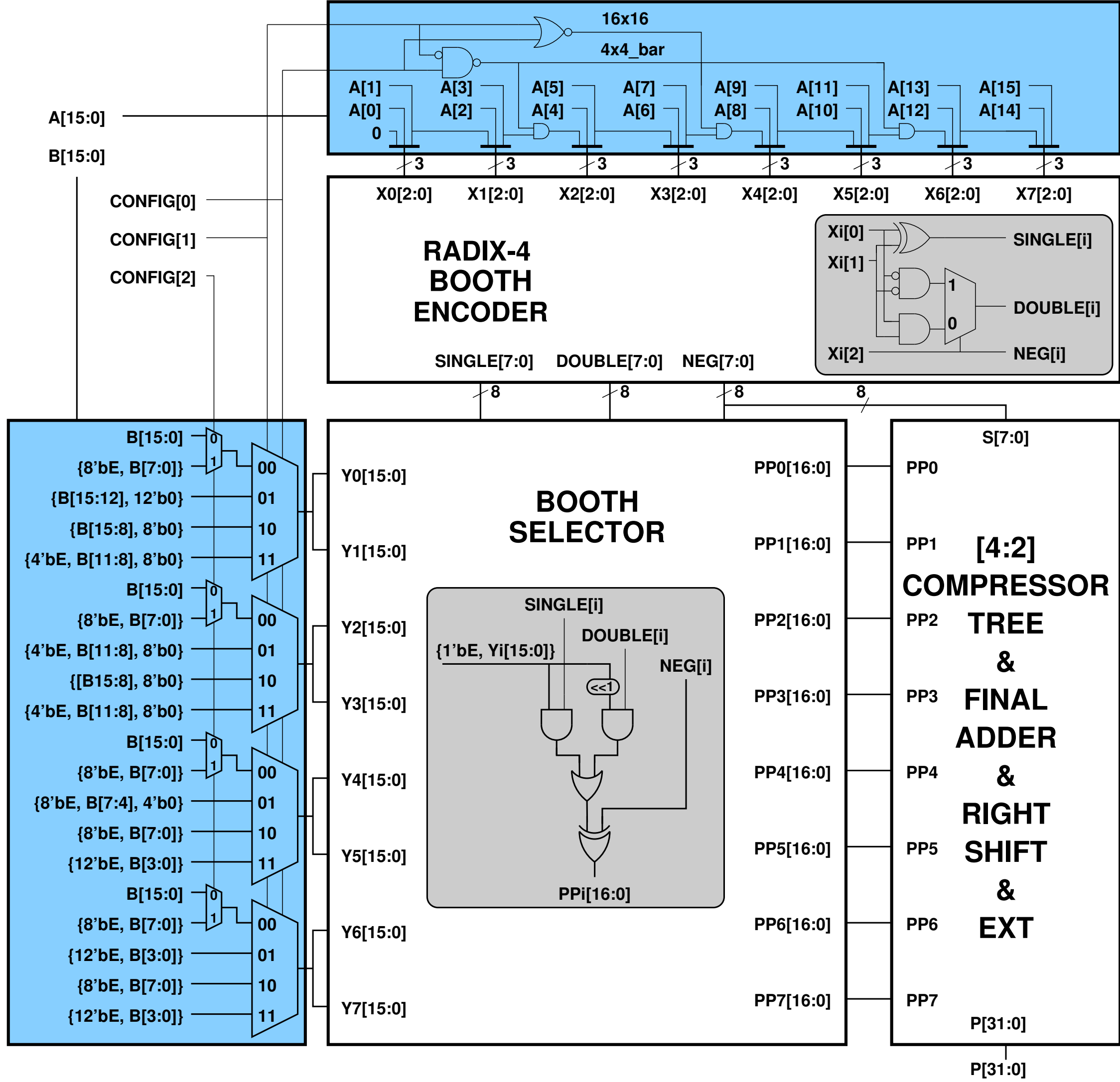


Figure 3.7: Radix-4 Booth ST (as in [26, 28]): reconfiguration logic (blue), 16-bit Booth multiplier (white and gray). Image taken from [28].

tree with 4:2 compressors and a Carry Propagate Adder with Prefix Network [50].

The main difference with respect to the majority of SoA ST multipliers is that our design does not require a dedicated adder to sum the low-precision products together, but it exploits the normal alignment of PPs within the multiplier structure. In fact, as shown in Fig. 3.8 for configuration 16×16/16×8 (Fig. 3.8a), 8×8/8×4 (Fig. 3.8b), and 4×4 (Fig. 3.8c), the bits of the output P (yellow circles) are obtained by vertically summing the full-colored circles representing the bits of the eight PPs (PP0–PP7). These full-colored bits are the result of the products between input operands with the same color, whereas the half-colored bits are gated by the reconfiguration logic.

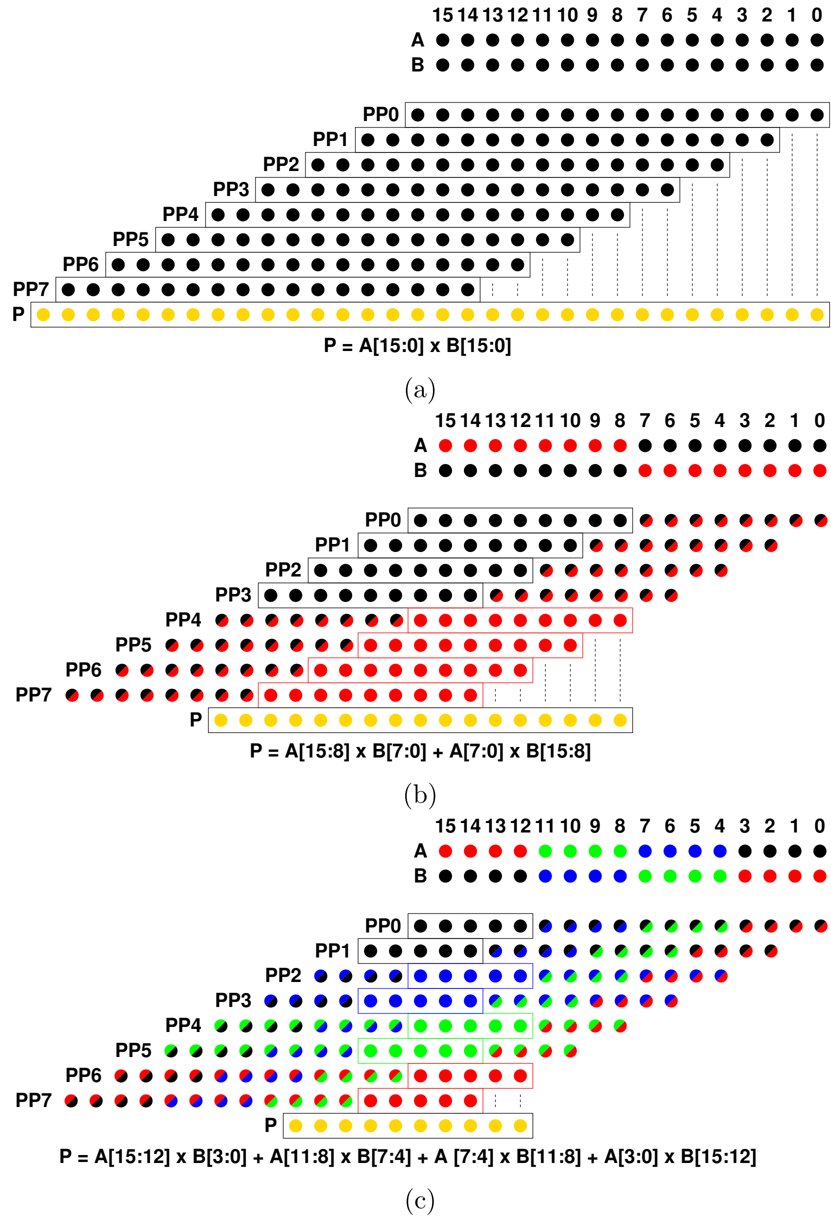


Figure 3.8: Alignment of PPi partial products for $CONFIG$ 16×16/16×8 (a), 8×8/8×4 (b) and 4×4 (c). Images taken from [26].

Through the configuration signal $CONFIG$, the reconfiguration logic controls: 1) How the bits of operand A are properly composed to form X_0-X_7 input triplets for the encoder; 2) How the sub-words of operand B are arranged and presented to the Y_0-Y_7 inputs of the selector, including the sign extension of the low-precision operands packed in B in case of asymmetric configurations; 3) Which half-colored bits of Fig. 3.8 should be gated; 4) The number of positions to right-shift the output

Listing 3.1: 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 }
```

to the LSB position.

We implement the architecture of this Booth ST multiplier with a structural description [26].

3.1.3 BW-ADD: a Baugh-Wooley ST multiplier with an improved final adder

In the light of the preliminary PPA results of the comparison of SoA ST multiplier of our previous work [24, 26], carried out in a 28-nm CMOS technology at 0.9V, we observed that the BW ST multiplier [25] was particularly area-efficient at clock frequencies lower than 600 MHz in the area vs clock period plot. 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. 3.2, are responsible for a significant area degradation [24], since the logic synthesizer uses logic gates with larger driving strength and/or super-low threshold to meet the stricter timing constraints. Thus, in [28] 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 [25].

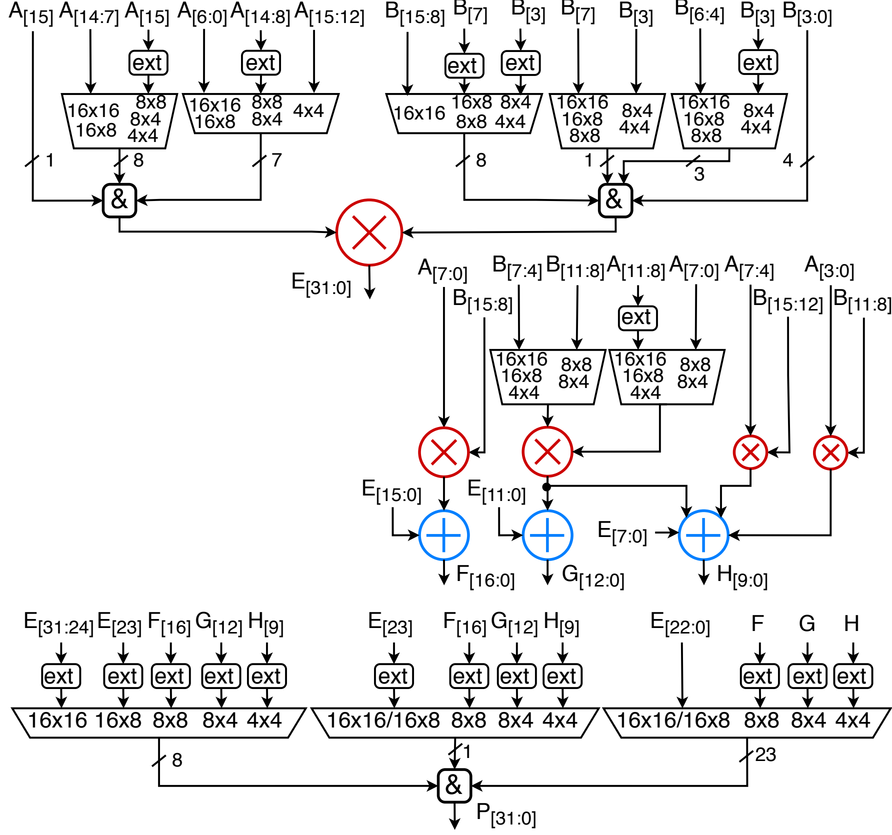


Figure 3.9: HLS ST derived from HLS (as in [28]): four multipliers and three adders interconnected by a network of muxes and concatenations. Image taken from [28].

3.1.4 HLS ST: an ST multiplier derived from HLS

As we present in Sec. 4.2, 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 previously presented ST multipliers of Secs. 3.1.1–3.1.3. As an alternative, we decide to describe the ST functionality at a high-level and let the HLS tool, which in our 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. 3.1. 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 sub-field 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. 3.9, 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.

3.2 Experimental Results

3.2.1 PPA Comparison of ST Multipliers

To compare all the ST multipliers considered in our analysis and identify the best in PPA, we synthesize their RTL descriptions using Synopsys Design Compiler (DC), on a 28-nm CMOS technology at 0.9 V, after adding I/O registers.

Fig. 3.10 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. The reported power is an average of three values obtained when the multipliers are configured in 16-bit mode (16×16 and 16×8), 8-bit mode (8×8 and 8×4) and 4-bit mode (4×4).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. It is important to note that, by subtracting a fixed quantity between $[t1; t2] = [0.06; 0.09]$ ns from the clock period (where $t1$ is the sum of the minimum setup and clock-to-output times, and $t2$ is the sum of the maximum setup and clock-to-output times among all the ST multipliers, respectively), we can approximate the delay of a multiplier. This information is useful when we want to include an ST multiplier into a pipeline stage with other components.

In the area vs clock period graph, the Booth design [26] shares the primacy with [25] 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 [26].

The design of [41] 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.

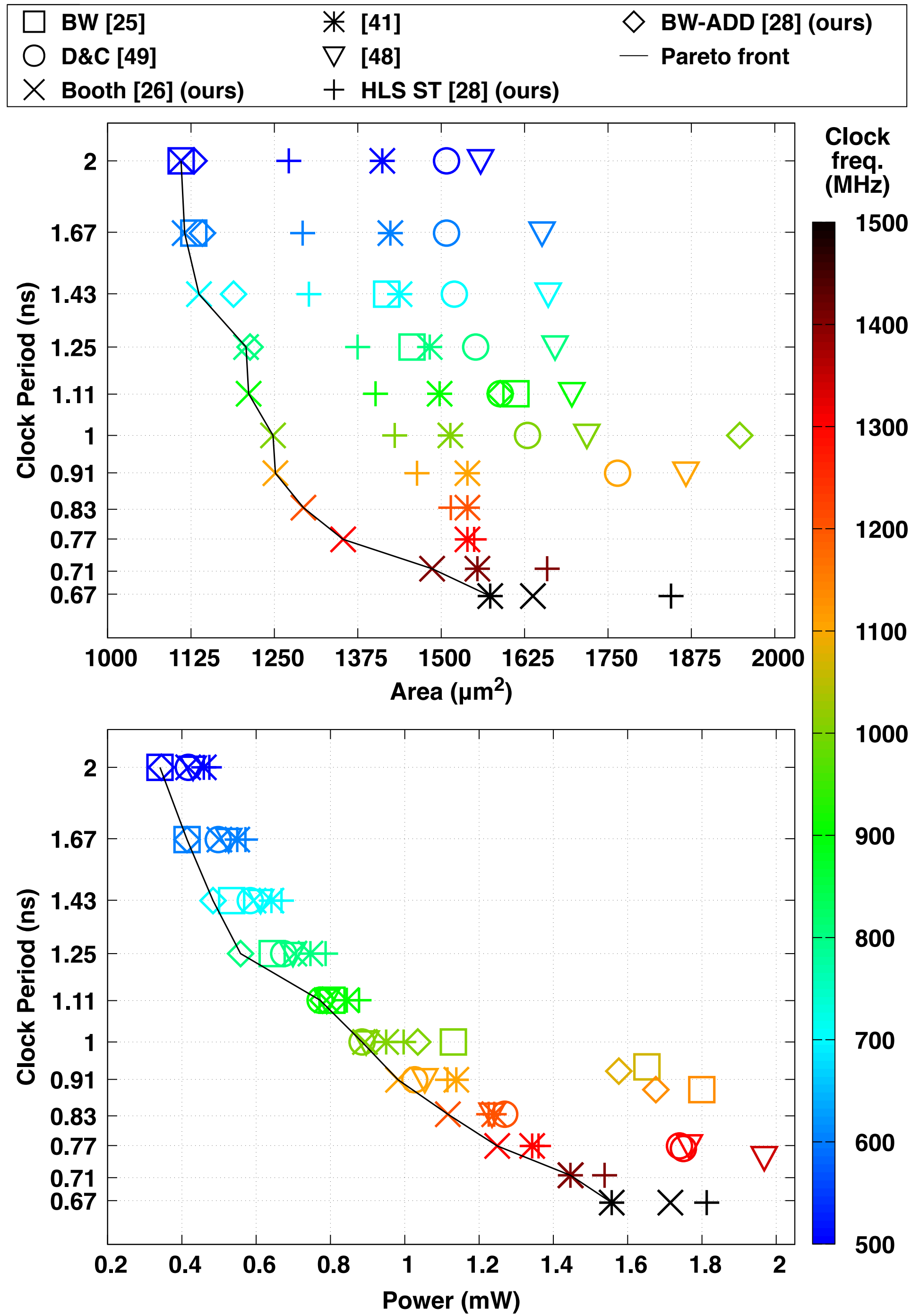


Figure 3.10: DSE of the SoA and newly proposed ST multipliers. Image taken from [28].

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 [25]. Our results confirm that the BW architecture, although very

efficient at low frequencies, is not suitable for higher frequencies[50], 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 [48], [41], 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 SWP manner (like [25], BW-ADD and [26]) 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 [49] 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 500 to 800 MHz, the optimal ST multipliers are those with a BW architecture (e.g., [25] and BW-ADD); from 1000 to 1300 MHz, [26] progressively dominates over [49] and [48]; at high frequencies [41] turns out to be the most power efficient.

To sum up this comparison of ST multipliers, the optimal solutions depend on the PPA constraints: [26] offers the best trade-off in area vs clock period for most of the frequencies, [25] and BW-ADD prove to be Pareto-optimal in power at low frequencies, whereas [26] and [41] are the best in both area and power at high frequencies.

Chapter 4

High-Level Design of ST-Based DNN Hardware Accelerators

Some of the work described in this chapter was also previously published in [24, 28, 39, 40].

4.1 Background

4.1.1 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 [52]. 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 [42], or specialized hardware operators like SWP ST multipliers [26].

In our work 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 (TFLite)), its efficient mapping on existing hardware (e.g., on 8-bit MCUs), and thus its widespread adoption on embedded devices for non-extreme quantization (> 2 bits) [1, 51, 89]. 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 FP ones [91], 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 [51, 52]. Notice that, since we target ST-based accelerators only for the inference phase of DNNs, our focus is on UIQ for inference,

and not for training.

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:

$$x_q = \text{clip}\Big(\text{round}\big(\frac{1}{s}x + z\big), \alpha_q, \beta_q\Big) \tag{4.1}$$

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:

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{4.2}$$

$$z = \text{round}\big(\frac{\beta\alpha_q - \alpha\beta_q}{\beta - \alpha}\big) \tag{4.3}$$

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

$$\hat{x} = s(x_q - z) \tag{4.4}$$

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 (4.1) performs only the scale transformation. In this case, the quantization mapping is commonly known as *scale* or *symmetric* [92] 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*.

Integer-only DNN kernels

Now consider the expression of an FC layer:

$$Y_k = b_k + \sum_{c=1}^{C} X_c W_{c,k} \qquad \forall\, k \in [1, K] \tag{4.5}$$

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 (4.4) to each of the four real variables in (4.5), 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:

$$\begin{aligned} Y_{q,k} = \underbrace{z_Y}_{(a)} + \underbrace{\frac{s_b}{s_Y}(b_{q,k} - z_b)}_{(b)} + \frac{s_X s_W}{s_Y} \Bigg[\underbrace{\left(\sum_{c=1}^{C} X_{q,c} W_{q,c,k} \right)}_{(c)} \\ - \underbrace{\left(z_W \sum_{c=1}^{C} X_{q,c} \right)}_{(d)} - \underbrace{\left(z_X \sum_{c=1}^{C} W_{q,c,k} \right)}_{(e)} + \underbrace{C z_X z_W}_{(f)} \Bigg] \quad \forall\, k \in [1, K] \end{aligned} \tag{4.6}$$

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 (4.6) 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 [51, 52], 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 (4.6), 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 (A.2), (A.4), and (4.6) 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. 4.2.2, 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 (4.5), we derive the

expression of the FP FC-ReLU fused layer:

$$R_k = \begin{cases} 0 & \text{if } Y_k < 0 \\ Y_k & \text{if } Y_k \geq 0 \end{cases} \quad \forall\, k \in [1, K] \tag{4.7}$$

where R_k is the k-th output of the ReLU layer. By repeating the same steps that brought to the derivation of (4.6) from (4.5)—applying (4.4) to each real variable of (4.7), 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:

$$R_{q,k} = \begin{cases} z_R & \text{if } T < 0 \\ z_R + s_R \cdot T & \text{if } T \geq 0 \end{cases} \quad \forall\, k \in [1, K] \tag{4.8a}$$

where

$$T = s_b(b_{q,k} - z_b) + s_X s_W \Bigg[\left(\sum_{c=1}^{C} X_{q,c} W_{q,c,k} \right) - \left(z_W \sum_{c=1}^{C} X_{q,c} \right) - \left(z_X \sum_{c=1}^{C} W_{q,c,k} \right) + C z_X z_W \Bigg]. \tag{4.8b}$$

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 (4.6). Notice that $R_{q,k}$ undergoes a round-and-clip operation, not shown for clarity in (4.8), 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.

4.1.2 MLPerf Tiny Benchmark

The Machine Learning Performance (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. In particular, the *Tiny* benchmark [53] 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, MCUs, IoT 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 [88]. 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).*

Visual Wake Words

The VWW dataset [93] is a collection of 109619 96×96 RGB images divided into 53140/56479 images which contain persons/not-person. It is derived from the Microsoft Common Objects in Context (MSCOCO) 2014 dataset [94] which has been pre-processed[1] to resize the images and to assign them to the *person* class when a person occupy at least 2.5% of the frame [53]. 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 [95], that we define *MobileNetV1Tiny*, with 96×96 input image resolution, alpha = 0.25 (i.e., the number of channels in each layer is reduced down to 25% of the original), and two output classes (person and no person). According to [53], the FP MobileNetV1Tiny reaches about 86% of accuracy across the Perf test set of 1000 images and should reach at least 80% after quantization and other optimizations.

Image Classification

The IC benchmark uses the Canadian Institute for Advanced Research, 10 classes (CIFAR-10) dataset [96], 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, IoT sensor nodes, and autonomous agents and vehicles. The model to use is a custom ResNetV1 [97], 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. The FP model of ResNetV1Tiny achieves 86.5% of accuracy across the 200 Perf test images and should retain at least 85% after quantization and other optimizations [53].

Keyword Spotting

The KS benchmark is derived from the Speech Commands v2 dataset [98], a collection of 105,829 English words spoken by 2,618 persons with various accents[2]. 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. By default the audio feature representation used in this benchmark is the Mel-Frequency Cepstral Coefficients (MFCC), a technique that converts audio signals to the frequency domain

[1] The pre-processing script, *buildPersonDetectionDatabase.py* can be found in the GitHub repository from Silicon Labs https://github.com/SiliconLabs/platform_ml_models/tree/master/eembc/Person_detection. Accessed on: Mar 9, 2024.

[2] Speech Commands v2 is also directly available in TensorFlow: https://www.tensorflow.org/datasets/catalog/speech_commands. Accessed on: Jan 19, 2024.

(Mel Spectrogram) using the Mel scale, which is a non-linear frequency perceptual scale of pitches judged by listeners to be equal in distance from one another. The Mel spectrogram is then transformed using a Discrete Cosine Transform to obtain the MFCCs which can be used to train the neural network [53]. 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 [99]. According to [53], the FP model of DS-CNN has 92.2% accuracy on the 1000 utterances of the Perf test set and should not fall below 90% after quantization and other optimizations.

Anomaly Detection

This benchmark uses one of the six machine types present in the dataset of the 2020 edition of Detection and Classification of Acoustic Scenes and Events (DCASE) competition [100], the toy-car machine type (ToyADMOS [101]), 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 [100] (thus, we name it *FC-AutoEncoder*). Encoder and decoder have four FC layers of 128 neurons with BN layers and ReLU activation, the bootleneck layer has 8 neurons, input and output layers have 640 neurons. Since the audio is too long for the model, it is pre-processed and divided into a log-mel-spectrogram with 128 bands of 32 ms of length. Then, the model processes five bands at a time (128x5=640) with a sliding window approach [53]. 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). The FP FC-AutoEncoder has an AUC of 0.88, whereas after quantization and optimizations it should reach at least 0.85.

4.2 ST-based Hardware Accelerators

4.2.1 Working Principle

We now illustrate the working principle of our three DNN accelerators integrating ST multipliers in their MAC units. Fig. 4.1 shows the different access patterns (red) that the 2D-Conv, DW-Conv and FC accelerators use to read data from the activation (blue) and weight (orange) tensors, and how these data are packed in the 16-bit inputs of the ST multipliers.

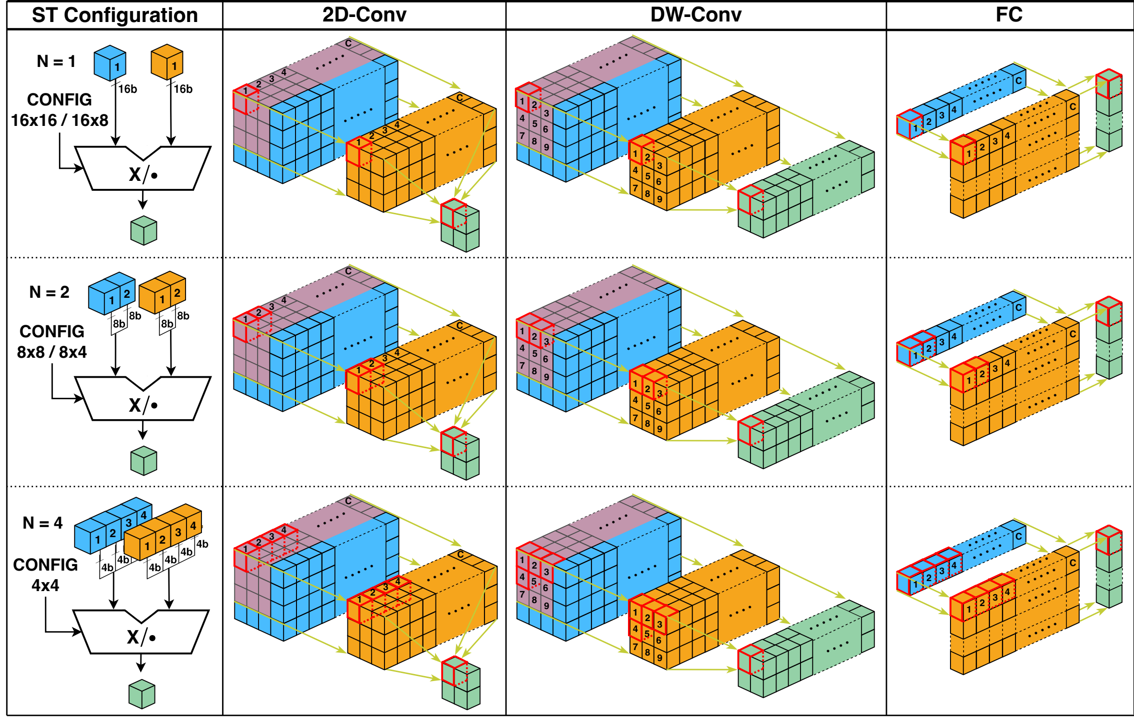


Figure 4.1: Working principle of our ST-based DNN accelerators: 2D-Convolution (2D-Conv), Depth-wise Convolution (DW-Conv) and Fully-Connected (FC). Image modified from [28].

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. 4.1 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 $1/N$.

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. 4.1.

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. 4.4.2. 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. 4.1 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 $\lceil K^2/N \rceil / K^2$, which typically is greater than $1/N$, with this overhead decreasing as K increases [40].

FC Accelerator

The working principle of this accelerator is shown in the last column of Fig. 4.1. 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 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.

4.2.2 Accelerators Architecture

Our ST-based DNN accelerators share the same general architecture, outlined in the grey rectangle of Fig. 4.2. It consists of four parts as illustrated later in Secs. 4.2.2–4.2.2: 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. 4.2, 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. 4.3.3.

Even though it is not the focus of our work, we assume that the three accelerators share an on-chip global buffer (not shown in Fig. 4.2), as shown for example in [60]

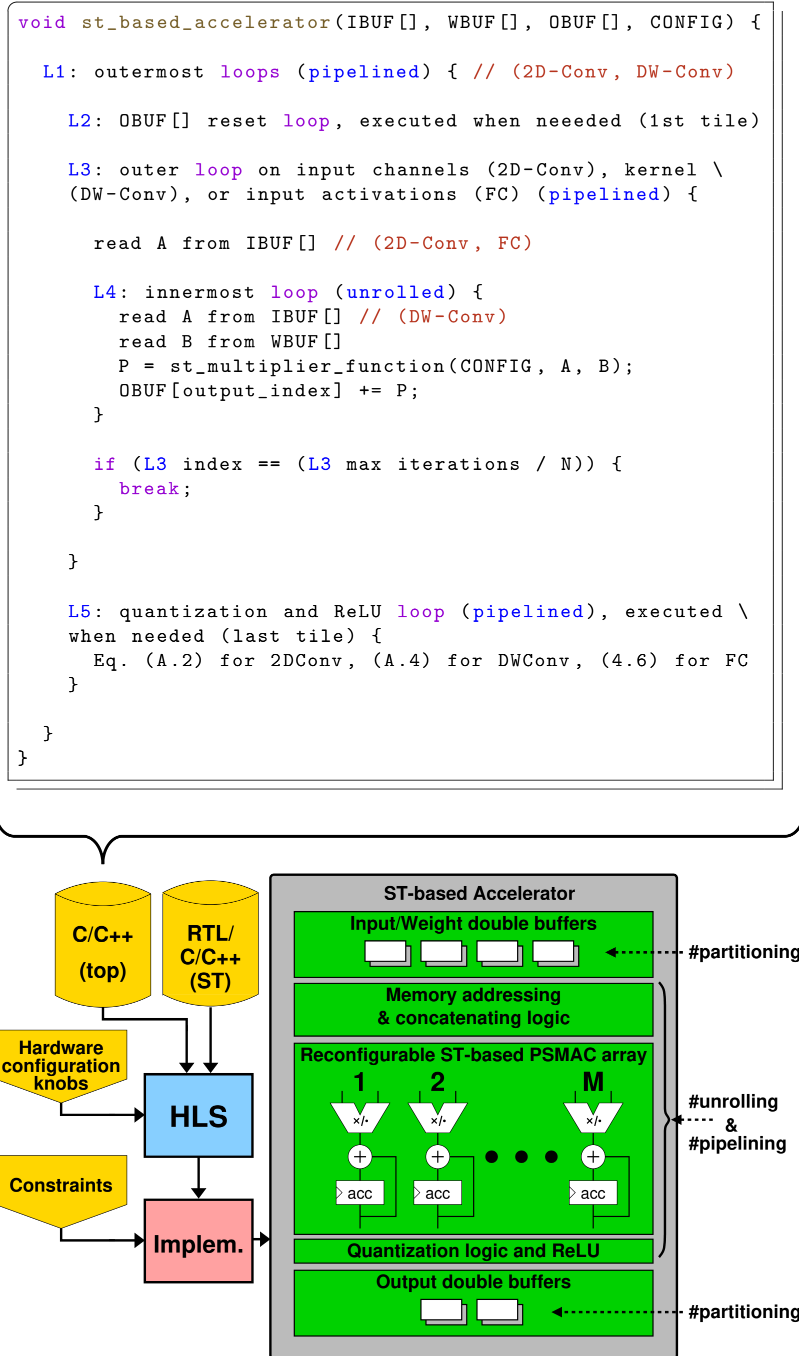


Figure 4.2: General architecture of the ST-based accelerators (bottom right), HLS flow (bottom left), and pseudo-code of the high-level C/C++ description (top) that produces the general architecture. Image taken from [28].

Table 4.1: 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. 4.4.1, denoted by the *DSE* entry, are listed in Table 4.2.

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

and other more recent papers on ML SoCs [85, 102]. 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 [103] 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 4.1 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 [39, 40] that are available in public Model Zoos for computer vision applications, such as TensorFlow [104] [105], Intel [106] [107], Xilinx [108], and Nvidia [109] [110]. 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 [39, 40]. 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).

Table 4.2: Hardware configuration knobs explored in the DSE of Sec. 4.4.1, including maximum tiles size, HLS directives, and implementation constraints.

Hardware config. 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 the *st_multiplier_function* to ST multiplier IP	*map_to_operator* *<component name (X)>*	yes → *IP mode*, $X = \{$[25], [26], [41], [49], [48], BW-ADD, HLS ST $\}$ / / no → *Inline mode*, i.e., the *st_multiplier_function* is inlined		
Target clock frequency		[100 ÷ 1000] MHz, 10 steps		

Regarding the input tile channels (IC)/input tile activations (IA) and output tile channels (OC)/output tile activations (OA), we vary their size during the DSE of ST-based accelerators as discussed in Sec. 4.4.1. The values explored are in the first two rows of Table 4.2, 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 [25], 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.

Let us now comment on the pseudo-code at the top of Fig. 4.2. 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. 4.1, 4.2, and 4.3, 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. 4.1–4.3).

Since L4 is unrolled, the HLS tool synthesizes it by generating the array of M parallel reconfigurable ST-based PSMAC units shown in Fig. 4.2. To comply with the working principle presented in Sec. 4.2.1, 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 sinput activations for FC, of the current tile execution. This is implemented by variables *ic_lim*, *k_lim*, and *ia_lim* in Lsts. 4.1, 4.2, and 4.3, 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. A break condition is also present at the end of each of the other *for* loops to guarantee the processing of tiles with dimensions smaller than the maximum values reported in Table 4.1 (i.e., leftovers handling). However, these break conditions have not been reported in the pseudo-codes to enhance clarity.

Finally, only when the accelerator creates the last output tile, $OBUF$ undergoes quantization using the corresponding UIQ formula (i.e, Eq. (A.2) for 2D-Conv, (A.4) for DW-Conv, (4.6) for FC), followed by ReLU (when 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 [60].

Our accelerators are latency-insensitive and stall until input data (i.e., features and weights) is not available in the internal buffers. However, when the inputs are ready, the accelerators complete their execution in a fixed amount of time.

Below we delve into the details of each architectural block of the ST-based accelerator illustrated in Fig. 4.2, highlighting the key differences between the three accelerators.

Listing 4.1: Pseudo-code of our ST-based 2D-Conv accelerator.

```
1 #include <ac_int.h>
2
3 #pragma map_to_operator "X"
4 int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}
5
6 void conv2d(
7         int4 IBUF_A[IS], IBUF_B[IS], IBUF_C[IS], IBUF_D[IS],
8         int4 WBUF_A[WS], WBUF_B[WS], WBUF_C[WS], WBUF_D[WS],
9         ac_int<28, true> OBUF[OS],
10        uint3 CONFIG, uint1 RESET) {
11 int ic_lim;
12 if (CONFIG==(8x8 || 8x4)) { ic_lim = IC/2-1; }
13 else if (CONFIG==4x4) { ic_lim = IC/4-1; }
14 else { ic_lim = IC-1; }
15
16 #pragma pipeline_init_interval 1
17 for (int oh=0; oh<OH; oh++) {
18  #pragma pipeline_init_interval 1
19  for (int ow=0; ow<OW; ow++) {
20   #pragma pipeline_init_interval 1
21   for (int oc=0; oc<OC; oc++) {
22    if (RESET==1) { OBUF[OC*(OH*oh+ow)+oc] = 0; }
23   }
24   #pragma pipeline_init_interval 1
25   for (int ic=0; ic<IC; ic++) {
26    #pragma pipeline_init_interval 1
27    for (int kh=0; kh<KH; kh++) {
28     #pragma pipeline_init_interval 1
29     for (int kw=0; kw<KW; kw++) {
30      // Memory addressing and concatenating logics for A
31      int A_idx = IC*(IW*(oh+kh)+(ow+kw))+ic;
32      int4 A_HH = IBUF_A[A_idx];
33      int4 A_HL = IBUF_B[A_idx];
34      int4 A_LH = IBUF_C[A_idx];
35      int4 A_LL = IBUF_D[A_idx];
36      int16 A = ((A_HH<<12)&0xF000) | ((A_HL<<8)&0x0F00) |
37                ((A_LH<< 4)&0x00F0) | ( A_LL    &0x000F);
38      #pragma unroll OC
39      for (int oc=0; oc<OC; oc++) {
40       // Memory addressing and concatenating logics for B
41       int B_idx = OC*(IC*(KH*kh+kw)+ic)+oc;
42       int4 B_HH = WBUF_A[B_idx];
43       int4 B_HL = WBUF_B[B_idx];
44       int4 B_LH = WBUF_C[B_idx];
45       int4 B_LL = WBUF_D[B_idx];
46       int16 B = ((B_HH<<12)&0xF000) | ((B_HL<<8)&0x0F00) |
47                 ((B_LH<< 4)&0x00F0) | ( B_LL    &0x000F);
48       // Reconfigurable ST-based PSMAC array
49       int28 P = st_multiplier_function(CONFIG,A,B);
50       OBUF[OC*(OH*oh+ow)+oc] += P;
51      } //oc
52     } //kw
53    } //kh
54    if (ic==ic_lim) { break; }
55   } //ic
56   #pragma pipeline_init_interval 1
57   for (int oc=0; oc<OC; oc++) {
58    // Quantization logic (Eq. A.2) and ReLU (when needed)
59    ...
60   }
61  } //ow
62 } //oh
63 }
```

Listing 4.2: Pseudo-code of our ST-based DW-Conv accelerator.

```
1 #include <ac_int.h>
2
3 #pragma map_to_operator "X"
4 int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}
5
6 void dwconv(
7         int4 IBUF_A[IS], IBUF_B[IS], IBUF_C[IS], IBUF_D[IS],
8         int4 WBUF_A[WS], WBUF_B[WS], WBUF_C[WS], WBUF_D[WS],
9         ac_int<28, true> OBUF[OS],
10        uint3 CONFIG, uint1 RESET) {
11  int k_lim,i0,i1,i2,i3,j0,j1,j2,j3;
12  if (CONFIG==(8x8 || 8x4)) { k_lim = (KH*KW)/2; }
13  else if (CONFIG==4x4) { k_lim = (KH*KW)/4; }
14  else { k_lim = (KH*KW)-1; }
15
16  #pragma pipeline_init_interval 1
17  for (int oh=0; oh<OH; oh++) {
18   #pragma pipeline_init_interval 1
19   for (int ow=0; ow<OW; ow++) {
20    #pragma pipeline_init_interval 1
21    for (int oc=0; oc<OC; oc++) {
22     if (RESET==1) { OBUF[OC*(OH*oh+ow)+oc] = 0; }
23    }
24    #pragma pipeline_init_interval 1
25    for (int k=0; k<KH*KW; k++) {
26     #pragma unroll OC
27     for (int oc=0; oc<OC; oc++) {
28      // Memory addressing and concatenating logics for A
29      i0,i1,i2,i3 = LUT_i(CONFIG,k);
30      j0,j1,j2,j3 = LUT_j(CONFIG,k);
31      int4 A_HH = IBUF_A[OC*(IN_W*(oh+i0)+(ow+j0))+oc];
32      int4 A_HL = IBUF_B[OC*(IN_W*(oh+i1)+(ow+j1))+oc];
33      int4 A_LH = IBUF_C[OC*(IN_W*(oh+i2)+(ow+j2))+oc];
34      int4 A_LL = IBUF_D[OC*(IN_W*(oh+i3)+(ow+j3))+oc];
35      int16 A = ((A_HH<<12)&0xF000) | ((A_HL<<8)&0x0F00) |
36                ((A_LH<< 4)&0x00F0) | ( A_LL    &0x000F);
37      // Memory addressing and concatenating logics for B
38      int4 B_HH = WBUF_A[OC*(KH*i3+j3)+oc];
39      int4 B_HL = WBUF_B[OC*(KH*i2+j2)+oc];
40      int4 B_LH = WBUF_C[OC*(KH*i1+j1)+oc];
41      int4 B_LL = WBUF_D[OC*(KH*i0+j0)+oc];
42      int16 B = ((B_HH<<12)&0xF000) | ((B_HL<<8)&0x0F00) |
43                ((B_LH<< 4)&0x00F0) | ( B_LL    &0x000F);
44      if ((k==k_lim)&&(CONFIG==(8x8 || 8x4)||CONFIG==4x4)) {
45        // Force two or three 4-bit chunks of a and b
46        // to zero in place of the missing elements
47        ...
48      }
49      // Reconfigurable ST-based PSMAC array
50      int28 P = st_multiplier_function(CONFIG,A,B);
51      OBUF[OC*(OH*oh+ow)+oc] += P;
52     } //oc
53     if (k==k_lim) { break; }
54    } //k
55    #pragma pipeline_init_interval 1
56    for (int oc=0; oc<OC; oc++) {
57      // Quantization logic (Eq. A.4) and ReLU (when needed)
58      ...
59    }
60   } //ow
61  } //oh
62 }
```

Listing 4.3: Pseudo-code of our ST-based FC accelerator (inspired by [25]).

```
#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. 4.6) and ReLU (when needed)
   ...
 }
}
```

Internal Buffers

Part III of Table 4.1 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.

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. 4.1. This is visible from the *int4* datatype in the function signatures of Lsts. 4.1–4.3. 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. 4.2.2.

To guarantee the proper accelerators' execution, the internal buffers are filled by an external Direct Memory Access (DMA) engine following the working principle illustrated in Fig. 4.1. 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. 4.3) for a reason clarified in Sec.4.2.2. 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. 4.2.2. 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. For the remaining internal buffers of DW-Conv (IBUF_C, IBUF_D, WBUF_A, and WBUF_B), and the internal buffers of the other two accelerators, we use SRAM ASIC memories. The type of memory to use in the accelerators can be specified in the HLS tool. However, the tool does not generate the actual memories, but the interface signals and protocols towards them. The designer is in charge of actually creating those memories and physically connecting them to the accelerators. We limit ourselves to the generation of the interface signals. However, in our PPA experiments of Sec. 4.4.1, we take into account the area and power consumption of the accelerators internal buffers.

Memory Addressing and Concatenating Logic

These two logic circuits are designed to implement the working principles outlined in Sec. 4.2.1. Depending on the type of accelerator and selected configuration (as depicted in Fig. 4.1), 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. 4.1 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. 4.1). For DW-Conv, these logic circuits are a bit more complex. Indeed, a pair of 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. 4.2). Moreover, as already discussed in Sec. 4.2.1, 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. 4.2). We report in Fig. 4.3 an example of the behavior of the memory addressing and concatenating logic acting on the four 4-bit memory banks of the internal input buffer (IBUF_A/B/C/D) of DW-Conv, already filled by an external DMA. Similar approaches apply for WBUF and the internal buffers of the other two accelerators.

Reconfigurable ST-based PSMAC array

The PSMAC array of our ST-based accelerators contains M MAC units, as shown in Fig. 4.2. Each MAC unit works on a distinct output channel/activation,

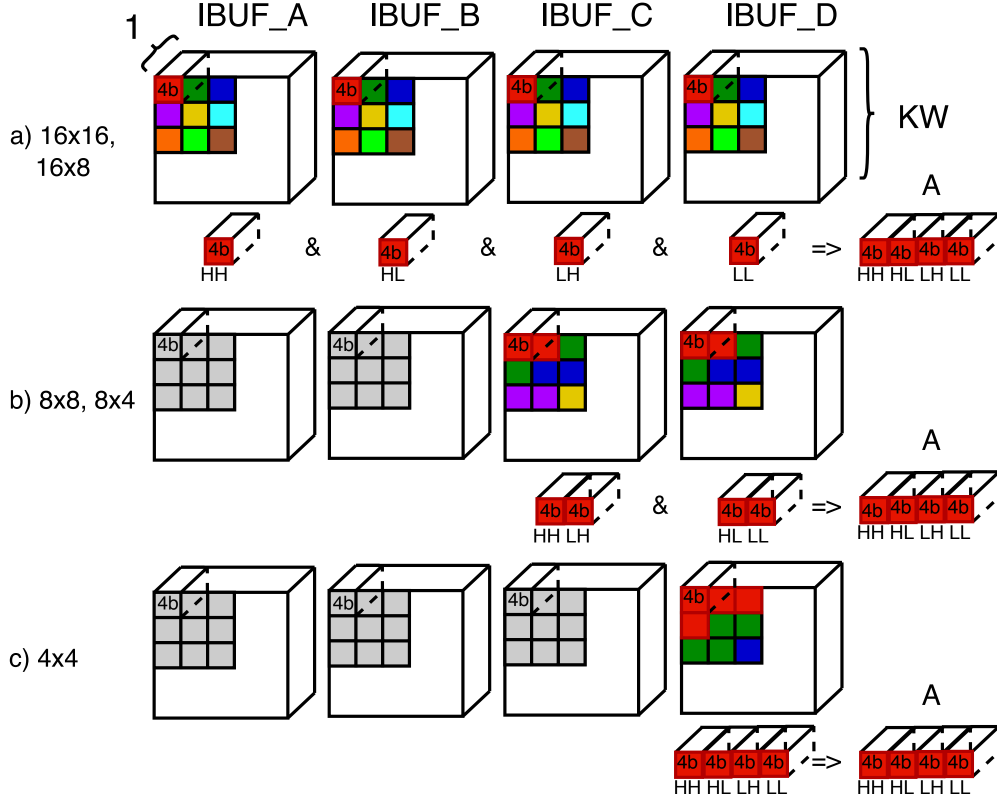


Figure 4.3: Memory addressing and concatenating logic acting on the four 4-bit memory banks of the internal input buffer (IBUF_A/B/C/D) of DW-Conv, already filled by an external DMA. Grey banks are unused in the corresponding configuration. Image derived from [40].

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 4.2, 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. 4.1, 27 for Lst. 4.2 and 38 for Lst. 4.3), because the unrolling factor matches their upper bound, thus replicating M times the ST-multiplier and the accumulation adder. As introduced in Sec. 4.2.2, 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 4.2 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. 4.1, lines 28–36 in Lst. 4.3).

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. 4.1 and 4.2). For FC, we got inspired from [25], 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. 4.3). This is also the reason why we have twice the input and weight buffers at the interface (lines 6–13 in Lst. 4.3).

The bitwidth of adders and accumulation registers is the result of the ablation study discussed in Sec. 4.3.2.

Quantization and ReLU Block

This block implements the UIQ formulas (A.2), (A.4), and (4.6) (with $z_W = 0$ and $z_b = 0$ [51, 52]) 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. 4.3.2. 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 (A.2), (A.4), and (4.6). 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. 4.1.1. 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. 4.1, Lst. 4.2, and Lst. 4.3, 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 [52, 89].

4.3 Accelerators Design Flow

To obtain our ST-based hardware accelerators, we use the design flow outlined in Fig. 4.4. It consists of the following three steps, which are analyzed in detail in Secs. 4.3.1–4.3.3:

A) **MP Quantization and Fine Tuning**. Quantizing a set of DNN models

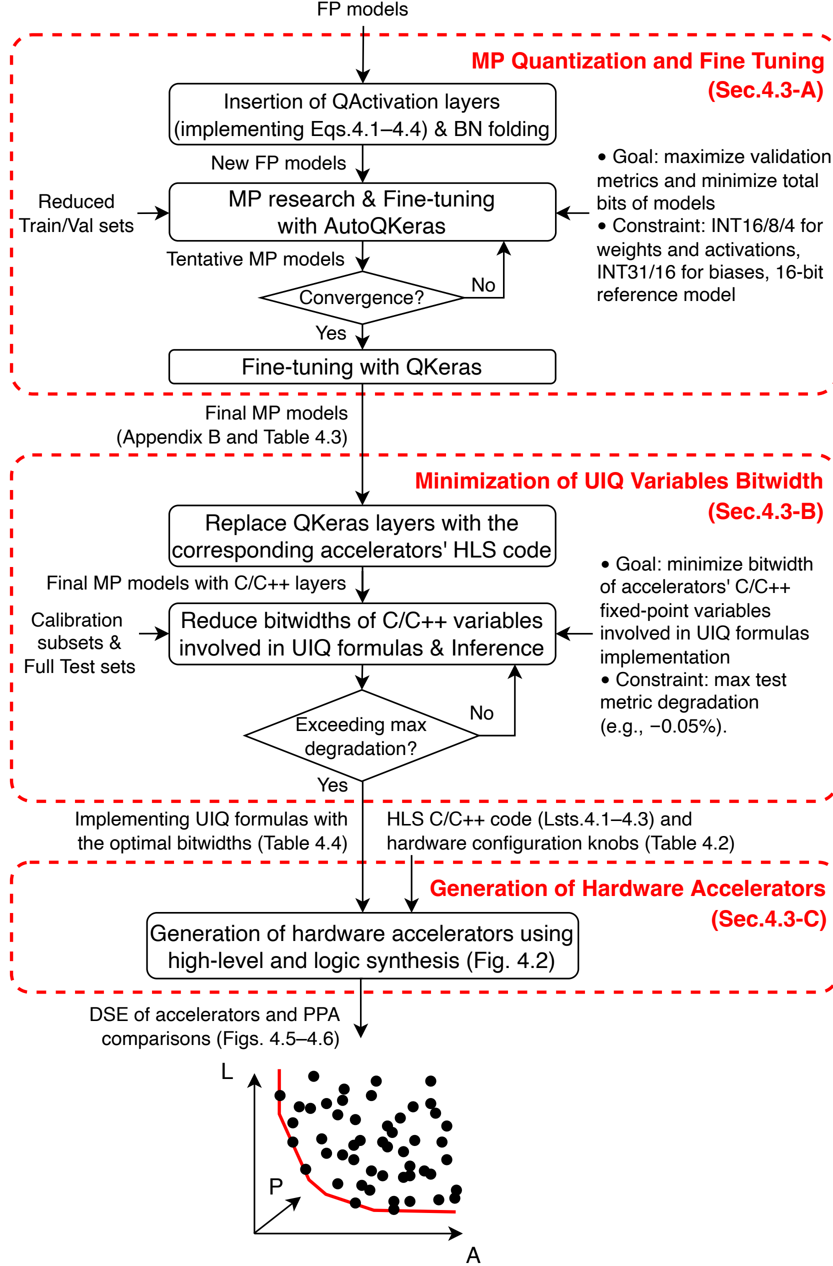


Figure 4.4: Accelerators design flow. Image taken from [28].

in MP is the first step of the proposed flow. For this work, we choose as case study the MLPerf Tiny benchmark [53] 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.

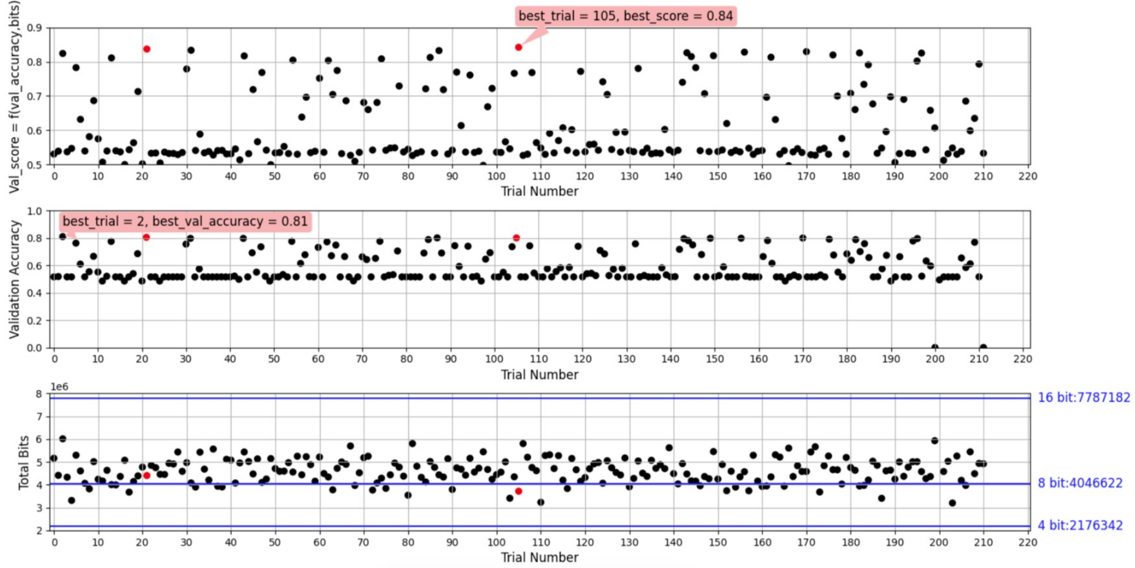
C) **Generation of Hardware Accelerators**. Using the optimal bitwidth precision determined in step B), we perform a DSE in both latency vs area and latency vs power for each accelerator. In the exploration we vary many hardware configuration knobs, as listed in Table 4.2, including HLS directives (e.g., pipelining and unrolling) and type of ST multiplier for the PSMAC array.

4.3.1 MP Quantization and Fine Tuning

We build the first step of our design flow on top of QKeras [14], 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. (4.1)–(4.4), resulting in a new version of QKeras for integer-arithmetic-only inference. This new version behaves similarly to TFLite [51], 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 [111]. As we show in Tables B.1–B.4 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 [14], 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 [51, 89]. 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

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



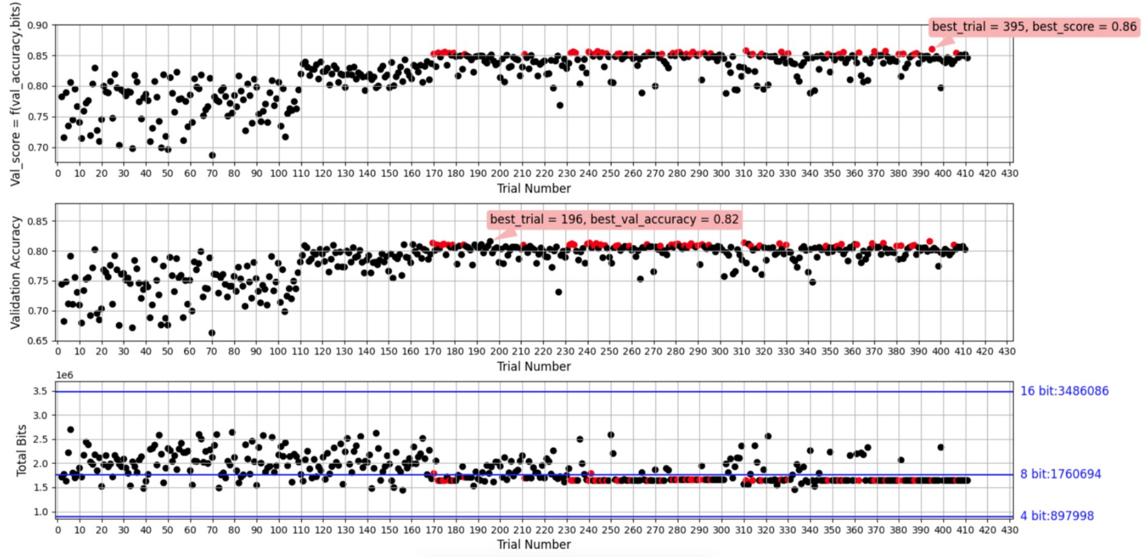
(a) MobileNetV1Tiny.

Figure 4.5: AutoQKeras search results. Image taken from [23].

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: $1/(1 + MSE_{loss}/10)$.

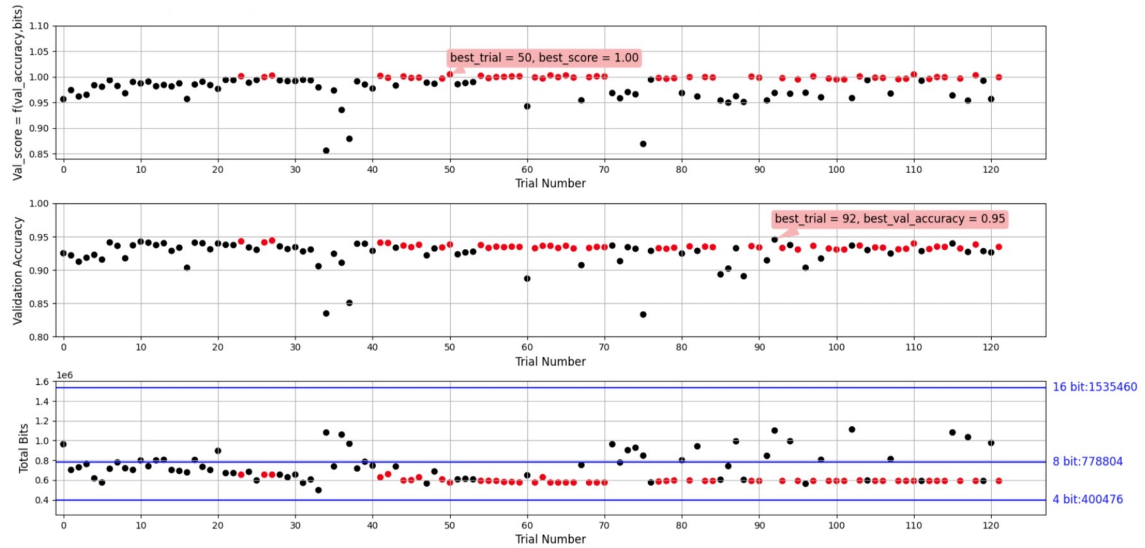
For each network, we use AutoQKeras to iteratively sample from the search space a different combination of feature map, weight, and bias bitwidths for each layer. Then, we let AutoQKeras fine-tune the resulting MP network for a few epochs starting from pre-trained FP weights, when available, to shorten 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 [112], 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.

Figs. 4.5a–4.5d report: in the first subplot, the AutoQKeras' score (*Val_score*); in the second subplot, the validation accuracy (or custom validation metric for FC-AutoEncoder) reached by the tentative MP models discovered by AutoQKeras at each search iteration (*Trial Number*); in the third subplot, the total number of bits (*Total Bits*) of the tentative MP models. Points within 1% of the maximum achieved score, along with their corresponding points in the second and third subplots, are highlighted in red. The total bits achieved by the 16-, 8-, and 4-bit flat



(b) ResNetV1Tiny.

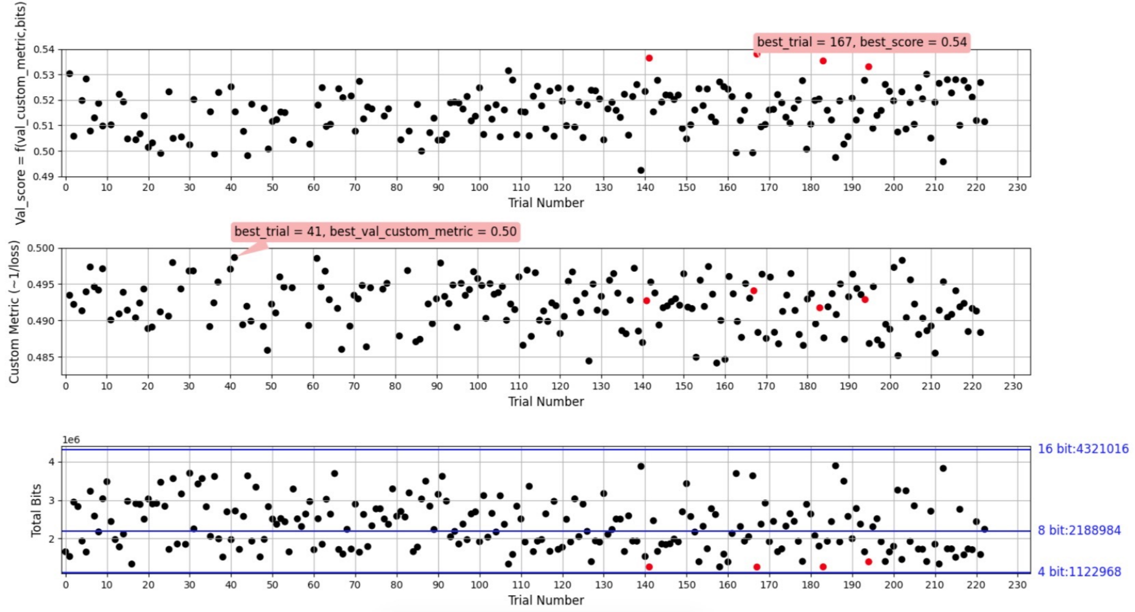
Figure 4.5: AutoQKeras search results. Image taken from [23].



(c) DS-CNN.

Figure 4.5: AutoQKeras search results. Image taken from [23].

reference models (i.e., models with all activations and weights quantized to INTX, where $X = 16, 8, 4$, and biases quantized to INT31) are marked by blue horizontal lines. We interrupt AutoQKeras' search when the score reaches convergence, i.e., stabilizes around a fixed value. This happens approximately after 200, 400, 100,



(d) FC-AutoEncoder.

Figure 4.5: AutoQKeras search results. Image taken from [23].

and 200 search iterations for MobileNetV1Tiny, ResNetV1Tiny, DS-CNN, and FC-AutoEncoder, respectively, as visible from the x-axes of Figs. 4.5a–4.5d. Finally, we select the four MLPerf Tiny MP models that give the best scores, as pointed out by the pink labels (*best_trial*) in the first subplot of Figs. 4.5a–4.5d, and we conclude by fine-tuning them with the default settings of the training scripts included in the MLPerf Tiny GitHub repository [112]. For the exploration, computational resources were provided by HPC@POLITO, a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino[4].

In Sec. 4.2.2, 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 into the weights of adjacent convolutional layers, or into the scaling factors and bias of the affine quantization formulas, using a technique known as BN folding [52]. 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

[4]HPC@PoliTo: https://hpc.polito.it. Accessed on: Jan 19, 2024.

Table 4.3: Performance of MLPerf Tiny models (column 1) on the corresponding Perf test sets (Sec. 4.1.2), using AUC for FC-AutoEncoder and accuracy for the other three models, for their FP (column 3), MP (column 4) and MP with optimal C/C++ bitwidths (column 5) versions.

MLPerf Tiny Model	**MLPerf Tiny Quality Target [53]**	**FP Model**	**MP Model (vs FP)**	**MP Model with optimal C/C++ bitwidth (vs FP)**	**Tot. Bits Reduction (vs 16-bit flat)**
	[Top-1 Acc./AUC]	**[Top-1 Acc./AUC]**	**[Top-1 Acc./AUC]**	**[Top-1 Acc./AUC]**	**[%]**
MobileNetV1Tiny	80.00	86.00	85.00 (−1.00)	85.00 (−1.00)	−52.25
ResNetV1Tiny	85.00	88.50	87.50 (−1.00)	87.50 (−1.00)	−52.91
DS-CNN	90.00	92.10	90.00 (−2.10)	90.40 (−1.70)	−62.56
FC-AutoEncoder	85.00	88.71	87.51 (−1.20)	87.13 (−1.58)	−70.85

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 B.2, 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 4.3, 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 [53] (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.

The test datasets of ResNetV1Tiny, FC-AutoEncoder, and DS-CNN are the Perf test sets available on the MLPerf Tiny GitHub repository [112]. Regarding MobileNetV1Tiny, its Perf dataset is not publicly available: only a list of 1000 indexes is provided without an extraction script. We managed to retrieve the 1000 corresponding images with a custom script [23], but we raised a GitHub issue[5] warning that some of these indexes were pointing to images used for training, potentially leading to an overestimation of the resulting test accuracy. However,

[5]Open issue about the lack of the Perf test set for MobileNetV1Tiny on the MLPerf Tiny GitHub repository: https://github.com/mlcommons/tiny/issues/135. Accessed Mar 9, 2024.

Table 4.4: Minimum bitwidths (row 2) of the C/C++ variables (row 1) resulting from the ablation study. The notation follows the format <integer bits>.<fractional bits>.

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

by running inference on the FP MobileNetV1Tiny model with these 1000 images, we obtained accuracy results similar to those reported in the MLPerf Tiny paper [53]. Therefore, for our experiments we continued to use these 1000 images as the Perf test set for MobileNetV1Tiny.

In Table 4.3 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%–2% 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 [53]. Moreover, the total bits reduction (column 7) is greater than 50% for all models, confirming the effectiveness of the MP optimization performed by AutoQKeras.

4.3.2 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 (4.6) 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 4.4. 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 (4.6) in (4.9)–(4.12):

$$v1_{q,k} = \left[\sum_{c=1}^{C} X_{q,c} W_{q,c,k} - z_X \sum_{c=1}^{C} W_{q,c,k} \right] \quad (4.9)$$

$$v2_{q,k} = s_X s_W \cdot v1_{q,k} \quad (4.10)$$

$$v3_{q,k} = s_b b_{q,k} + v2_{q,k} \quad (4.11)$$

$$Y_{q,k} = clip(round(z_Y + s_Y^{-1}\ v3_{q,k}), \alpha_q, \beta_q) \quad (4.12)$$

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 (4.6) in Sec. 4.1.1. 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 *QConv-2DBatchnorm*, *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 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 4.3. In the future, we plan to find these optimal bitwidths automatically through hardware-aware training [113].

The so-obtained optimal bitwidths for the MLPerf Tiny benchmark are in Table 4.4, 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 4.3. 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 [52]. We use these optimal values to synthesize the accelerators in the third step of the accelerators design flow (Sec. 4.3.3).

4.3.3 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. 4.2. 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. 4.1-4.3;

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 4.2, 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 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/loading cells, compilation strategy.

We use the HLS directives to perform several optimizations. As mentioned in Secs. 4.2.2 and 4.2.2, we fully unroll the innermost loops in Lsts. 4.1–4.3 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 outermost loops (more details in Sec. 4.4.1). The clock frequency constraint is common to both high-level and logic synthesis. In the HLS tool, we set a clock uncertainty constraint 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 [114]. 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. In the logic synthesis process, we use conventional methods to set margins on the clock period.

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 4.2. For this we use the directive *map_to_operator* (e.g., line 3 of Lst. 4.1), followed by the name of the multiplier's RTL top-level entity that we want to use $X = \{$[25], [26], [41], [49], [48], 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 4.2. The second option, explored in [28] and not in [24], 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 4.2. 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 [114], 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.

4.4 Experimental Results

4.4.1 DSE of ST-based Accelerators

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. 4.3.3 and vary hardware configuration knobs, implementation constraints, accelerators' internal buffers, and maximum tile sizes, according to the values in Table 4.2. We also vary the target clock frequency (last row of Table 4.2) in ten steps from 100 to 1000 MHz, which we verified being the maximum clock frequency reachable by all

Table 4.5: *For* loops of the high-level C/C++ descriptions Lsts. 4.1–4.2 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	$\geq$500
DW-Conv	oh, ow, k	16	not HLS ST Inline	≥ 600
DW-Conv	oh, ow, k	32	all	≥ 500

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. 4.6–4.7), as explained in Sec. 4.3.3.

Despite the suboptimal performance of certain ST multipliers, as indicated by our findings in Sec. 3.2.1, 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. 4.2, 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 4.5 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. 3.2.1. 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. 4.2.2); the second is the first depth-wise layer of MobileNetV1 [95]; 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. 4.6–4.7, which can be therefore extended to any DNN layer. Furthermore, we plot the results normalized so as to make them layer-independent.

Figs. 4.6-4.7 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)

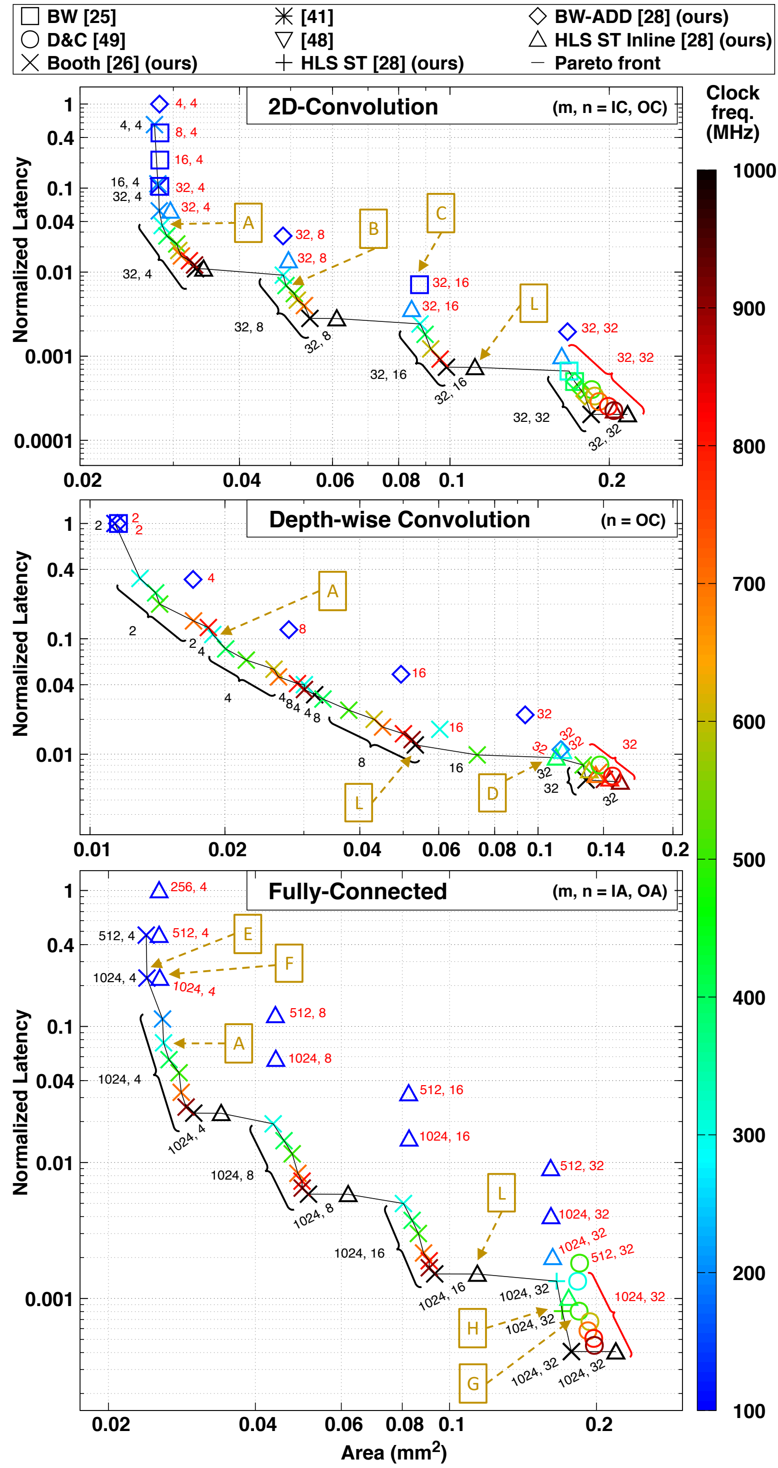


Figure 4.6: Results of DSE in Latency vs Area (LA). Points with black and red labels are Pareto points in LA and Latency vs Power, respectively. Image taken from [28].

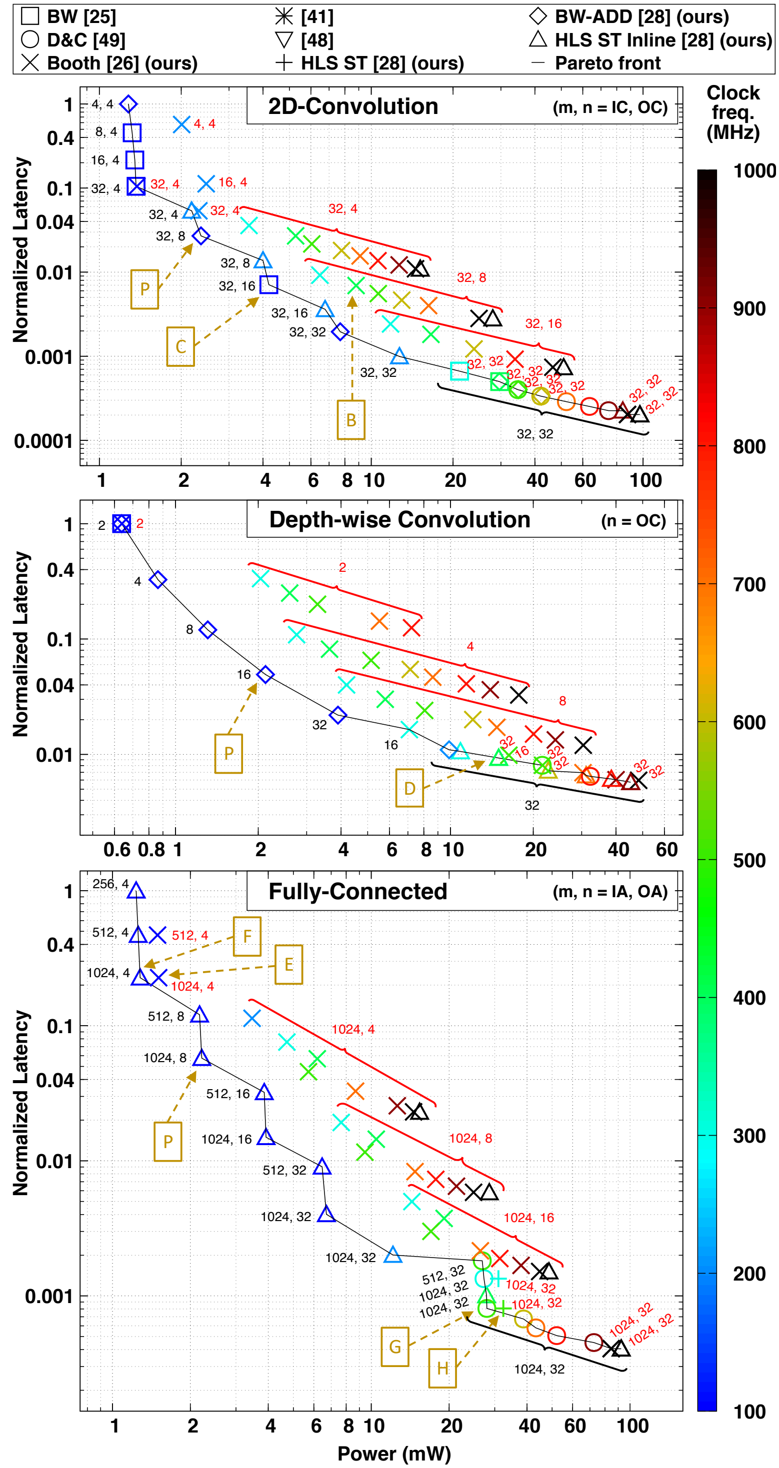


Figure 4.7: Results of DSE in Latency vs Power (LP). Points with black and red labels are Pareto points in Latency vs Area and LP, respectively. Image taken from [28].

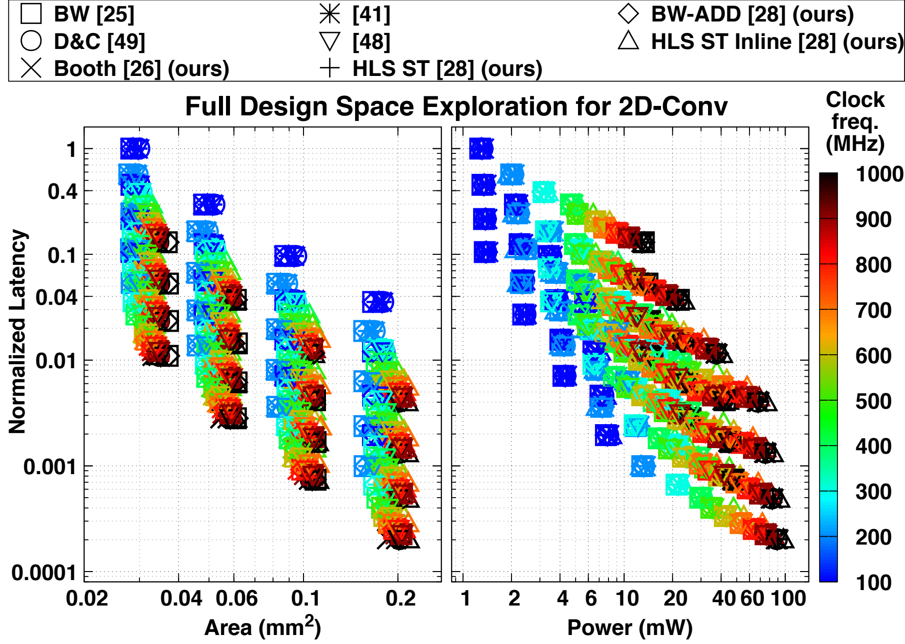


Figure 4.8: Example of a complete DSE for 2D-Conv. Image taken from [28].

and Latency vs Power (LP), respectively. An illustrative example of a complete DSE for the 2D-Conv accelerator is instead reported in Fig. 4.8, which shows the extensive range of design variations explored. The points in Fig. 4.6 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. 4.7. 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 4.1. In a dual manner, Fig. 4.7 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 0.007, and using IP [26] 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 [25]. 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 [26] 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 [49], 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. 4.6–4.7 we observe that:

- The DSE and the PPA results are especially sensitive to two main variables that control the PSMAC array 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 4.1). 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. 4.1 and Lst. 4.3: 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. 3.10 and the types of ST multipliers used in the dominating accelerators in Figs. 4.6–4.7.

  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 [26], the dominant point in the Clock Period vs Area subplot of Fig. 3.10. A few Pareto points, however, are based on BW-ADD and HLS ST, which are indeed sub-optimal in the top graph of Fig. 3.10 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. 4.6 no Pareto-optimal accelerators are based on the ST multipliers that are largely sub-optimal in the top graph of Fig. 3.10 ([41], [49], [48]).

Similarly, in the LP space, since in the Clock Period vs Power graph at the bottom of Fig. 3.10 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.

4.4.2 Performance on MP-quantized MLPerf Tiny Models

In this section, 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. 4.3.1. 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. 4.6–4.7 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. 4.2.2). 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. 4.1–4.3). We decide to let the processor compute the pooling layers, which, in the case of MLPerf Tiny networks, exclusively consist of average pooling. However, we could have also used DW-Conv with all weights set at $1/K^2$. In general, other pooling methods, such as those developed by [115], 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 4.2), 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. These ideal accelerators are hypothetically capable of accelerating their entire execution latency by a factor N, including the latency required by the filling and draining phases of pipelined loops. Since these phases are not scalable in precision, ideal accelerators are actually unattainable in reality. On the contrary, in our ST-based accelerators only some *for* loops are accelerated by a factor N, as seen in Lsts. 4.1–4.3: 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, and the filling and draining phases of the pipelined ones,

Table 4.6: 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. 4.6–4.7)	**MLPerf Tiny Model**	**Theoretical Latency Speedup**	**Actual Latency Speedup**	**Actual Energy Reduction [%]**
Low-area, $< 0.03\,\text{mm}^2$, 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, $< 3\,\text{mW}$, 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, $< 0.12\,\text{mm}^2$ 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

represent an overhead for the actual execution of the ST-based accelerator when compared with the ideal one (as we partially discussed in Sec. 4.4.1). Thus, the theoretical speedup, compared with the actual one, allows us to seize the impact of these computational overheads. 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.

In Table 4.6 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 ($< 0.03\,\mathrm{mm}^2$), low-power ($< 3\,\mathrm{mW}$), and low-latency ($< 0.12\,\mathrm{mm}^2$). We mark the selected accelerators with letters A, P, L for the three scenarios, respectively, in both Table 4.6 and Figs. 4.6–4.7. 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 4.6 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. 4.2.1. 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. 4.6) 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 have already planned to tackle these inefficiencies by developing a new PS DW-Conv accelerator based on an SA multiplier (Sec. 1.1). 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 [54], cache included) and the SRAM-based global 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.

Chapter 5

Precision-Scalable Multipliers: Sum-Together/Apart Reconfigurable (STAR) Multipliers

Some of the work described in this chapter is taken from [27], which has been published in the proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), March 25–27, 2024, Valencia, Spain.

As previously discussed in Sec. 1.2, STAR represents a new family of PS multipliers capable of operating in either SA mode or ST mode. This versatility, enabled by the dynamic allocation of hardware resources within the multiplier, opens up various potential applications, including integration within MAC units or hardware accelerators tailored for DL workloads. For instance, in Fig. 5.1, we show how a STAR-based MAC unit can be used in a single hardware accelerator to enable the support for both 2D and DW convolutions [24, 28]. For 2D-Conv (Fig. 5.1(a)), the STAR multiplier is configured in ST mode to multiply and accumulate pairs of low-precision feature maps (light blue) and weights (orange) by reading them channel-wise [39]. An external accumulator further sums up the partial results of the STAR unit until the entire input tensors are scanned and the specific element of the output tensor (green) is computed. For DW-Conv (Fig. 5.1b), instead, the STAR multiplier is configured in SA mode to perform multiple low-precision products in parallel (without internal accumulation) between features and weights belonging to different channels, according to the DW-Conv algorithm [40]. In this case the external accumulator is reconfigured, depending on the precision, to keep the accumulated results of the multiplications in $N = 2$ or 4 separate elements. STAR could also be used in ST mode for Fully-Connected layers, as described in [24, 25, 28].

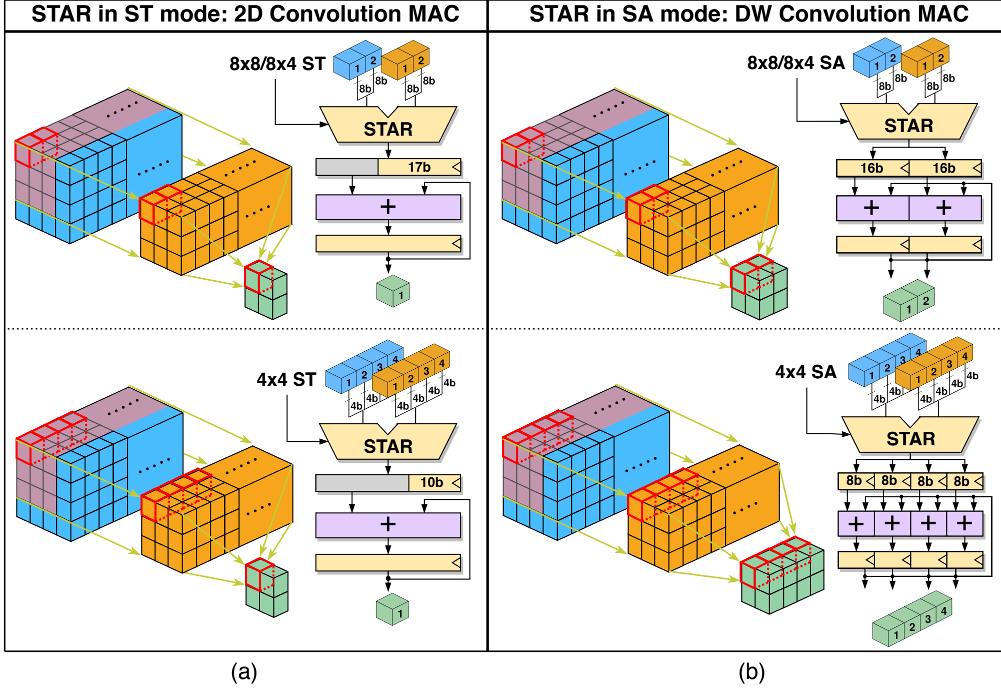


Figure 5.1: STAR-enabled reconfigurable MAC for 2D/DW Convolution. Image from [27].

5.1 STAR Architectures

In this section we propose four STAR multiplier architectures, shown in Figs. 5.2–5.4, which support the configuration modes reported in Table 5.1. A and B are the 16-bit input operands, O is the 32-bit output, and *CONFIG* is the control signal used to select the operating mode (not shown in Figs. 5.2–5.4 for better readability). In addition to symmetric configurations (i.e., 16×16, 8×8, and 4×4), STAR multipliers also support asymmetric ones (i.e. 16×8 and 8×4) which are implemented by properly sign-extending operand B, as shown in Figs. 5.2–5.4 in the dashed-line blocks. Should the support for asymmetric configurations be unnecessary, its removal is immediate.

Before delving into the details of STAR architectures, it is worth recalling the taxonomy of [29], which further divides the SA and ST multipliers into SWP and D&C classes (Sec. 2.1). The first class comprises multipliers that can work in full- or reduced-precision mode by disabling specific arithmetic logic cells, like BW multipliers [25, 35] and Booth-based ones [26]. Instead, the elements of the second class are composed by many low-precision multipliers (e.g. 4-bit) that can be combined by means of shift-add logic to form higher precision multipliers (e.g. 16-bit), like [49] and [37].

Table 5.1: Operating modes of STAR.

CONFIG	STAR output
16×16	$O_{[31:0]} = A_{[15:0]} \times B_{[15:0]}$
16×8	$O_{[31:0]} = A_{[15:0]} \times B_{[7:0]}$
4×4 ST	$O_{[21:12]} = A_{[3:0]} \times B_{[15:12]} + A_{[7:4]} \times B_{[11:8]} + A_{[11:8]} \times B_{[7:4]} + A_{[15:12]} \times B_{[3:0]}$
8×8 ST	$O_{[24:8]} = A_{[7:0]} \times B_{[15:8]} + A_{[15:8]} \times B_{[7:0]}$
8×4 ST	$O_{[24:8]} = A_{[7:0]} \times B_{[11:8]} + A_{[15:8]} \times B_{[3:0]}$
4×4 SA	$O_{[31:24]} = A_{[15:12]} \times B_{[15:12]}$ $O_{[23:16]} = A_{[11:8]} \times B_{[11:8]}$ $O_{[15:8]} = A_{[7:4]} \times B_{[7:4]}$ $O_{[7:0]} = A_{[3:0]} \times B_{[3:0]}$
8×8 SA	$O_{[31:16]} = A_{[15:8]} \times B_{[15:8]}$ $O_{[15:0]} = A_{[7:0]} \times B_{[7:0]}$
8×4 SA	$O_{[31:16]} = A_{[15:8]} \times B_{[11:8]}$ $O_{[15:0]} = A_{[7:0]} \times B_{[3:0]}$

The STAR architecture in Fig. 5.2a is based on a **naive approach**, which consists of multiplexing the outputs of two multiplier components, one SA and one ST, using *CONFIG* as control signal. We create two RTL implementations using this architecture: one called *ST+SA SWP (BW)*, where the two components are SWP multipliers inspired from the ST and SA BW schemes introduced in [25]; the other one called *ST+SA D&C*, where the two components are two D&C multipliers inspired from [49] to avoid the explosion of the input memory bandwidth [29]. In particular, we re-implement [49] with 4-bit multipliers as basic building blocks.

Fig. 5.2b illustrates a **SWP STAR** architecture. We name it *STAR SWP (BW)* because we employ a BW multiplier approach. The block *shift* & *ext* is used to align the output of ST operations to the least significant position [25] and to extend its sign to 32 bits.

The STAR architecture in Fig. 5.3 is called ***3-way*** and is inspired by the dot-product unit of the RISC-V core of [41]. It consists of three datapaths activated by signal *CONFIG* in a mutually exclusive way: one 16-bit multiplier for configurations 16×16 and 16×8, two 8-bit multipliers and one adder for configurations 8×8 and 8×4, and four 4-bit multipliers and three adders for configuration 4×4. The blocks named *ext*, located after the last adders of the 8-bit and 4-bit datapaths, sign-extend the low-precision outputs to 32 bits in case of ST operations, while the "*&*" blocks concatenate them in case of SA operations. We create only one instance of this architecture, letting the logic synthesizer choose the best implementation of the various multipliers.

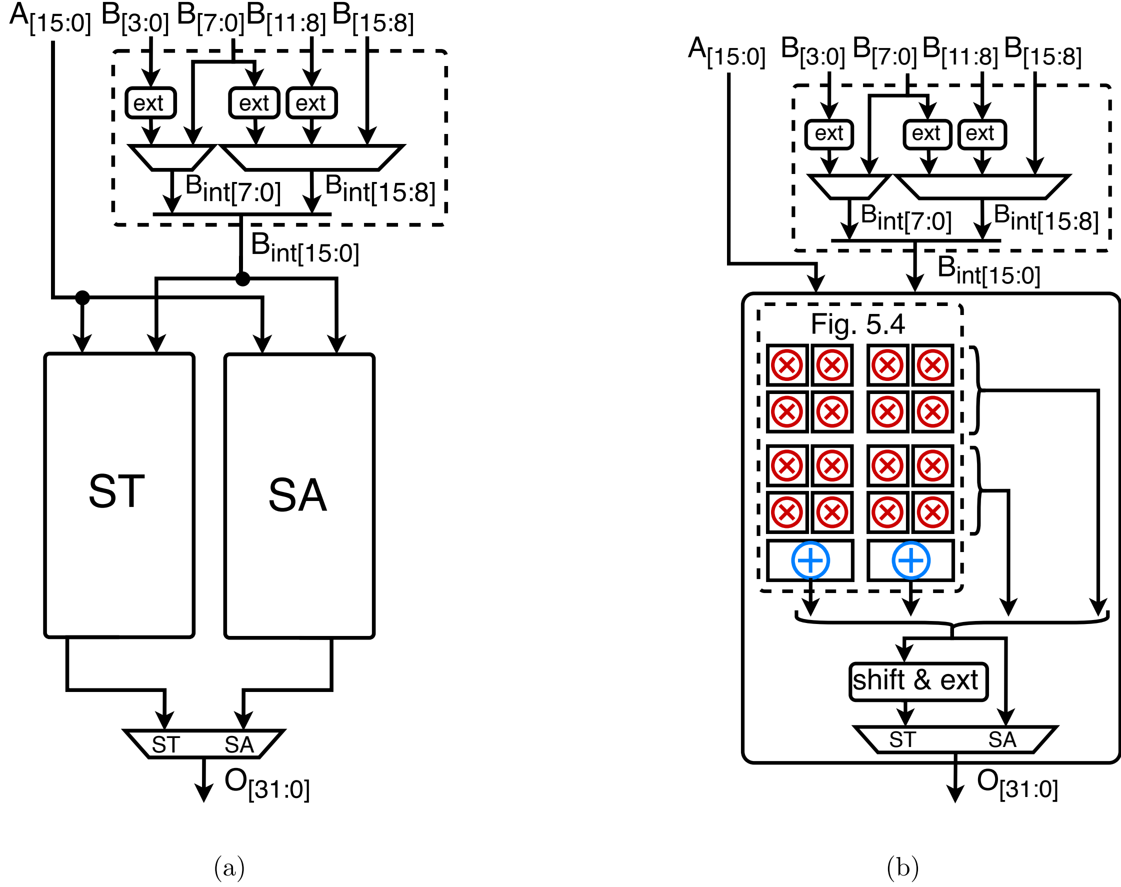


Figure 5.2: The proposed STAR (a) Naive and (b) SWP (BW) architectures. Images taken from [27].

The fourth STAR architectures in Fig. 5.4 is named ***STAR D&C*** because it is derived from the D&C architecture of [49], which we discussed in Sec.3.1.1 and Fig. 3.5.

In Sec. 5.2 we provide a detailed explanation of the design of STAR SWP (BW), while for the other STAR architectures the presented schematics are already self-explanatory.

5.2 STAR Sub-word Parallel Baugh-Wooley Design

To support all the previously defined operations, STAR SWP (BW) operates in five different modes, which are illustrated in Figs. 5.5a–e. The upper square in each subfigure represents the BW partial products matrix (PPM), while the lower

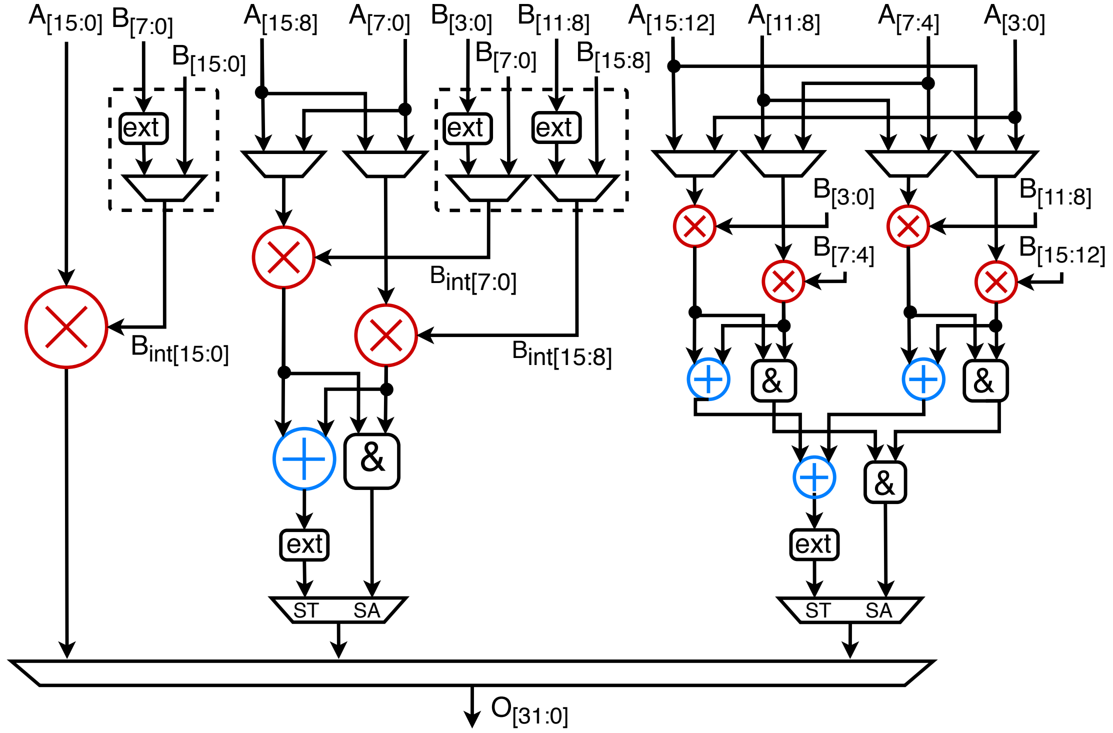


Figure 5.3: The proposed STAR 3-way architecture. Image taken from [27].

part shows the 32-bit output. According to the selected operating mode, some PPs become active (yellow squares) and contribute to generate the valid output bits (yellow), whereas other PPs are inactive (grey squares) and do not contribute to the final 32-bit result.

The PPM of STAR SWP (BW) is detailed in Fig.5.6 (top). Like any standard BW, each block computes the PP between a different pair of bits of the two 16-bit input operands using an AND gate. Then, through a Full Adder (FA), it compresses the output of the AND gate together with the input sum S_i and carry C_i bits coming from the previous row of PPs, and it provides the output sum S_o and carry C_o bits to the blocks of the next row of the PPM. The sixteen S_o bits exiting from the right-most column of the PPM represent the least significant part of the multiplier's output. The most significant part is instead obtained by compressing, through a 16-bit RCA, the S_o and C_o output bits exiting from the last row of the PPM. To deal with signed numbers, a standard BW inverts the PPs of the left-most column and of the last row [50]. This is accomplished by substituting the AND gate with a NAND gate, in each of these blocks. Moreover, to perform the BW algorithm as in [50], the addition of logic 1s is required: this is done via S_i inputs of the first row and the left-most column.

To derive STAR SWP (BW) from a standard BW multiplier, we add few logic

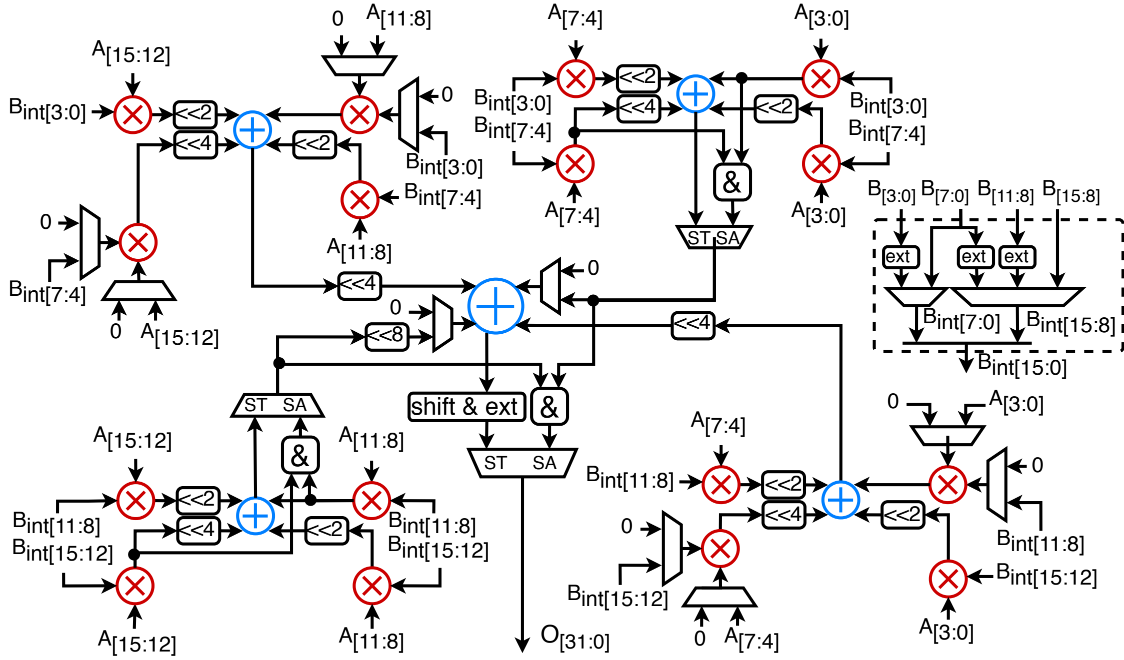


Figure 5.4: The proposed STAR D&C architecture. Image taken from [27].

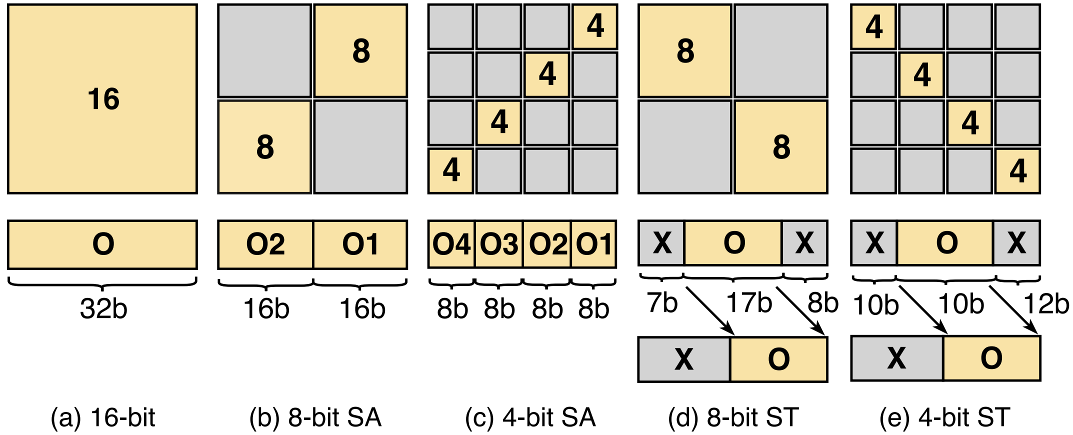


Figure 5.5: STAR SWP (BW) operating modes. Partial product matrix (top square) and output (bottom rectangle). Right shift for ST modes only (d)-(e). Images taken from [27].

gates to each block of the PPM, resulting in the five different versions shown in Fig. 5.6 (bottom). The reconfiguration of these blocks is achieved through the binary control signals P and I, which are generated from a decoding logic acting on *CONFIG*, the signal controlling the operating mode as in Table 5.1. Each block of the PPM receives specific binary values for P and I, according to the letters

reported in the block itself (see Fig. 5.6, top): when a block contains a single letter, it receives only logic values for P (in fact, white and grey blocks do not have input I); when a block contains two letters, it receives logic values for I and P, respectively (e.g., c/d means $I = \mathrm{c}$, $P = \mathrm{d}$).

As shown in Fig. 5.5, depending on the selected operating mode, the output of each block requires potentially to be turned off in order not to contribute to the final result. For this reason, we add an AND gate in all blocks to control the propagation of PPs towards the internal FAs, gating it ($P = 0$) or letting it pass ($P = 1$). The white and grey blocks shown in Fig. 5.6 (bottom), are almost identical to those of a standard BW, with the exception of this AND gate.

When dealing with signed operations at reduced precision, we need to guarantee the PP inversion in the left-most and bottom blocks of all the 8-bit and 4-bit sub-precision multipliers inside the PPM array, as well. These blocks are the red ones in Fig. 5.6 (top), with details of the internal logic in Fig. 5.6 (bottom): we add an XOR gate to invert the internal PP ($I = 1$), or leave it unchanged ($I = 0$).

In red blocks we can also force the input of their internal FAs to be logic 1, regardless the actual value of the PP, using a proper control signal configuration (i.e. $P = 0$ and $I = 1$). This feature can be exploited to add the 1s required by the BW algorithm in place of the S_i inputs of the PPM blocks in the top row and left-most column, thus saving resources and reducing the propagation path of these inputs [50]. However, the red blocks do not cover all the positions that would be required by the insertion of the logic 1s. For this reason we have blue blocks and green blocks (see Fig. 5.6, bottom), in which an OR gate can force the input of the FA to be a logic 1 (when $I = 1$).

In case of SA operations, the carry chains connecting the MSB bit of a sub-word result to the LSB of the next one need to be interrupted. For this reason, we add a few AND gates in selected positions to stop the propagation with control signal $M = 0$, as shown in Fig. 5.6g. These AND gates also affect the carry chain of the 16-bit RCA, which is halved in two independent 8-bit RCAs when needed. The "X" symbols in Fig. 5.6 (top) mark the positions of these AND gates. Notice the letter associated to each diagonal of "X" symbols, which corresponds to the signal associated to the M inputs of the AND gates of that diagonal. Like P and I, M is derived from *CONFIG* by the decoding logic and the letters m and d in Fig. 5.6 (top) correspond to the logic values associated to the green and violet "X" symbols, respectively.

In the following, we explain in detail how P, I and M enable the reconfiguration of STAR SWP (BW).

16-bit multiplications

For 16×16 and 16×8 multiplications, the PPM is configured to work as in a standard full-precision BW multiplier [50]. In particular, all the blocks are configured with $P=1$ ($\{a,b,c,d\}=1$), and all but the red blocks in the left-most column and the last row are configured with $I=0$ ($\{e,f,g,h,i,j\}=0$). These red blocks instead have to invert the PPs, having $I=1$ ($\{c,k\}=1$). Finally, the addition of logic 1s, required by signed operations [50], is introduced through the carry-in signal of the right 8-bit RCA ($k=1$) and through the most significant FA of the left 8-bit RCA ($a=1$), as clear in Fig. 5.6 (top).

Sum-Apart multiplications

In SA mode, the STAR SWP (BW) multiplier is configured to operate as two or four sub-word parallel BW multipliers.

In the 4×4 configuration, the yellow squares of the PPM in Fig.5.5c correspond to the four groups of $4\times4=16$ blocks (64 blocks in total) in the right-to-left diagonal of the PPM array in Fig. 5.6 (top). These four diagonal squares can be seen as four independent 4-bit BW multipliers. The 16 blocks inside each of these 4-bit multipliers are configured with $P=1$ ($a=1$) and $I=0$ ($j=0$), except for the red blocks in the left-most column and in the last row, having $I=1$ ($\{g,h\}=1$). The other blocks in the grey squares in Fig. 5.5c are gated, thus have $P=0$ ($\{b,c,d\}=0$), and become inactive. To perform the addition of 1s required by the BW algorithm [50], we use some of these inactive blocks, since they do not need to propagate the internal PPs when in SA mode. In particular, we use the red, blue and green blocks with magenta borders in Fig. 5.6 (top), which can add a 1 in the FAs chain with $I=1$ ($\{g,h\}=1$, $i=0$). All the other inactive blocks have $I=0$ ($\{c,e,f,i,j,k\}=0$) thus they do not add any 1. In positions where no inactive blocks are available, we exploit the S_i inputs of the blocks in the left-most column of the PPM and the input of the most significant FA of the left-most 8-bit RCA ($\{a,g\}=1$, $i=0$). Finally, to keep the four 8-bit multiplication results separated in the final 32-bit result, we have to interrupt the propagation of the carry-out bits in all the positions marked by the green and violet “X” symbols in Fig.5.6 (top) ($\{d,m\}=0$), as explained before.

In the 8×8 and 8×4 cases, the two yellow squares of the PPM in Fig.5.5b correspond to the two groups of $8\times8=64$ blocks (128 blocks in total) in the right-to-left diagonal of the array in Fig.5.6 (top). Also in this case, these two squares can be seen as independent 8-bit BW multipliers. The 64 blocks inside each 8-bit BW are configured with $P=1$ ($\{a,b\}=1$) and $I=0$ ($\{g,j\}=0$), while the red blocks in the left-most column and the last row receive $I=1$ ($\{h,i\}=1$). Inactive blocks require $P=0$ ($\{c,d\}=0$). To add logic 1s, we use again the blue and green blocks present in inactive positions, with magenta border in Fig. 5.6 (top), by setting $I=1$ ($\{i,h\}=1$, $g=0$), while other blocks have $I=0$ ($\{c,e,f,i,j,k\}=0$). When

no inactive blocks are available, we use the S_i inputs of the blocks in the left-most column and of the most significant FA of the left 8-bit RCA ($\{a,i\}=1,\ g=0$). Finally, to maintain the two independent 16-bit multiplication results, we gate the carry-out bits in all the positions marked by the violet “X” symbols in Fig.5.6 (top) ($d=0,\ m=1$).

Sum-Together multiplications

In ST mode, STAR SWP (BW) always operates in sub-word parallel mode, but the active yellow squares of the PPM are mirrored with respect to SA mode, as shown in Figs.5.5d–e. This implies that the output sum bits S_o of one yellow square, propagating diagonally from left to right, can be used as inputs for the next yellow square, hence allowing the addition of the low-precision multiplications and ultimately achieving a dot-product operation.

Like in SA mode, blocks belonging to the yellow squares can be seen as independent BW multipliers. These blocks are configured with $P=1$ and $I=0$, except for the red blocks in the left-most column and in the last row, having $I=1$ (precise signal assignment is now trivial and left to the reader). The blocks belonging to the grey squares are gated, thus have $P=0$ to make them inactive. As in SA mode, we use some of these inactive blocks to accomplish the 1s addition required for signed operations. In this case, we use blue and green blocks with orange borders, as visible in Fig. 5.6 (top), which propagate, given $I=1$, a logic 1 to the internal FAs. The remaining inactive blocks receive $I=0$.

As explained in Sec. 5.1, ST operations need to align the final result to the LSB position [29]. Indeed, the *shift & ext* block in Fig. 5.2b is used to right-shift the result by 8 bits for 8×8 and 8×4 operations (Figs. 5.5d), and by 12 bits for 4×4 operation (Figs. 5.5e).

5.3 Experimental Results

5.3.1 Power, Performance and Area Comparison of STAR Multipliers

To rank the STAR solutions and establish the best PPA trade-off, we synthesize their RTL description after adding I/O registers using Synopsys DC and targeting a 28-nm CMOS technology. Fig. 5.7 reports the results of area and power vs clock period (displayed at the top and bottom, respectively), obtained by varying the target clock frequency from 0.4 to 2 GHz. The clock period also takes the reconfiguration time into account, allowing to change configuration at every cycle. Pareto-optimal points represent the solutions with lowest area or power for a given target clock period. The reported power is an average of eight values obtained when the multipliers are configured in 16-bit mode (16×16 and 16×8), ST mode

(8×8, 8×4, and 4×4 ST) and SA mode (8×8, 8×4, and 4×4 SA). For simplicity, the power is determined by applying random input bits evenly distributed between zero and one. While this may not be representative of realistic ML workloads, it still allows for a correct relative comparison. In the future, we plan to use distributions derived from quantized DNNs to provide more accurate estimates of absolute power consumption.

In the area vs clock period graph of Fig. 5.7, we observe that STAR SWP is Pareto-optimal in the low frequency range up to 0.8 GHz (1.25 ns) with runners-up STAR D&C and STAR 3-way. This is because in this range the BW implementation manages to effectively share the logic gates among all the operating modes in the best possible way, while the other solutions have redundant gates that result in larger area. The lower area, and so also lower capacitance, results in lower power for a given clock period, making STAR SWP Pareto-optimal also in power vs clock period up to 0.9 GHz (1.11 ns), as clear in the power vs clock period graph of Fig. 5.7.

In the range 0.9–1.1 GHz, both STAR D&C and STAR 3-way become Pareto-optimal in terms of area vs clock period, with STAR SWP third-best. This is because the BW implementation has longer critical paths, which is well known for BW multipliers in general [50] and results in larger gates to satisfy tighter frequency constraints.

In the same middle range, more precisely between 1.0 and 1.3 GHz, the STAR D&C solution manages to obtain the best trade-off in power vs clock period.

In the upper frequency range, STAR 3-way emerges without any contenders and achieves the best area and power. This is because the synthesizer freely selects the best implementation for the three types of multipliers, recovering area in the smallest 4×4 ones with shorter paths and using effectively gate sizing for the larger 16×16 one.

As expected, naively combining ST and SA solutions (ST+SA) is not effective and results in Pareto-dominated solutions both in area and power vs clock period. For a fair comparison, we do not analyze ST-only or SA-only designs because they do not support all the STAR configurations.

To summarize, the results of this exploration allow designers to choose the best implementation according to the design target. For low-power and low-area, a STAR SWP solution is the most appropriate implementation. At the other end of the spectrum, for high-performance designs, the best choice is STAR 3-way. In the middle, also STAR D&C can become competitive with STAR 3-way, especially in terms of power.

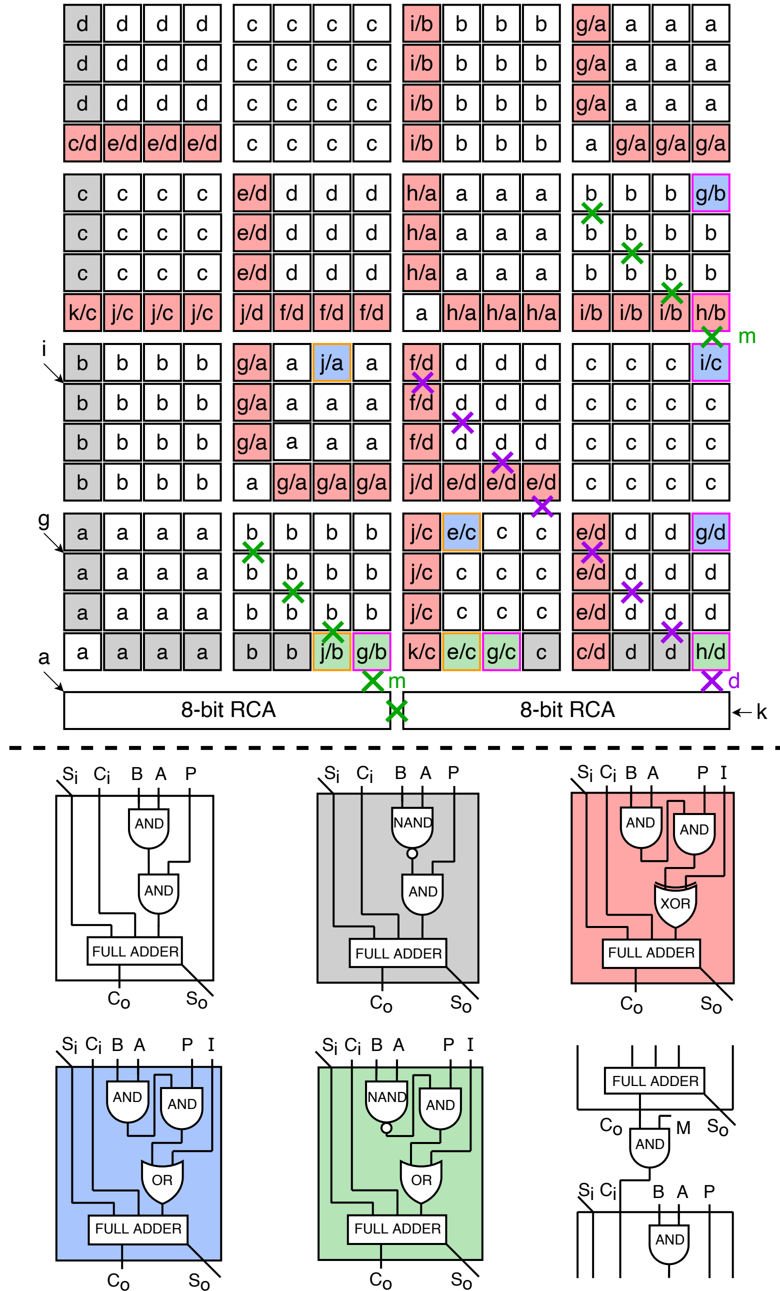


Figure 5.6: STAR SWP (BW): PPM with two 8-bit RCAs (top), PPM blocks (bottom), carry propagation "stopper" (bottom right). Each block receives specific binary values (marked with letters from a to i in the block itself) in P and I: a single letter represents logic values for P, while two letters for I and P, respectively (e.g., c/d means $I = \text{c}$, $P = \text{d}$). White and grey blocks, similar to a standard BW, have an extra AND gate to control the propagation of PPs. Red blocks guarantee the PP inversion and, together with blue and green ones, the generation of logic 1. The "stoppers" (marked with "X") halt the carry propagation of the low-precision results in SA mode, based on the binary values m and d received in M. All binary values depend on *CONFIG*. Image from [27].

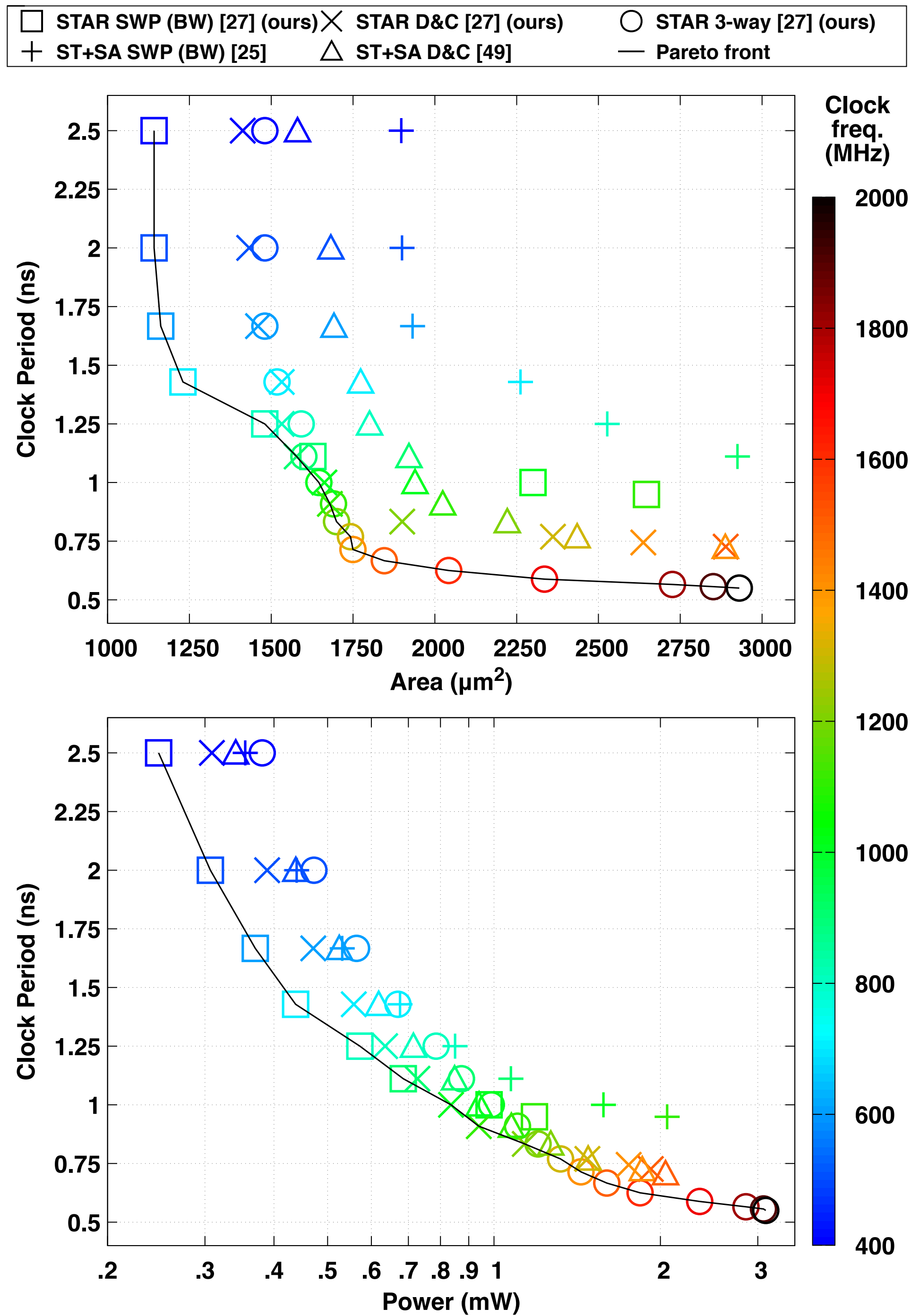


Figure 5.7: PPA comparison of STAR architectures. Image taken from [27].

Chapter 6

Accelerating Quantized DNN Layers on RISC-V with a STAR MAC Unit

Some of the work described in this chapter was also previously published in [31].

In this chapter, we convert a fixed-precision RISC-V processor, known as Ibex [54], into a PS core by replacing the original standard multiplier with a STAR BW one. After discussing the hardware design of the original core and the modification that led to the new PS solution, we assess the execution time, area, and power of the new PS processor compared to the original one by executing various quantized 2D-Conv, DW-Conv and FC layers.

6.1 Ibex: The Baseline RISC-V Processor

The Ibex processor, formerly known as Zero-riscy [54], is an in-order single-issue processor that adheres to the RV32I RISC-V base set [116]. It is available in four different versions: *micro*, *small*, *maxperf* and *maxperf-pmp-bmfull*. In the *micro* and *small* variants, the core features a two-stage pipeline. The first stage handles data fetching operations from memory, while the second stage encompasses all other operations, including writeback. Conversely, the *maxperf* and *maxperf-pmp-bmfull* versions have a three-stage pipeline, with the writeback phase occurring in the third stage. Moreover, the four versions differ in their RISC-V ISA support. The *micro* variant supports the RV32E base set[1] with the C-extension for having compressed instructions on 16 bit, aimed at reducing the code size. The *small* and *maxperf* variants support the RV32I base set with the M- and C-extensions. The

[1] *RV*32*E* stands for 32-bit base RISC-V ISA (*RV*32), with a specific variation for embedded systems (*E*): the number of registers is reduced from the 31 to 15 register.

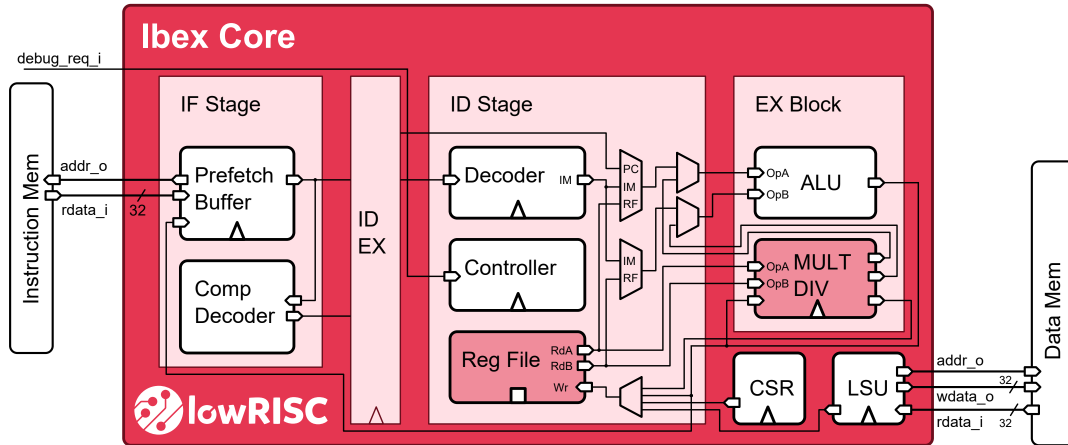


Figure 6.1: The *small* version of the Ibex core: two-stage pipeline, low-power, RV32IMC RISC-V ISA. Image taken from [54].

M-extension adds instructions related to multiplication and division operations. The *maxperf-pmp-bmfull* also supports the B-extension, which provides support for operations involving bit-level manipulation, such as bit insertion, masking, bit test, rotation, funnel shift, bit permutation, among others. Lastly, the *small* version lacks a dedicated branch target ALU and integrates a small Multiplier/Divider (MULT/DIV) unit named *Fast* (*fast-netlist* in [54]). On the other hand, the *maxperf* and *maxperf-pmp-bmfull* versions include the branch target ALU and feature a larger and faster MULT/DIV unit named *SingleCycle*.

In our study, we focus on the *small* version of the Ibex core, in particular on its *Fast* MULT/DIV unit [54][2]. Readers interested in the *SingleCycle* MULT/DIV unit can refer to [117], while those interested in all the other parts of the processor can visit the Ibex official reference [118]. The *small* Ibex is an area-optimized RISC-V core designed to target arithmetic-control mixed applications. It implements the RV32IMC instruction set architecture. Fig. 6.1 provides a simplified overview of its microarchitecture, which comprises two pipeline stages: the Instruction Fetch (IF) and the Instruction Decode and Execute (IDE) stage. The IF stage interacts with the instruction memory subsystem and includes a prefetch buffer for collecting data from the instruction memory, handling compressed instructions, and generating the instruction address and program counter values. It also features a First In First Out (FIFO) buffer to store instructions when the subsequent stage is not ready to process them. The IDE stage is responsible for decoding instructions, reading operands from the register file, preparing operands for the ALU and the multiplier unit, and executing instructions. The register file is a 2-read-1-write

[2]The version of the Ibex used in our work can be found at `https://github.com/lowRISC/ibex/tree/8db89a9dfc0cb08371d079cfc76e83d9ffc66480`. Accessed on: Mar 9, 2024.

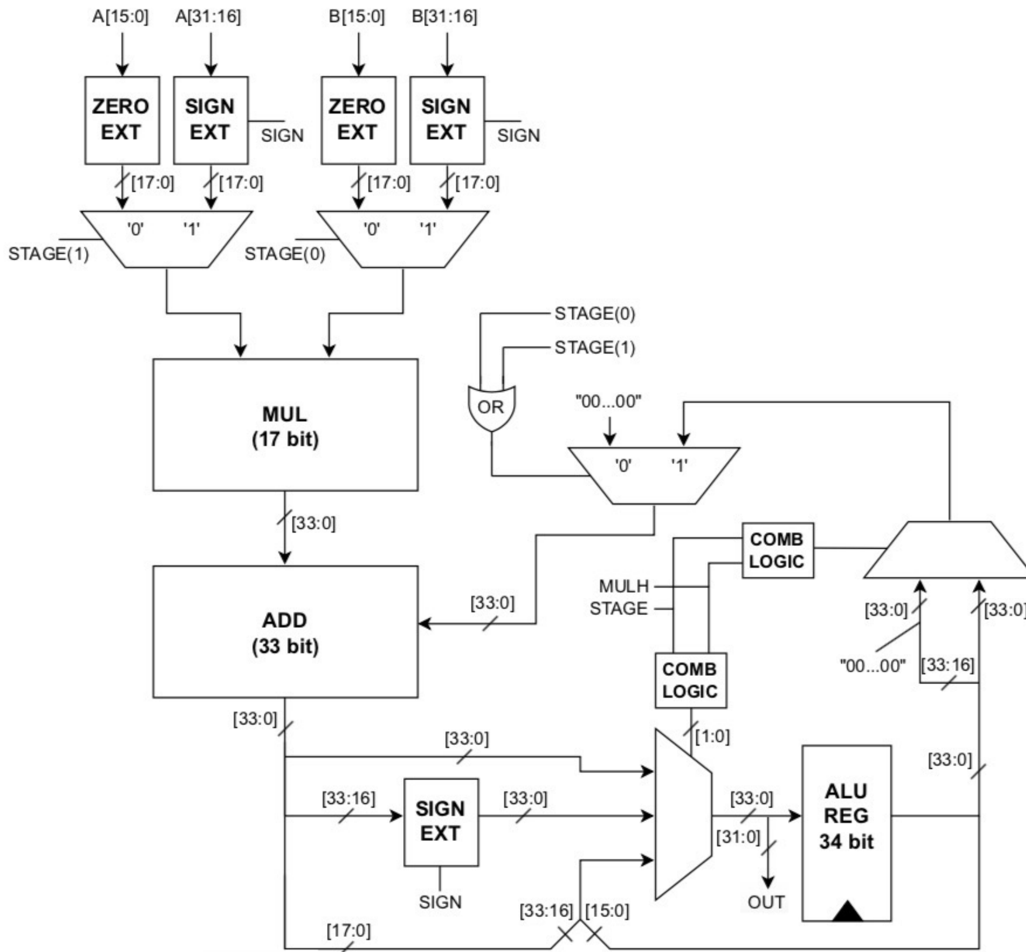


Figure 6.2: Overview of the original *Fast* MULT/DIV unit of the Ibex core. Image taken from [117].

latch-based register file. The ALU consists of essential hardware resources to support the RV32IMC ISA, including a 32-bit adder, a 32-bit shifter, and a logic unit. The 32-bit adder performs additions, subtractions, and comparisons, and is shared with the data address generation unit, branch engine, and divider. Instructions involing branches, multiplications, divisions, loads and stores are executed iteratively, causing the next instruction to stall until completion.

6.2 The *Fast* MULT/DIV unit of the *small* Ibex

The *Fast* version of Ibex's MULT/DIV unit is depicted in Fig. 6.2. It includes a single signed 17-bit multiplier, which handles all the multiplication instructions of the RISC-V RV32IM ISA, that are MUL, MULH, MULHU, and MULHSU. It is

implemented with 17 bits, rather than the more common 16-bit version, to accommodate all potential multiplication scenarios with a single multiplier, supporting signed×signed, unsigned×unsigned, and signed×unsigned/unsigned×signed operations. Indeed, when dealing with 16-bit operands, unsigned values are zero-extended to 17 bits, while signed values are sign-extended. The MULT/DIV unit also includes a 33-bit adder, which takes the output of the multiplier and the content of *ALU REG* as inputs. The latter is the output register of the ALU. This register is re-used by the MULT/DIV unit to accumulate the multiplier's partial results over time, allowing for 32-bit multiplications to be completed in multiple clock cycles. A finite state machine (FSM) controls the operation of the MULT/-DIV unit and generates the 2-bit signal *STAGE*, which indicates the iteration clock cycle (ICC) of the FSM. During each ICC of a 32-bit multiplication, the appropriate 16-bit chunks from the 32-bit operands A and B, after being sign- or zero-extended depending on the multiplication type (i.e., signed/unsigned), are selected by a couple of input multiplexers controlled by the *STAGE* signal and sent to the 17-bit multiplier. The resulting product is fed to the 33-bit adder, where it is eventually accumulated with the previous partial results stored in *ALU REG*. The selection of the values to be stored in the *ALU REG* and delivered to the adder is determined by a minimal combinational logic (*COMB LOGIC*) based on *STAGE* and the instruction type. This process continues until the 32-bit multiplication is completed. As illustrated in Fig. 6.7, the number of ICCs required depends on the instruction type: for the MUL instruction, which returns the lower 32 bits of the resulting 64 bits through the 32-bit *OUT* signal (Fig. 6.2), the computation takes three clock cycles; for MULH, MULHU, or MULHSU instructions, which return the upper 32 bits of the result, it takes four clock cycles. Specifically, MULH returns the upper-part if both operands are signed, MULHU if they are both unsigned, and MULHSU if the first operand is signed and the second is unsigned.

6.3 The novel STAR MAC unit integrated in the *small* Ibex

We replace the 17-bit multiplier within the *Fast* MULT/DIV unit of the *Small* Ibex processor with our SWP STAR multiplier, as depicted in Fig. 6.3 and discussed in Sec. 6.3.1. This substitution results in a novel STAR MAC unit, showcased in Fig. 6.5 and analyzed in Sec. 6.3.2. The STAR multiplier executes one 16-bit multiplication, or N=2, 4 parallel low-precision multiplications with 16/N-bit operands for both SA/ST operating modes. The STAR MAC unit still supports all multiplication instructions of the RISC-V RV32IM ISA and preserves the iterative approach of the original MULT/DIV unit, requiring multiple ICCs to complete an instruction. Moreover, it supports MAC operations, which are not available in the original MULT/DIV unit. Specifically, we add a standard 32-bit MAC instruction, which

Table 6.1: New MAC instructions and number of required clock cycles.

Instruction	**Operation**	**Clock Cycles**
MAC	$O_{[31:0]} = A_{[31:0]} \times B_{[31:0]} + D_{[31:0]}$	3
MAC16ST	$O_{[31:0]} = A_{[15:0]} \times B_{[31:16]} + A_{[31:16]} \times B_{[15:0]} + D_{[31:0]}$	2
MAC8ST	$O_{[23:0]} = A_{[7:0]} \times B_{[31:24]} + A_{[15:8]} \times B_{[23:16]} + + A_{[23:16]} \times B_{[15:8]} + A_{[31:24]} \times B_{[7:0]} + D_{[31:8]}$	2
MAC4ST	$O_{[19:0]} = A_{[3:0]} \times B_{[31:28]} + A_{[7:4]} \times B_{[27:24]} + + A_{[11:8]} \times B_{[23:20]} + A_{[15:12]} \times B_{[19:16]} + + A_{[19:16]} \times B_{[15:12]} + A_{[23:20]} \times B_{[11:8]} + + A_{[27:24]} \times B_{[7:4]} + A_{[31:28]} \times B_{[3:0]} + D_{[31:12]}$	2
MAC16SA	$O_{[31:0]} = A_{[15:0]} \times B_{[31:16]} + D_{[31:0]}$	2
MAC8SA	$O_{[15:0]} = A_{[7:0]} \times B_{[23:16]} + D_{[15:0]}$ $O_{[31:16]} = A_{[15:8]} \times B_{[31:24]} + D_{[31:16]}$	2
MAC4SA	$O_{[7:0]} = A_{[3:0]} \times B_{[19:16]} + D_{[7:0]}$ $O_{[15:8]} = A_{[7:4]} \times B_{[23:20]} + D_{[15:8]}$ $O_{[23:16]} = A_{[11:8]} \times B_{[27:24]} + D_{[23:16]}$ $O_{[31:24]} = A_{[15:12]} \times B_{[31:28]} + D_{[31:24]}$	2
MAC16SAH	$O_{[31:0]} = A_{[31:16]} \times B_{[15:0]} + D_{[63:32]}$	2
MAC8SAH	$O_{[15:0]} = A_{[23:16]} \times B_{[7:0]} + D_{[47:32]}$ $O_{[31:16]} = A_{[31:24]} \times B_{[15:8]} + D_{[63:48]}$	2
MAC4SAH	$O_{[7:0]} = A_{[19:16]} \times B_{[3:0]} + D_{[39:32]}$ $O_{[15:8]} = A_{[23:20]} \times B_{[7:4]} + D_{[47:40]}$ $O_{[23:16]} = A_{[27:24]} \times B_{[11:8]} + D_{[55:48]}$ $O_{[31:24]} = A_{[31:28]} \times B_{[15:12]} + D_{[63:56]}$	2
MACSET	$\text{MAC REG}_{[63:0]} = \{A_{[31:0]}, B_{[31:0]}\}$	1

takes 3 clock cycles, and custom low-precision ST/SA MAC instructions (MACySA and MACyST, $y \in \{4,8,16\}$), which exploit the STAR multiplier and take 2 clock cycles. All the new MAC instructions are reported in Table 6.1, where A and B are the two-input operands, D is the value to accumulate, and O is the output of the STAR MAC unit. We report only the MAC instructions that support signed operands because these are the instructions that will be used in the benchmarks of quantized DNN layers discussed in Sec. 6.4. However, we have also defined and implemented the equivalent MAC instructions (MACHSU and MACHU) of the corresponding unsigned RV32IM MUL instructions (MULHSU and MULHU). We will not provide further details regarding MACHSU and MACHU instructions in this thesis. For additional information, the interested readers can refer to [117], which covers the encoding of the new instructions, modifications to the decode unit, and

customization of the GCC compiler for recognizing the new assembly instructions.

6.3.1 STAR BW Multiplier

The core of our STAR MAC unit is the 16-bit STAR multiplier, whose BW architecture is depicted in Fig. 6.3. Its design resembles the one of STAR SWP (BW), previously described in Fig.5.6 (top), Sec. 5.2; however, it now supports unsigned and MAC operations, too. In particular, the latter are performed by accumulating a third input, denoted as C, within the architecture during the multiplication process of inputs A and B (explained later). In addition, note that SA instructions now multiply different subwords of A and B compared to STAR SWP (BW). For example, MAC16SA now performs the operation $\mathrm{A}_{[15:0]} \times \mathrm{B}_{[31:16]}$, whereas in STAR SWP (BW) it was $\mathrm{A}_{[15:0]} \times \mathrm{B}_{[15:0]}$. This optimization reduces the number of input multiplexers and increases their reuse among SA and ST operations.

Let us recap the functionalities of STAR applied to a BW architecture and discuss how the new features affect the previous design.

STAR BW consists of a 16×16 PPM that works as a typical BW multiplier [50]: each block receives a pair of bits from the two input operands, computes the partial product (PP) between them using an AND gate; then it generates the output sum So and carry Co bits by compressing the PP, along with the input sum Si and input carry Ci bits, using a FA. The sum bits propagate diagonally, while the carry bits propagates vertically, connecting all the blocks of the PPM (not shown in Fig. 6.3a for better clarity).

To support low-precision SA and ST operating modes, in addition to the standard full-precision 16-bit multiplication, the PPM requires to be reconfigured in one of the five ways illustrated in Fig. 6.4, where the top square of each configuration represents the PPM and the bottom rectangle corresponds to the final 32-bit result R of the STAR multiplier. The PPs in the yellow areas of the PPM contribute to generate the yellow bits of R, while the PPs in the grey areas are gated and do not contribute to it. To obtain this behavior, we create three configurable blocks (white, red and blue, Fig. 6.3b–d) by adding few logic gates controlled by signals P (*propagate*) and I (*invert*), and we place them in specific positions in the STAR PPM.

As shown in Fig. 6.4, depending on the STAR configuration, any block of the PPM could be potentially be gated. Thus, each block of the STAR PPM must have the possibility of blocking the PP propagation. For this reason all the three configurable blocks have an AND gate which stops the propagation of the PP towards the FA when $P = 0$.

To support signed operations, we need to take into account two aspects. The first is that the PPs belonging to the blocks in the left-most column and bottom row of a typical BW PPM have to be inverted. Since in any ST/SA configuration of STAR each yellow square of Fig. 6.4 behaves like an independent low-precision

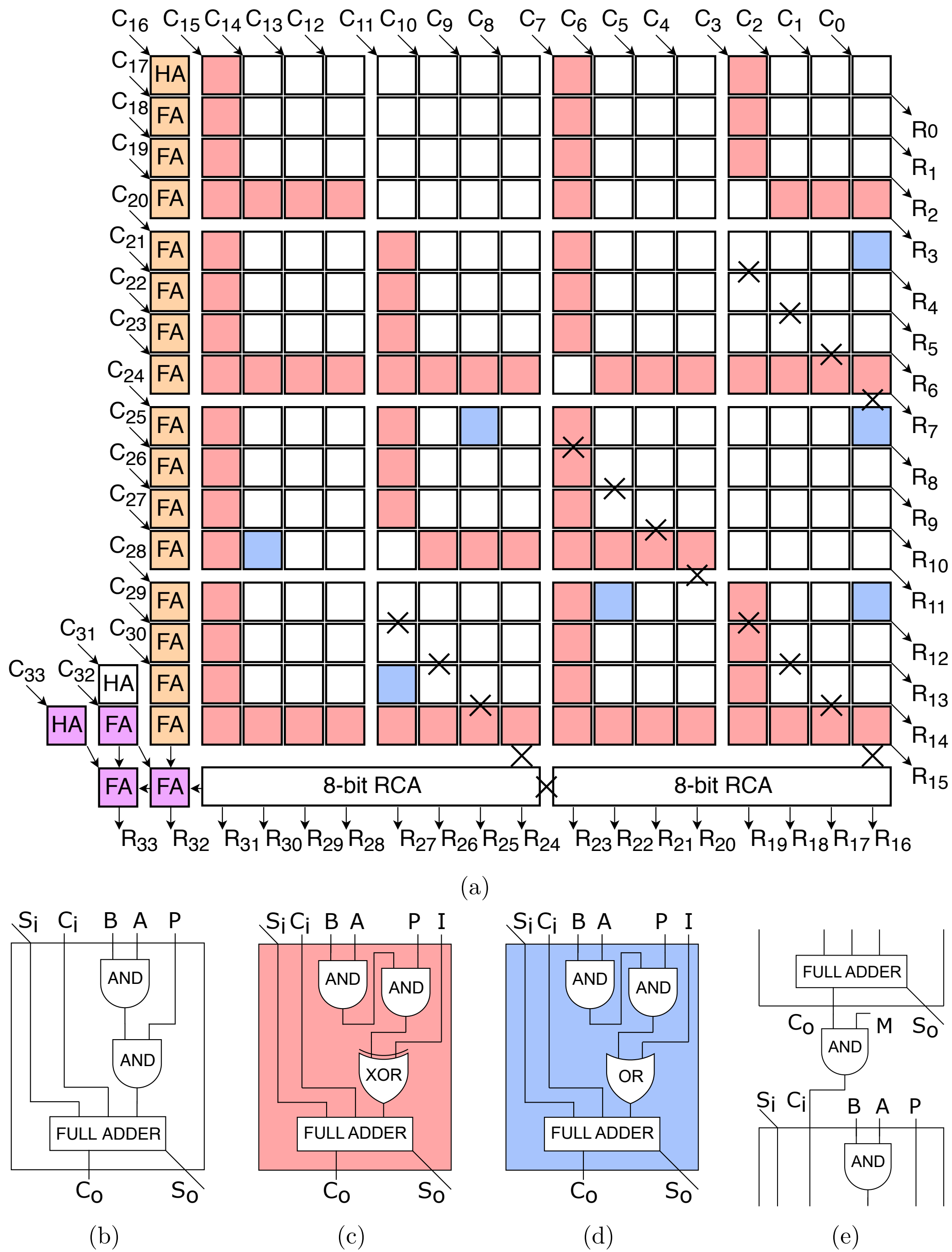


Figure 6.3: PPM of the STAR BW multiplier (a), three versions of PPM blocks (b-d), and carry propagation blocking strategy (e). Images taken from [31].

BW multiplier, we need to guarantee the PP inversion in all the left and bottom blocks of these sub-precision multipliers, as well. For this task, we conceive the

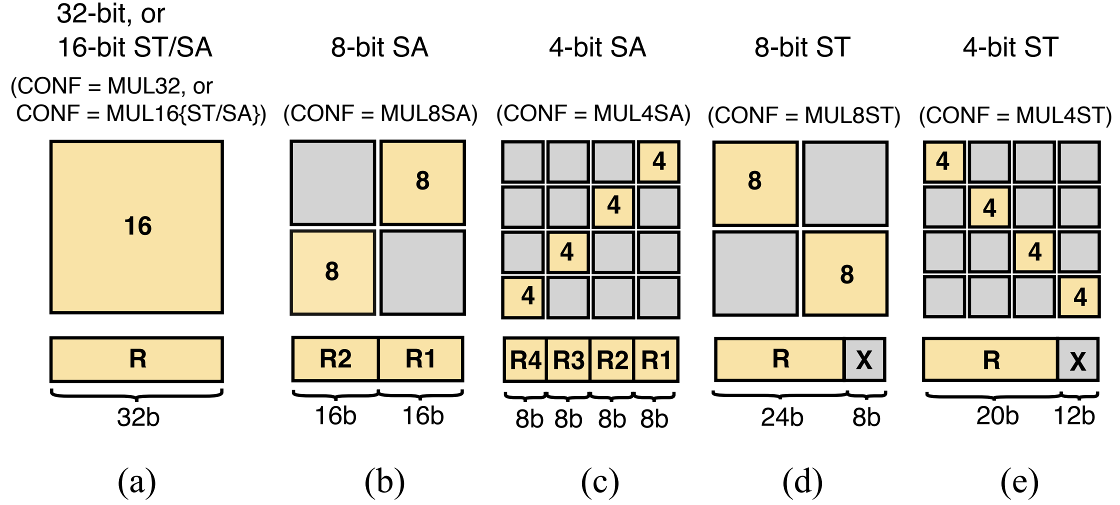


Figure 6.4: The five operating modes of the STAR BW multiplier, where the top square of each configuration is the BW PPM and the bottom rectangle is the multiplier's output. Images taken from [31].

red block (Fig. 6.3c), which inverts the output of the PP when $I = 1$ via an XOR gate. The second aspect is the insertion of logic 1s in specific positions of the PPM to accomplish the BW algorithm [50]. Typically, the insertion of these 1s occurs through the Si inputs of the top and left-most blocks of the BW PPM as it happens for i and g in the PPM of Fig.5.6 (top), Sec. 5.2. However, we decide to use those inputs to add the third operand of the MAC operation, i.e., the 34-bit signal C (as cleared later). Hence, when possible, we generate these 1s using the inactive red blocks (with $P = 0$ and $I = 1$) inside the PPM. When not possible, we insert the blue blocks (Fig. 6.3d), which use an OR gate to force the PP to be logic 1 (again with $P = 0$ and $I = 1$). Compared to the previous PPM shown in Fig.5.6 (top), Sec. 5.2, we also replace the green blocks with more generic red ones and substitute the white block in the bottom left corner with a red one to enable STAR to perform unsigned operations. For what concerns the orange FAs placed on the left side of the PPM and of the pink ones at the bottom-left corner, we will explain their purpose later in Sec. 6.3.2.

The final output of STAR BW is the concatenation of the eighteen bits at the output of the RCA (R[33:16]), which compresses the So and carry Co bits of the last row of the PPM, and the sixteen So bits exiting from the rightmost column of the PPM (R[15:0]). To guarantee that R is separated in two or four independent subwords for 8-bit and 4-bit SA operations, we have to interrupt the propagation of the carry between two consecutive low-precision multiplications. Therefore, we insert a few AND gates, controlled by signal M (Fig. 6.3e), between two vertically adjacent blocks, as shown in Fig. 6.3a by the X symbols. We also add an AND gate between the two 8-bit RCAs.

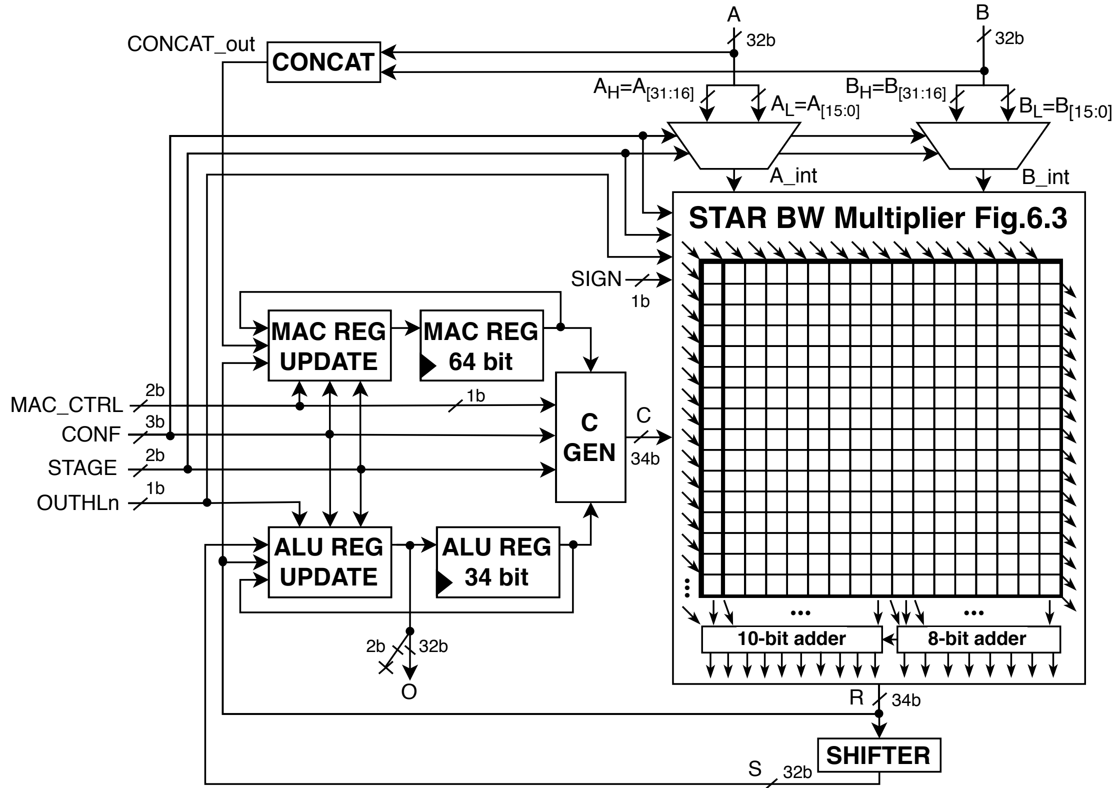


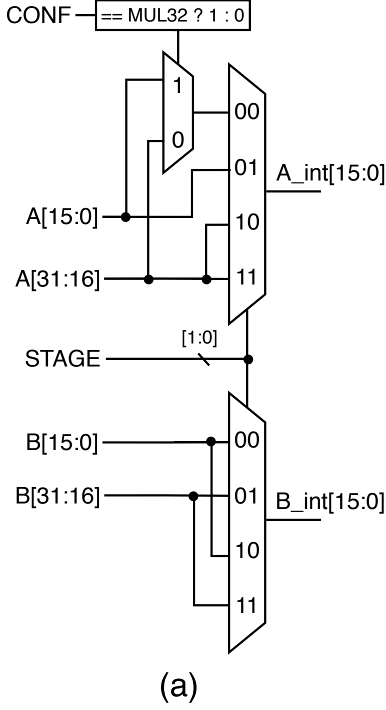
Figure 6.5: The STAR MAC unit implemented in the Ibex core. The *STAR BW Multiplier* is the one reported in Fig. 6.3. Image taken from [31].

Finally, signals P and I are generated by a combinational logic that uses the following control signals: *CONF*, a 3-bit signal indicating the actual operation type (SA/ST at 32/16/8/4 bits); *OUTHLn*, a single bit indicating which part (32-bit high/low) of the 64-bit final result has to be returned by the instruction (note: SA operations can have multiple separate results in the 32-bit high/low of the result, according to Table 6.1); *SIGN*, a bit indicating if the instruction works with signed or unsigned operands; and *STAGE*. Note that the first three signals come directly from the instruction opcode.

6.3.2 STAR MAC unit

Like in the original MULT/DIV unit, the 32-bit input operands A and B come from the register file. The proper upper and lower 16-bit chunks from A and B are selected by the input multiplexers depending on the ICC. However, in our MAC STAR unit these two multiplexers are controlled not only by *STAGE*, but also by *CONF* because we have to deal with ST and SA instructions (Fig. 6.6a).

For MUL instructions, at each ICC the combinational logic *ALU REG UPDATE*



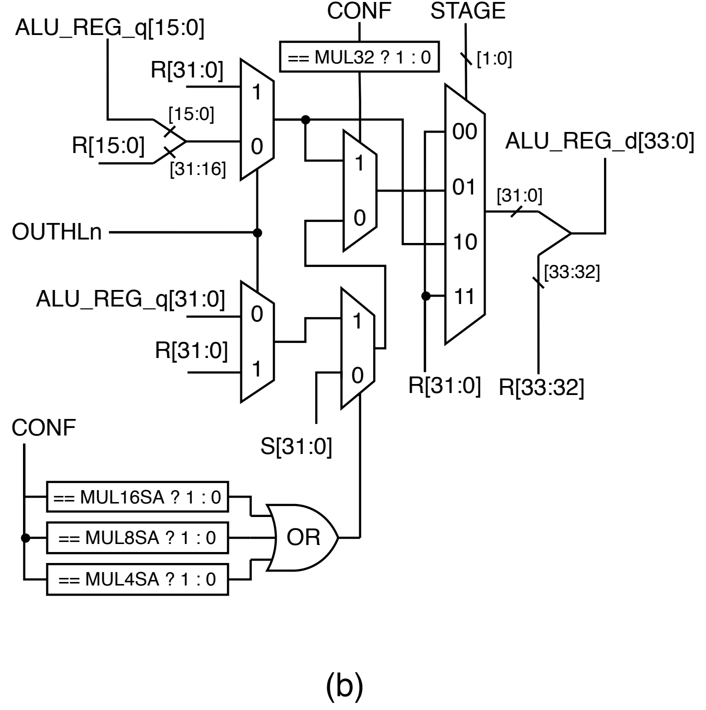
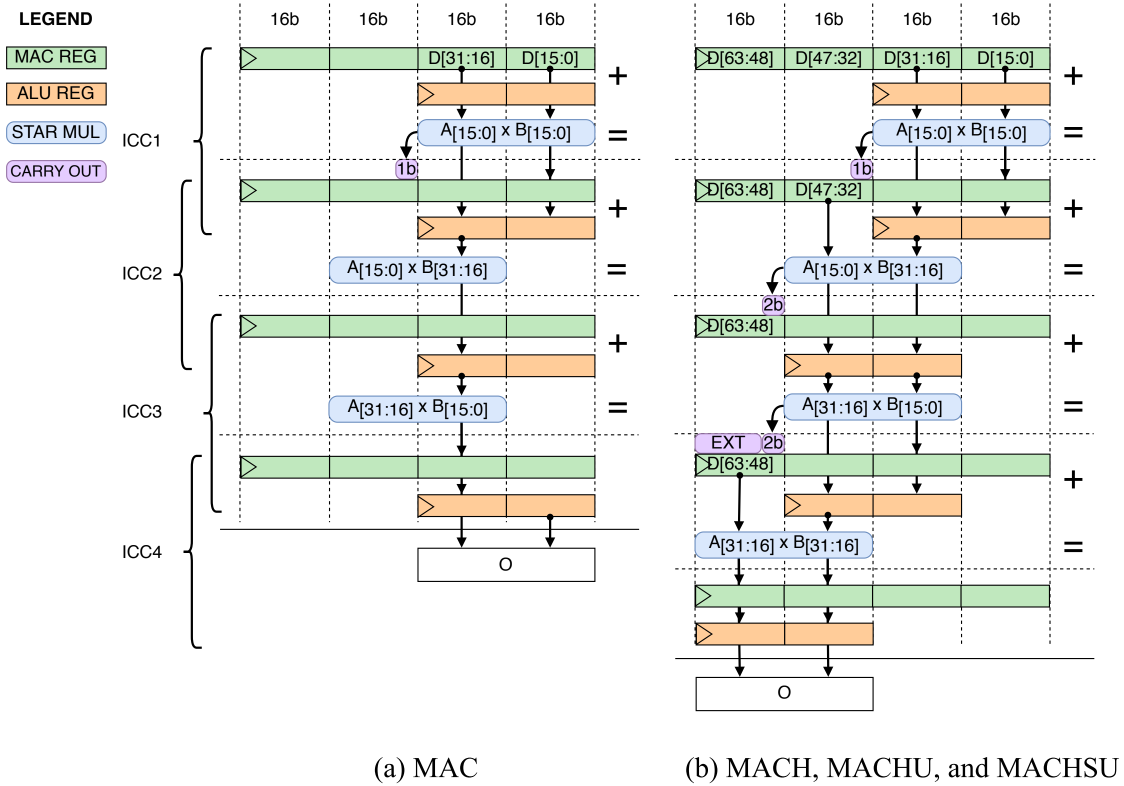


Figure 6.6: Schematics describing: (a) the input multiplexers (of Fig. 6.5) and (b) the *ALU REG UPDATE* block. Images taken from [31].

in Fig. 6.5 selects which subword of R to store in the register *ALU REG* for the next ICC. In fact, the content of *ALU REG* is used by the multiplier in the subsequent ICC to continue the iterative multiplication process (as clarified later). *ALU REG UPDATE* takes as inputs the multiplier's output R, the *SHIFTER*'s output S and the content of *ALU REG*, and is controlled by *STAGE*, *OUTHLn* and *CONF* (the first two are present also in the original MULT/DIV unit), as reported in Fig. 6.6b.

For MAC instructions, we introduce the 64-bit accumulation register *MAC REG*, along with the combinational logic *MAC REG UPDATE* responsible for its updating. During MAC instructions, the contents of *MAC REG* and *ALU REG* are both used by the STAR multiplier in the subsequent ICC to continue the iterative multiply and accumulate process. Indeed, *ALU REG* serves the same purpose as in the MUL instructions case, which is to temporarily store R to use it in the next ICC, whereas *MAC REG* is used to store the partial accumulated value. *MAC REG UPDATE*, whose schematic is reported in Fig. 6.8a, behaves similarly to *ALU REG UPDATE*: it takes R, the content of *ALU REG* and the concatenation of A and B (i.e., the output of the *CONCAT* block, which is called *CONCAT_out*) to update *ALU REG* according to *STAGE*, *CONF* and *MAC_CTRL* control signals.

For both MUL and MAC instructions, the third input C to the STAR BW

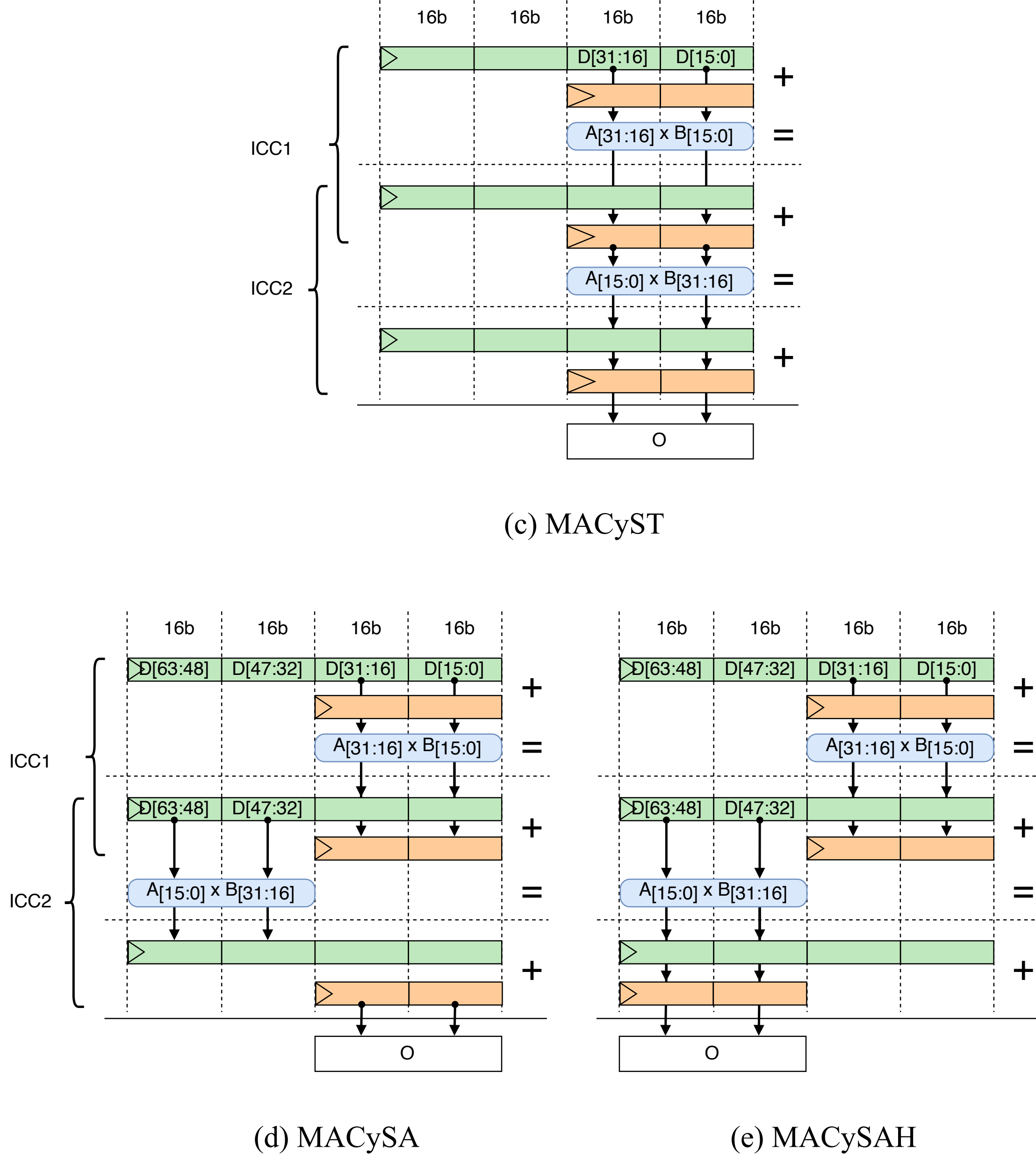


(a) MAC (b) MACH, MACHU, and MACHSU

Figure 6.7: The operations performed by STAR MAC at each ICC for (a) MAC, (b) MACH, MACHU, and MACHSU, (c) MACyST, (d) MACySA, and (e) MACySAH instructions ($y \in \{4,8,16\}$).

multiplier comes from the combinational logic *C GEN*. The schematic of *C GEN* is reported in Fig. 6.8b. Through control signals *MAC_CTRL* (a 2-bit signal which indicates if the current instruction is a MAC (01b), a MACSET (11b), or a MUL (00b)), *STAGE* and *CONF*, *C GEN* selects *ALU REG* in case of standard MUL operations, or the proper sub-words of *MAC REG* and *ALU REG* in case of MAC operations.

Fig. 6.7 illustrates in detail which operations are performed by STAR MAC at each ICC for MAC (Fig. 6.7a), MACH (Fig. 6.7b), MACyST (Fig. 6.7c), MACySA (Fig. 6.7d), and MACySAH (Fig. 6.7e) instructions ($y \in \{4,8,16\}$). The blue rounded-edge rectangles show the 16-bit subwords of A and B processed by the STAR multiplier at each ICC. The green and orange sharp-edge rectangles are *MAC REG* and *ALU REG*, respectively. The bits generated by STAR at each ICC are always saved in a portion (16 bits) or the entire content (32 bits) of *ALU REG* (as shown by the arrowheads) and return in input to STAR at the next ICC through C in order to get the expected final MAC result. Only when the generated bits represent the



(c) MACyST

(d) MACySA

(e) MACySAH

Figure 6.7: The operations performed by STAR MAC at each ICC for (a) MAC, (b) MACH, MACHU, and MACHSU, (c) MACyST, (d) MACySA, and (e) MACySAH instructions ($y \in \{4,8,16\}$).

final MAC results, they are saved in *MAC REG*, allowing the next MAC operation to start with the previously accumulated result already stored in *MAC REG*. In particular, for 32-bit MAC instructions the result is accumulated in *MAC REG* up

to 32-bit precision, while for 32-bit MACH, MACHSU, MACHU instructions the result is accumulated up to 64-bit precision. For SA instructions, instead, *MAC REG* stores the accumulated results in separate data chunks: two 32-bit chunk for MAC16SA, four 16-bit chunks for MAC16SA, or eight 8-bit chunks for MAC4SA. For ST instructions, results are accumulated and stored with a precision of 32, 24, or 20 bits for MAC16ST, MAC8ST, or MAC4ST, respectively.

At the last ICC, each instruction returns the output O as final result, which corresponds to the 32 LSBs coming from the output of *ALU REG UPDATE* (Fig. 6.5). This means that while MACySA computes the complete 64-bit result and stores it in *MAC REG*, it only returns the lower 32 bits (Fig. 6.7d). To return the upper 32 bits, which remain stored in *MAC REG*, the user needs to use MACySAH (Fig. 6.7e) with both A and B set to zero. The output of *ALU REG UPDATE* can be: S for MAC8ST and MAC4ST operations, or R for MAC16ST and all the other non-ST operations (as visible in Fig. 6.6b). In fact, in the former case the *SHIFTER* is used to get rid of the 8 or 12 invalid LSBs of R (as reported in grey in the result of Fig. 6.4d–e).

Thanks to Fig. 6.7a–b, we are now able to explain the purpose of the orange and pink FAs placed on the left and bottom-left sides of the PPM of the STAR BW multiplier. These FAs enable the correct accumulation of C inside the PPM. For MAC, MACH, MACHU, and MACHSU instructions, the 32-bit multiplication result between A and B needs to be summed with C. Thus, one or two carry output bits are produced at ICC1, ICC2 and ICC3, as highlighted in purple in Fig. 6.7a–b. In order to have the correct final MAC result, these bits need to be accumulated along the ICCs. In particular, these bits also need to be sign extended before accumulation at ICC4 during MACH, MACHU, and MACHSU instructions. Therefore, in the STAR BW multiplier we place a column of orange FAs and we insert these carry bits in the their available input (not shown in Fig. 6.3a for better readability). Furthermore, having these two more bits to sum requires a longer RCA. This is the reason for the two additional pink FAs that extend the left RCA from 8 to 10 bits.

Regarding the MACSET instruction (Table 6.1), it is commonly used to initialize *MAC REG* with the value of *CONCAT_out* at the beginning of a software routine that requires a series of MAC instructions. *CONCAT_out* is the concatenation of A and B as shown in Fig. 6.5. Therefore, through this instruction, the user can initialize *MAC REG* with operand D passing it through A and B signals.

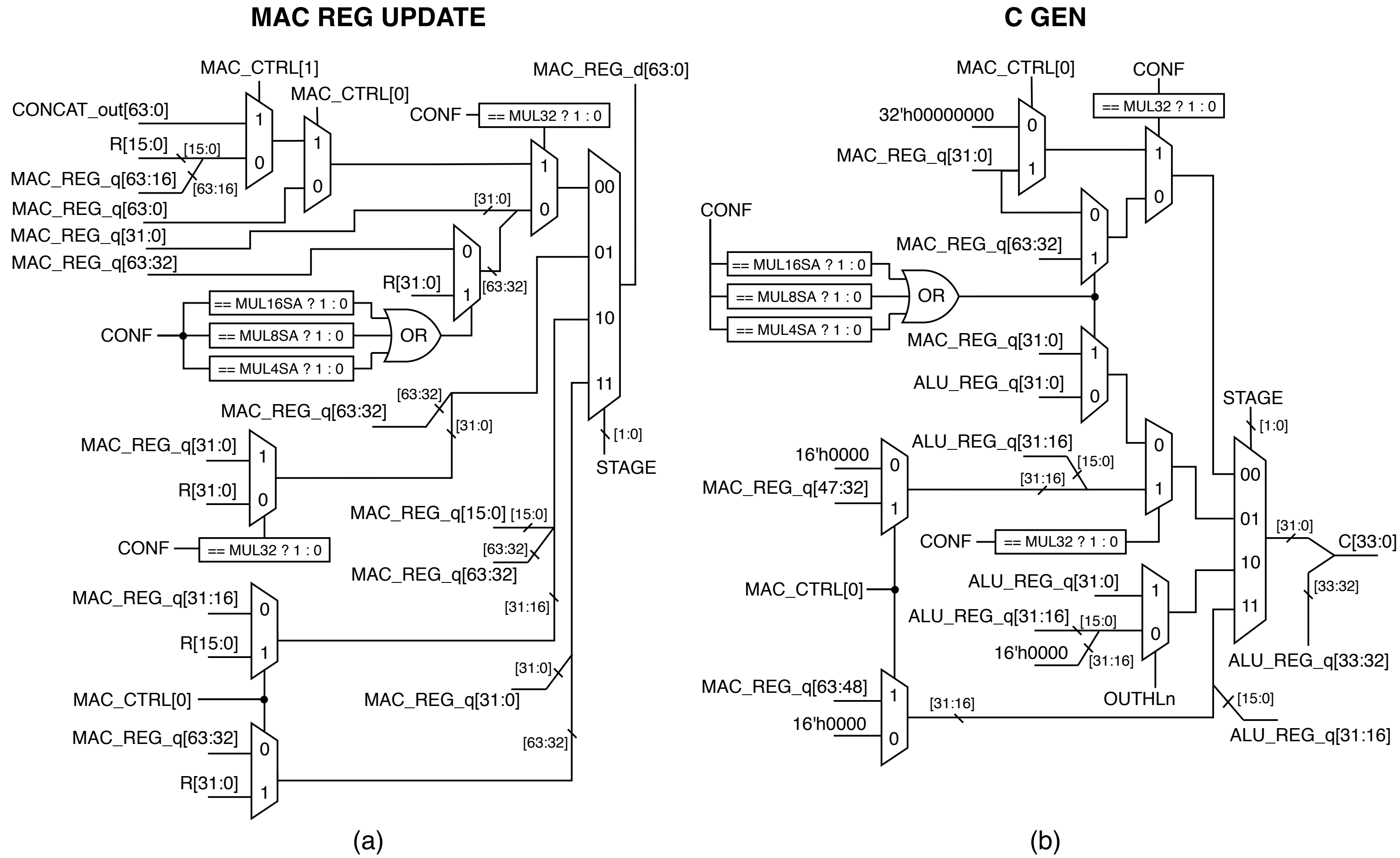


Figure 6.8: Schematics of: (a) *MAC REG UPDATE* and (b) *C GEN* blocks. Images taken from [31].

Table 6.2: Logic synthesis results of *Orig.*, *Orig. + MAC* and *STAR-based*.

Ibex type	Clk. Freq. [MHz]	Area [μm^2]	Area vs *Orig.* [%]	Area vs *Orig.+ MAC* [%]	Power [mW]	Power vs *Orig.* [%]	Power vs *Orig.+ MAC* [%]
Orig.	200	14241			1.46		
Orig.+MAC	200	14658	+2.9		1.50	+2.7	
STAR-based	200	15299	+7.4	+4.4	1.50	+2.7	+0.0
Orig.	600	15135			4.17		
Orig.+MAC	600	15588	+3.0		4.27	+2.4	
STAR-based	600	16528	+9.2	+6.0	4.29	+2.9	+0.5

6.4 Experimental Results

The results presented in this chapter are preliminary, with further experiments currently ongoing.

6.4.1 Implementation Results

The possibility to perform parallel low-precision MAC operations, due to the STAR MAC unit, enables the acceleration of quantized DNN layers on the modified Ibex core (which we call call *STAR-based*). To have a fair comparison in area, power and execution latency of DNN layers, we decide to slightly modify the original Ibex (*Orig.*) to add the support to MAC instructions, resulting in a third Ibex implementation (which we name *Orig. + MAC*).

We synthesize all the three Ibex versions on a 28-nm technology (0.9 V) at 200 and 600 MHz, two tight constraints for this processor according to the original work [54]. The results of area and estimated power are reported in Table 6.2. As we can see, *STAR-based* has limited area and power overheads compared to *Orig.*: around 7% and 3% at 200 MHz and around 9% and 3% at 600 MHz. When compared to *Orig. + MAC*, the area overhead decreases even further: around 4% at 200 MHz, and 6% at 600 MHz, while the power overheads drop almost to 0%.

6.4.2 Performance on Quantized DNN layers

Then, we evaluate the performance of *STAR-based* vs *Orig. + MAC* on these quantized layers: FC, 128-256 input and 32 output neurons; 2D-Conv, 32-128 input channels (inch.), 8x8 feature map (fmap.), 3x3 kernel (kern.) and 4 output channels (outch.); DW-Conv, 16-64 channels (ch.), 16x16 feature map size and 3x3

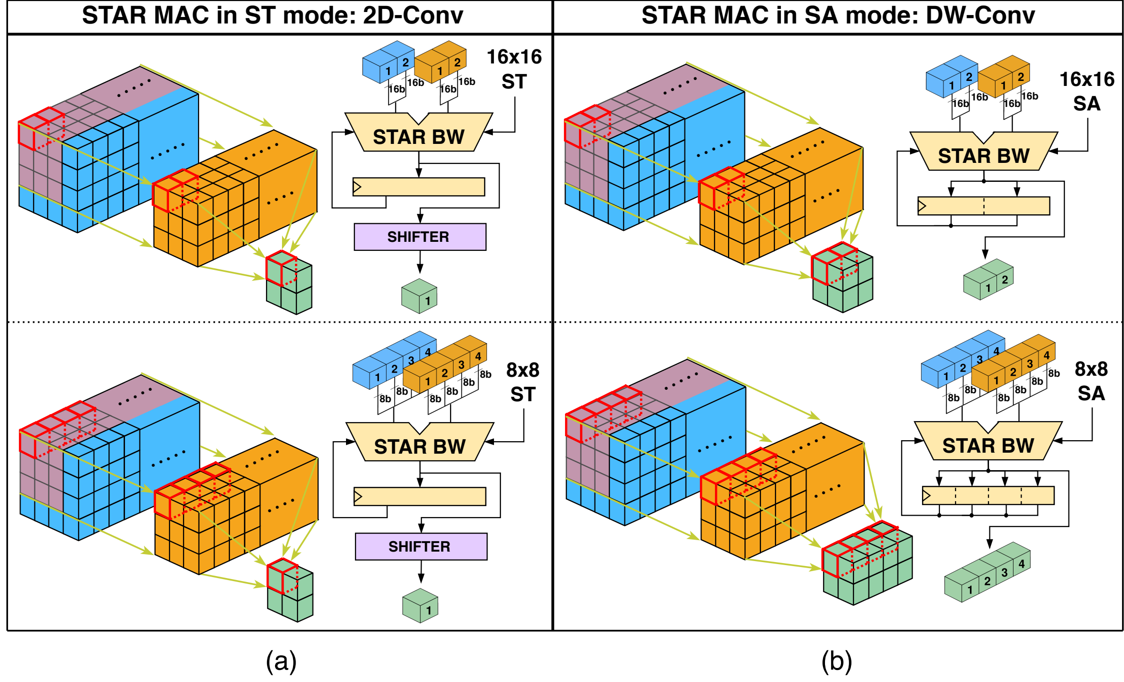


Figure 6.9: Two examples of STAR MAC used in 2D-Conv and DW-Conv layers: MAC16{ST/SA} in the upper part, MAC8{ST/SA} in the lower part. Image taken from [31].

kernel. We assume that the low-precision operands are already packed/prearranged in memory in such a way that the instructions listed in Table 6.1 can be directly executed without incurring any costs due to operand reordering. To exploit the acceleration provided by STAR at reduced precision, we code these layers with an output-stationary dataflow [60]. We use MACyST instructions for FC/2D-Conv and MACySA instructions for DW-Conv ($y \in \{4,8,16\}$) because the former layers require the accumulation along the input channels/neurons dimension, while the latter does not [24, 28]. In this regard, Fig. 6.9 shows two examples, for 2D-Conv (MAC16ST and MAC8ST, Fig. 6.9a) and DW-Conv (MAC16SA and MAC8SA, Fig. 6.9b), on how low-precision input features (light blue) and weights (orange) are read from the corresponding tensors and packed in the two 32-bit input registers of the STAR MAC unit to produce the output features (green). Both the approaches can be easily extended to MAC4{ST/SA} instructions and to the FC layer [24, 28]. Furthermore, to have a fair comparison between the *STAR-based* and *Orig. + MAC* cores, we ensure that the code of each DNN layer for both cores features an equal number of memory accesses. As a result, the comparison exclusively highlights the computational advantages stemming from the new STAR MAC unit.

The results of the average speedup, i.e., the ratio between the clock cycles, of *STAR-based* vs *Orig. + MAC* for these three DNN layers at different features and

Table 6.3: Average speedup (i.e., ratio between clock cycles) of *STAR-based* vs *Orig.* (column 2) and of *STAR-based* vs *Orig.+MAC* (column 3), for three DNN layers for different features and weights bitwidths.

DNN layers	**Instruction**	**Avg. Speedup *STAR-based* vs. *Orig.* ($\mathbf{y = 16, 8, 4}$)**	**Avg. Speedup *STAR-based* vs. *Orig.+MAC* ($\mathbf{y = 16, 8, 4}$)**
FC (128-256 input, 32 output)	MACySТ	2.0x, 3.3x, 5.8x	1.7x, 2.7x, 4.5x
2D-Conv (32-128 inch., 8x8 fmap., 3x3 kern., 4 outch.)	MACyST	1.6x, 2.3x, 3.7x	1.4x, 1.9x, 3.0x
DW-Conv (16-64 ch., 16x16 fmap., 3x3 kern.)	MACySA	1.4x, 1.9x, 2.8x	1.3x, 1.6x, 2.3x

weights precision is reported in the last column of Table 6.3. For the 16-, 8-, and 4-bit cases, respectively, the average speedup for FC is 1.7×, 2.7× and 4.5×; for 2D-Conv is 1.4×, 1.9× and 3.0×; for DW-conv is 1.3×, 1.6× and 2.3×.

We also report the average speedup of *STAR-based* vs *Orig.* for the same DNN layers in the second column of Table 6.3. We ensure, once again, that the code of these DNN layers has the same number of memory accesses for both processors for a fair comparison. For this comparison, the average speedup for the 16-, 8-, and 4-bit cases is, respectively, 2.0×, 3.3× and 5.8× for FC; 1.6×, 2.3× and 3.7× for 2D-Conv; is 1.4×, 1.9× and 2.8× for DW-Conv.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With our research, we delved into the realm of Precision-Scalable (PS) hardware architectures, focusing on multipliers/Multiply-and-Accumulate (MAC) units, Application-Specific Integrated Circuit (ASIC) accelerators, and processors tailored for Transprecision Computing (TC) and/or computing Deep Neural Networks (DNNs) quantized in mixed-precision (MP) at the edge. The main findings of this thesis are outlined below.

- **In Chapter 3**, we provided a comprehensive comparison of the main *Sum-Together (ST)* multipliers documented in the literature [28].

    A) We outlined the architectures of the State-of-the-Art (SoA) ST multipliers.

    B) We expanded upon previous work by proposing three new designs. The first, named *BW-ADD*, shortens the critical path of the Ripple Carry Adder (RCA) of the original Baugh-Wooley (BW) ST multiplier [25]. The second, *HLS ST*, is derived from High-Level Synthesis (HLS) [28]. The third, called *Booth ST*, is the first ST multiplier based on the Booth architecture [26, 50].

    C) We evaluated their performance, power, and area (PPA) characteristics with a design space exploration (DSE) in a 28-nm technology in the [0.5, 1.5 GHz] clock frequency range, after normalizing their architectures to support 16, 8, and 4 bits of precision.

  The results showed that:

    - BW-ADD is among the best in area in the low-frequency range, confirming that the BW architecture is not suitable for higher frequencies, even with a faster adder, due to its inherently long critical paths;

  - Solutions based on dedicated multipliers for each configuration (like [48], [41], HLS ST) are inefficient in area due to redundant logic gates, unlike single high-precision multipliers working in a *Sub-word Parallel (SWP)* manner ([25], BW-ADD and Booth ST), which have a higher utilization ratio of their logic gates, resulting in lower area, especially with less stringent timing constraints;
  - Booth ST offers the best trade-off in area vs clock period for most frequencies;
  - [25] and BW-ADD are Pareto-optimal in power at low frequencies;
  - Booth ST and [41] are the best in both area and power at high frequencies.

  In summary, the optimal ST multiplier solutions depend on the PPA constraints.

- **In Chapter 4**, we focused on ST-based Deep Learning (DL) accelerators for the three most common layers of DNNs: 2D-Convolution (2D-Conv), Depthwise Convolution (DW-Conv), and Fully-Connected (FC) [28]. These ASIC accelerators are PS and can be reconfigured at runtime to support operands at 16-, 8-, and 4-bit. Here are the main contributions:

  A) We provided detailed insights into the working principles, hardware architectures, and HLS design flow that we used to derive the three ST-based accelerators for ASIC.

  B) In contrast to existing PS DNN accelerators, we integrated hardware support for uniform integer quantization (UIQ) [51]. Moreover, in our design flow, we included the minimization of the bitwidths of the fixed-point variables required by the UIQ formulas.

  C) We showcased the Pareto-optimal accelerators resulting from the HLS-driven DSE in Latency vs Area and Latency vs Power spaces. We conducted a rich DSE for each type of ST accelerator thanks to HLS, varying many parameters including: memory sizes, parallelism, clock frequency, and ST multiplier implementation in the accelerators' MAC units.

  D) Lastly, we demonstrated the 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 and energy reduction in the inference of Machine Learning Performance (MLPerf) Tiny models [53] quantized in MP [111] as a case study, along with the area overheads of ST-based accelerators, when comparing against SoCs with equivalent accelerators based on non-ST fixed-precision 16-bit multipliers (named *standard* accelerators).

  - The results of the DSE allow designers to select the best type of ST multiplier in conjunction with the optimal configuration of hardware parameters for a given target in the PPA space.
  - 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 reduction in most of the cases; and a marginal area overhead of 0.9%, 2.5% and 8.0% compared to SoCs equipped with standard accelerators.

  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 solutions to identified inefficiencies.

- **In Chapter 5**, we introduced a novel class of multipliers termed *Sum-Together/Apart Reconfigurable (STAR)*, capable of operating in both SA and ST modes within a single design [27]. This is a novelty for the literature as *Sum-Apart (SA)* and ST multipliers have been proposed as stand-alone implementations until now [29].

  A) We developed four STAR multiplier architectures, accommodating 16, 8 and 4 bits of operand precision. These architectures include variants based on established *Divide-and-Conquer (D&C)* [37, 49] and SWP families [25], as well as innovative designs incorporating a 3-way approach (i.e., three mutually exclusive datapaths) [41, 42] and separate SA and ST multipliers with multiplexed outputs [27].

  B) We compared them in terms of PPA in a 28-nm technology across the [0.4 to 2] GHz clock frequency range to identify the best solutions for different PPA requirements.

  The main findings are:

  - STAR SWP emerges as the most suitable choice for low-power and low-area designs;
  - STAR 3-way excels in high-performance scenarios;
  - STAR D&C presents a competitive option for mid-range PPA requirements.

  Our results identify optimal solutions catering to different design requirements which offer valuable insights for designers seeking to implement efficient multipliers tailored to specific design targets.

- **In Chapter 6**, to support quantized DNNs in low-power extreme-edgeCPUs, we proposed *STAR MAC*, a PS MAC unit based on a modified BW architecture that operates at a variable reduced precision [31].

  A) We integrated it in a small RISC-V processor called *Ibex* [54]. Specifically, we replaced the default 16-bit multiplier inside the Multiplier/-Divider (MULT/DIV) unit of that processor with a 16-bit STAR BW multiplier;

  B) We added new MAC instructions which were not available in the original MULT/DIV unit: standard 32-bit MAC and 16/8/4-bit MAC operations in ST/SA mode;

  C) We compared our new Ibex processor with the original one (*Orig.*) and with a modified version of the latter that supports standard 32-bit MAC operations (*Orig.+MAC*). Comparisons are in terms of area, estimated power, and performance speed up on a set of quantized 2D-Conv, DW-Conv and FC layers.

  The results that we obtained show:

  – An acceleration up to 5.8× in FC layers, 3.7× in 2D-Conv layers, and 2.8× in DW-Conv layers, with respect to *Orig.*, and up to 4.5× in FC layers, 3.0× in 2D-Conv layers, and 2.3× in DW-Conv layers, against *Orig.+MAC*;

  – Values of area and power estimates in a 28-nm technology with 200 and 600 MHz target clock frequency of 0.015 and 0.017 mm^2, and 1.5 and 4.3 mW, respectively, with a limited overhead within 10% and 3% with respect to *Orig.*, and within 6% and 3% against *Orig.+MAC*.

  These results make our proposed STAR MAC a promising solution for enabling inference of MP-quantized DNNs on resource-limited devices.

7.2 Future Work

As future work, we have planned the following activities:

- **SA-based DW-Conv hardware accelerator**: As discussed at the end of Chapter 4, the ST-based DW-Conv accelerator has several inefficiencies. Therefore, we decide to develop a new PS DW-Conv accelerator based on an SA multiplier. This accelerator will allow for the parallel multiplication of low-precision input and weight elements along the channels dimension without summing them together, while maintaining separate multiplication results according to the DW-Conv algorithm.

- **STAR-based 2D/DW-Conv hardware accelerator**: The aim is to create a new PS hardware accelerator that leverages STAR in its MAC unit. As mentioned at the beginning of Chapter 5, we believe that integrating a single engine for 2D/DW-Conv will yield greater efficiency in terms of area and power consumption compared to maintaining two separate accelerators—an ST-based 2D-Conv and an SA-based DW-Conv—due to enhanced resource sharing between the two operating modes.

- **SoC design with PS accelerators and tensor tiling**: In Chapter 4, we conducted experiments with a hypothetical SoC, assuming a global buffer and efficient tensor tiling management by the processor. Moving forward, our plan involves integrating our ST-based accelerators into a more realistic SoC using the ESP tool from Columbia University [119], along with implementing tensor tiling management similar to that described in [103]. As part of our experiments, we aim to evaluate pros and cons of different approaches, such as processor-managed tiling versus accelerator-managed tiling, in terms of hardware resource utilization and inference time. Additionally, we intend to integrate the RISC-V processor discussed in Chapter 6 into the SoC to evaluate trade-offs of executing a quantized DNN layer in software on the PS processor, in hardware using the corresponding PS accelerator, or with an interleaved approach that mixes the first two.

- **Accelerators place and route, and smart DSE for accelerators and SoCs**: The DSE that we conducted in Chapter 4 for ST accelerators was done using a grid-search and manual approach. We believe that a smarter selection of knobs could lead to new Pareto-optimal solutions and enrich the Pareto front. However, the search space could be very large, making manual exploration time-consuming and prone to suboptimal solutions. To address this, we are considering Bayesian Optimization (BO) [120] as a potential candidate, implemented by tools like Spearmint [121]. Furthermore, we plan to extend the DSE beyond logic synthesis to include place and route for more accurate area and power consumption estimates. Similarly, we aim to perform a DSE at the SoC level, exploring knobs that impact not only individual accelerator designs but also the entire SoC architecture. Lastly, in contrast to more traditional BO approaches, we envision integrating neural architecture search (NAS) into the Bayesian Optimization tool to make hardware-software co-design or joint optimization [113]. For instance, we could explore the optimal combination of bit precision along with hardware knobs for accelerators and SoC to achieve an efficient final implementation. Recent research conducted by our group [113] has demonstrated that a unified approach, integrating both software and hardware optimization, yields superior results compared to disjoint approaches where NAS and hardware parameter optimization are conducted separately.

- **Automatic framework for mapping MP-quantized DNNs from QKeras to TFLite Micro targeting the STAR-based Ibex processor**: While the results obtained for our PS processor in Chapter 6 are promising, they only pertain to the execution of individual layers rather than entire DNNs in MP. To evaluate the true benefits of the STAR approach at network level, we are developing an open-source automated framework to execute MP networks with TFLite Micro on our STAR-based Ibex RISC-V core. However, since TFLite does not support quantization below 8 bits, we plan to use the modified version of QKeras described in Sec. 4.2.2 to create MP DNNs with 16, 8, and 4 bits. Subsequently, we will modify the execution kernels (i.e., low-level C/C++ routines) of the 2D-Conv, DW-Conv, and FC layers defined in TFLite Micro to invoke the STAR instructions presented in Chapter 6. Lastly, we will use the framework to convert the MP-quantized DNNs from QKeras to the flatbuffer format of TFLite, enabling the execution of any given MP network on our STAR-based RISC-V processor.

Appendix A

Integer-only DNN kernels for 2D- and DW-Conv

In this appendix, we use the notation of Table 4.1. As mentioned in Sec. 4.1.1, the quantized kernels of 2D- and DW-Conv are derived similarly to FC. Let us start from non-quantized 2D-Conv:

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

$X \in \mathbb{R}^{IH \times IW \times IC}$ 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 and OC the number of input and output channels, and KH and KW the kernel dimensions. To quantize it, we first apply (4.4) to each real variable in (A.1). 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 (A.2) valid for the

(oh, ow, oc)-th output element:

$$
\begin{aligned}
Y_{q,\,oh,\,ow,\,oc} = &\underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,\,oc} - z_b)}_{\text{(b)}} \\
+\frac{s_X s_W}{s_Y}\Bigg[&\underbrace{\left(\sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,ic} \cdot W_{q,\,kh,\,kw,\,ic,\,oc} \right)}_{\text{(c)}} \\
&- \underbrace{\left(z_W \sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,ic} \right)}_{\text{(d)}} \\
&- \underbrace{\left(z_X \sum_{ic=1}^{IC} \sum_{kh=1}^{KH} \sum_{kw=1}^{KW} W_{q,\,kh,\,kw,\,ic,\,oc} \right)}_{\text{(e)}} \\
&+ \underbrace{(IC \cdot KH \cdot KW) \cdot z_X z_W}_{\text{(f)}} \Bigg] \\
&\forall\, oh \in [1, OH], ow \in [1, OW], oc \in [1, OC]
\end{aligned}
\tag{A.2}
$$

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 (A.2) 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. 4.1.1. Compared to (4.6), 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 (A.3) and (A.4), respectively. For (A.3), 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. (A.4) is derived from (A.3) with the same steps used for FC and 2D-Conv. It differs from (A.2) in the number of summations (two

instead of three) and in the absence of the IC constant in term (f).

$$Y_{h,\,w,\,oc} = b_{oc} + \left(\sum_{kh\,=\,1}^{KH} \sum_{kw\,=\,1}^{KW} X_{oh+kh,\,ow+kw,\,oc} \cdot W_{kh,\,kw,\,oc} \right) \qquad \forall\, oh \in [1, OH], ow \in [1, OW], oc \in [1, OC] \tag{A.3}$$

$$\begin{aligned} Y_{q,\,oh,\,ow,\,oc} = {} & \underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,\,oc} - z_b)}_{\text{(b)}} \\ & + \frac{s_X s_W}{s_Y} \Bigg[\underbrace{\left(\sum_{kh=1}^{KH} \sum_{kw=1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,oc} \cdot W_{q,\,kh,\,kw,\,oc} \right)}_{\text{(c)}} \\ & - \underbrace{\left(z_W \sum_{kh\,=\,1}^{KH} \sum_{kw\,=\,1}^{KW} X_{q,\,oh+kh,\,ow+kw,\,oc} \right)}_{\text{(d)}} \\ & - \underbrace{\left(z_X \sum_{kh\,=\,1}^{KH} \sum_{kw\,=\,1}^{KW} W_{q,\,kh,\,kw,\,oc} \right)}_{\text{(e)}} \\ & + \underbrace{KH \cdot KW \cdot z_X z_W}_{\text{(f)}} \Bigg] \\ & \forall\, oh \in [1, OH], ow \in [1, OW], oc \in [1, OC] \end{aligned} \tag{A.4}$$

As discussed in Sec. 4.1.1 for (4.6), in this thesis we assume that $z_W = 0$ and $z_b = 0$ for (A.2) and (A.4) as well.

Appendix B

Mixed-precision results of MLPerf Tiny models

Tables B.1–B.2 report the architecture of the MP-quantized MLPerf Tiny models obtained in the first step of our accelerators design-flow of Sec. 4.3.1. The last two columns show the number of bits for activation/weight and bias, respectively.

Table B.1: 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

Continued on next page

Table B.1: MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization) (continued).

#	QKeras Layer	Output Shape	Activation / Weight Bits (INT)	Bias Bits (INT)
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

Continued on next page

Table B.1: MP-quantized model of MobileNetV1Tiny (using QKeras' syntax and with the new QActivation layer implementing affine uniform quantization) (continued).

#	QKeras Layer	Output Shape	Activation / Weight Bits (INT)	Bias Bits (INT)
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 B.2: 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	

Table B.3: 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 B.4: 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		

Appendix C

Published Papers and Awards

Journal papers (related to this thesis):

- **Urbinati, Luca** and Casu, Mario R., "High-Level Design of Precision-Scalable DNN Accelerators Based on Sum-Together Multipliers," IEEE Access, vol. 12, pp. 44163-44189, 2024.
  doi: 10.1109/ACCESS.2024.3380472.

Conference papers (related to this thesis):

- Manca, Edward, **Urbinati, Luca** and Casu, Mario R., "STAR: Sum-Together/Apart Reconfigurable Multipliers for Precision-Scalable ML Workloads," Accepted for publication in Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE), Valencia, Spain: IEEE, 2024.
- Manca, Edward, **Urbinati, Luca** and Casu, Mario R., "Accelerating Quantized DNN Layers on RISC-V with a STAR MAC Unit," in Proc. of SIE 2023, in Lecture Notes in Electrical Engineering, vol. 1113: Springer Nature Switzerland, pp. 43–53, 2024.
  doi: 10.1007/978-3-031-48711-8_6.
- **Urbinati, Luca** and Casu, Mario R., "Design-Space Exploration of Mixed-precision DNN Accelerators based on Sum-Together Multipliers," in Proc. of 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Valencia, Spain: IEEE, pp. 377–380, 2023.
  doi: 10.1109/PRIME58259.2023.10161835.
- **Urbinati, Luca** and Casu, Mario R., "A Reconfigurable Multiplier/Dot-Product Unit for Precision-Scalable Deep Learning Applications," in Proc. of SIE 2022, Pizzo, Italy, in Lecture Notes in Electrical Engineering, vol. 1005: Springer Nature Switzerland, pp. 9–14, 2023.
  doi: 10.1007/978-3-031-26066-7_2.
- **Urbinati, Luca** and Casu, Mario R., "A Reconfigurable 2D-Convolution Accelerator for DNNs Quantized with Mixed-Precision," in Proc. of Applications in Electronics Pervading Industry, Environment and Society (ApplePies), Genoa, Italy, in Lecture Notes in Electrical Engineering, vol. 1036: Springer Nature Switzerland, pp. 210–215, 2023.
  doi: 10.1007/978-3-031-30333-3_27.

- **Urbinati, Luca** and Casu, Mario R., "A Reconfigurable Depth-Wise Convolution Module for Heterogeneously Quantized DNNs," in Proc. of Int. Symp. on Circuits and Systems (ISCAS), Austin, TX, USA: IEEE, pp. 128–132, 2022.
  doi: 10.1109/ISCAS48785.2022.9937753.

Awards (related to this thesis):

- Gold Leaf Certificate for our work titled "Design-Space Exploration of Mixed-precision DNN Accelerators based on Sum-Together Multipliers" [24], recognized as one of the top 10% papers presented at the 18th International Conference on PhD Research in Microelectronics and Electronics (PRIME), held in Valencia, Spain, from June 18th to 21st, 2023.

Supervised M.Sc. Theses:

- Bueno Pacheco, Diego R, "Efficient Tiling Architecture for Scalable CNN Inference: Leveraging High-Level Design and Embedded Scalable Platform (ESP)," Master's Thesis. Supervisors: Mario R. Casu and Luca Urbinati. Politecnico di Torino, 2023.
  `https://webthesis.biblio.polito.it/29513/`.
- Manca, Edward, "Design of a Novel Precision Scalable Multiplier to Improve Quantized Neural Network Computation on a Low-Power RISC-V Processor," Master's Thesis. Supervisors: Mario R. Casu and Luca Urbinati. Politecnico di Torino, 2023.
  `https://webthesis.biblio.polito.it/27724/`.
- Terlizzi, Marco A, "Mixed-precision Quantization and Inference of MLPerf Tiny DNNs on Precision-Scalable Hardware Accelerators," Master's Thesis. Supervisors: Mario R. Casu and Luca Urbinati. Politecnico di Torino, 2023.
  `https://webthesis.biblio.polito.it/26664/`.
- Perenno, Federico, "High-Level Design of 2D-Convolution Accelerators for AI Leveraging Embedded Scalable Platform (ESP)," Master's Thesis. Supervisors: Mario R. Casu and Luca Urbinati. Politecnico di Torino, 2022.
  `https://webthesis.biblio.polito.it/25415/`.
- Capodicasa, Riccardo, "High-level design of a Depthwise Convolution accelerator and SoC integration using ESP," Master's Thesis. Supervisors: Mario R. Casu and Luca Urbinati. Politecnico di Torino, 2022.
  `https://webthesis.biblio.polito.it/25410/`.

Acronyms

2D-Conv 2D-Convolution. 3, 18, 114

AD Anomaly Detection. 46

ALU Arithmetic Logic Unit. 27

ASIC Application-Specific Integrated Circuit. 4, 24, 113

AUC Area Under The Receiver Operating Characteristics Curve. 48

BN Batch Normalization. 45

BO Bayesian Optimization. 117

BSC bit-split-and-combination. 21

BW Baugh-Wooley. 3, 18, 113

CCORE Catapult C Optimized Reusable Entity. 72

CIFAR-10 Canadian Institute for Advanced Research, 10 classes. 47

D&C Divide-and-Conquer. 4, 19, 115

DC Design Compiler. 40

DCASE Detection and Classification of Acoustic Scenes and Events. 48

DL Deep Learning. 3, 15, 114

DMA Direct Memory Access. 58

DNNs Deep Neural Networks. 3, 15, 113

DS-CNN Depth-wise Separable Convolutional Neural Network. 48

DSE design space exploration. 4, 18, 113

DSP digital signal processing. 21

DW Depth-wise. 19

DW-Conv Depth-wise Convolution. 3, 18, 114

FA Full Adder. 87

FC Fully-Connected. 3, 18, 114

FIFO First In First Out. 96

FP Floating-Point. 24

FPGA Field Programmable Gate Array. 24

FSM finite state machine. 98

FUs Fusion Units. 21

GEMM General Matrix Multiply. 23

HDR high dynamic range. 29

HLS High-Level Synthesis. 3, 18, 113

HPC High-Performance Computing. 25

IA input tile activations. 53

IBUF internal input buffer. 54

IC input tile channels. 53

ICC iteration clock cycle. 98

IDE Instruction Decode and Execute. 96

IF Instruction Fetch. 96

ImgClass Image Classification. 46

IoT Internet-of-Things. 25

ISA Instruction Set Architecture. 21

KS Keyword Spotting. 46

LA Latency vs Area. 73

LP Latency vs Power. 76

LSB least-significant bit. 17

LUTs look-up tables. 23

MAC Multiply-and-Accumulate. 3, 17, 113

MCUs microprocessors. 16

MFCC Mel-Frequency Cepstral Coefficients. 47

ML Machine Learning. 16

MLPerf Machine Learning Performance. 4, 46, 114

MP mixed-precision. 4, 15, 113

MPQ Mixed-Precision Quantization. 3, 15

MSCOCO Microsoft Common Objects in Context. 47

MULT/DIV Multiplier/Divider. 96, 116

NAS neural architecture search. 117

OA output tile activations. 53

OBUF internal output buffer. 54

OC output tile channels. 53

Perf test set performance evaluation dataset. 46

PEs processing elements. 27

PPA performance, power, and area. 3, 18, 113

PPM partial product matrix. 31

PPs partial products. 32

PS Precision-Scalable. 3, 15, 113

PSMAC Precision-Scalable Multiply-and-Accumulate. 16

RCA Ripple Carry Adder. 18, 113

ReLU rectified linear unit. 45

RTL Register-Transfer Level. 18

SA Sum-Apart. 3, 16, 115

SIMD Single Instruction Multiple Data. 3, 17

SoA State-of-the-Art. 3, 18, 113

SoC System-on-Chip. 18, 114

ST Sum-Together. 3, 16, 113

STAR Sum-Together/Apart Reconfigurable. 4, 19, 115

SWP Sub-word Parallel. 4, 19, 114

TC Transprecision Computing. 3, 15, 113

TFLite TensorFlow Lite. 43

TFLM TensorFlow Lite Micro. 30

UIQ uniform integer quantization. 18, 114

VWW Visual Wake Words. 46

WBUF internal weight buffer. 54

Bibliography

[1] Amir Gholami et al. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence*. 1st. New York, NY, USA: Chapman and Hall/CRC, 2022. Chap. 1.2.12, pp. 14–17. DOI: 10.1201/9781003162810.

[2] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[3] George K. Thiruvathukal et al. "Low-Power Computer Vision". In: Chapman and Hall/CRC, 2022. Chap. Improve the Efficiency of Artificial Intelligence. DOI: 10.1201/9781003162810.

[4] Mariam Rakka et al. "Mixed-Precision Neural Networks: A Survey". In: *arXiv* (2022). 2208.06064.

[5] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Fixed point optimization of deep convolutional neural networks for object recognition". In: *Proc. Int. Conf. Acoustics, Speech and Signal Processing (ICASSP)*. South Brisbane, QLD, Australia: IEEE, 2015, pp. 1131–1135. DOI: 10.1109/ICASSP.2015.7178146.

[6] Bert Moons et al. "Energy-efficient ConvNets through approximate computing". In: *Proc. Winter Conf. on Applications of Computer Vision (WACV)*. Lake Placid, NY, USA: IEEE, 2016, pp. 1–8. DOI: 10.1109/WACV.2016.7477614.

[7] Tim Hotfilter et al. "Leveraging Mixed-Precision CNN Inference for Increased Robustness and Energy Efficiency". In: *Proc. 36th Int. System-on-Chip Conference (SOCC)*. Santa Clara, CA, USA: IEEE, 2023, pp. 1–6. DOI: 10.1109/SOCC58585.2023.10256738.

[8] Junnosuke Suzuki et al. "Pianissimo: A Sub-mW Class DNN Accelerator With Progressively Adjustable Bit-Precision". In: *IEEE Access* 12 (2024), pp. 2057–2073. DOI: 10.1109/ACCESS.2023.3347578.

[9] Karina Vasquez et al. "Activation Density based Mixed-Precision Quantization for Energy Efficient Neural Networks". In: *arXiv* (2021). 2101.04354.

[10] Cristiano A. I. Malossi et al. “The transprecision computing paradigm: Concept, design, and applications”. In: *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden, Germany: IEEE, 2018, pp. 1105–1110. DOI: 10.23919/DATE.2018.8342176.

[11] Zhen Dong et al. “HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks”. In: *arXiv* (2019). 1911.03852.

[12] Qian Lou et al. “AutoQ: Automated Kernel-Wise Neural Network Quantization”. In: *arXiv* (2020). 1902.05690.

[13] Ahmed T. Elthakeb et al. “ReLeQ : A Reinforcement Learning Approach for Automatic Deep Quantization of Neural Networks”. In: *IEEE Micro* 40.5 (2020), pp. 37–45. DOI: 10.1109/MM.2020.3009475.

[14] Claudionor N. Coelho et al. “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors”. In: *Nature Machine Intelligence* 3.8 (2021), pp. 675–686. DOI: 10.1038/s42256-021-00356-5.

[15] Qigong Sun et al. “Effective and Fast: A Novel Sequential Single Path Search for Mixed-Precision Quantization”. In: *arXiv* (2021). 2103.02904.

[16] Zhang Zhaoyang et al. “Differentiable Dynamic Quantization with Mixed Precision and Adaptive Resolution”. In: *arXiv* (2021). 2106.02295.

[17] Zhi-Gang Liu and Matthew Mattina. “Learning low-precision neural networks without Straight-Through Estimator(STE)”. In: *arXiv* (2019). 1903.01061.

[18] Kuan Wang et al. “HAQ: Hardware-Aware Automated Quantization With Mixed Precision”. In: *Proc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, 2019, pp. 8604–8612. DOI: 10.1109/CVPR.2019.00881.

[19] Bichen Wu et al. “Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search”. In: *arXiv* (2018). 1812.00090.

[20] Jungwook Choi et al. “Accurate and Efficient 2-bit Quantized Neural Networks”. In: *Proc. of Machine Learning and Systems*. Vol. 1. Stanford, California, 2019, pp. 348–359.

[21] Stefan Uhlich et al. “Mixed Precision DNNs: All you need is a good parametrization”. In: *arXiv* (2020). 1905.11452.

[22] Shuchang Zhou et al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *arXiv* (2018). 1606.06160.

[23] Marco Alessio Terlizzi. "Mixed-precision Quantization and Inference of MLPerf Tiny DNNs on Precision-Scalable Hardware Accelerators". MA thesis. Turin, Italy: Politecnico di Torino, 2023. URL: https://webthesis.biblio.polito.it/26664/.

[24] Luca Urbinati and Mario R. Casu. "Design-Space Exploration of Mixed-precision DNN Accelerators based on Sum-Together Multipliers". In: *Proc. 18th Conf. on Ph.D Research in Microelectronics and Electronics (PRIME)*. Valencia, Spain: IEEE, 2023, pp. 377–380. DOI: 10.1109/PRIME58259.2023.10161835.

[25] Linyan Mei et al. "Sub-Word Parallel Precision-Scalable MAC Engines for Efficient Embedded DNN Inference". In: *Proc. Int. Conf. on Artificial Intelligence Circuits and Systems (AICAS)*. Hsinchu, Taiwan: IEEE, 2019, pp. 6–10. DOI: 10.1109/AICAS.2019.8771481.

[26] Luca Urbinati and Mario R. Casu. "A Reconfigurable Multiplier/Dot-Product Unit for Precision-Scalable Deep Learning Applications". In: *Proc. of SIE 2022*. Pizzo, Italy: Springer Nature Switzerland, 2023, pp. 9–14.

[27] Edward Manca, Luca Urbinati, and Mario R. Casu. "STAR: Sum-Together/Apart Reconfigurable Multipliers for Precision-Scalable ML Workloads". In: *Accepted for publication in Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Valencia, Spain: IEEE, 2024.

[28] Luca Urbinati and Mario R. Casu. "High-Level Design of Precision-Scalable DNN Accelerators Based on Sum-Together Multipliers". In: *IEEE Access* 12 (2024), pp. 44163–44189. DOI: 10.1109/ACCESS.2024.3380472.

[29] Vincent Camus et al. "Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing". In: *IEEE Trans. Emerg. Sel. Topics Circuits Syst. (JETCAS)* 9.4 (2019), pp. 697–711. DOI: 10.1109/JETCAS.2019.2950386.

[30] Wenjie Li et al. "Low-Complexity Precision-Scalable Multiply-Accumulate Unit Architectures for Deep Neural Network Accelerators". In: *IEEE Trans. Circuits Syst. II Express Briefs (TCAS-II)* 70.4 (2023), pp. 1610–1614. DOI: 10.1109/TCSII.2022.3231418.

[31] Edward Manca, Luca Urbinati, and Mario R. Casu. "Accelerating Quantized DNN Layers on RISC-V with a STAR MAC Unit". In: *Proc. of SIE 2023*. Noto, Italy: Springer Nature Switzerland, 2024, pp. 43–53. DOI: 10.1007/978-3-031-48711-8_6.

[32] Wenjian Liu, Jun Lin, and Zhongfeng Wang. "A Precision-Scalable Energy-Efficient Convolutional Neural Network Accelerator". In: *IEEE Trans. Circuits Syst. I Regul. Pap. (TCAS-I)* 67.10 (2020), pp. 3484–3497. DOI: 10.1109/TCSI.2020.2993051.

[33] Wei Mao et al. “An Energy-Efficient Mixed-Bitwidth Systolic Accelerator for NAS-Optimized Deep Neural Networks”. In: *IEEE Trans. VLSI Syst.* 30.12 (2022), pp. 1878–1890. DOI: 10.1109/TVLSI.2022.3210069.

[34] Sungju Ryu et al. “BitBlade: Energy-Efficient Variable Bit-Precision Hardware Accelerator for Quantized Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 57.6 (2022), pp. 1924–1935. DOI: 10.1109/JSSC.2022.3141050.

[35] Bert Moons et al. “DVAFS: Trading computational accuracy for energy through dynamic-voltage-accuracy-frequency-scaling”. In: *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* Lausanne, Switzerland: IEEE, 2017, pp. 488–493. DOI: 10.23919/DATE.2017.7927038.

[36] Sayeh Sharify et al. “Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks”. In: *Proc. 55th Design Automation Conference (DAC).* San Francisco, CA, USA: IEEE, 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465915.

[37] Hardik Sharma et al. “Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network”. In: *Proc. 45th Int. Symp. on Computer Architecture (ISCA).* Los Angeles, CA, USA: IEEE, 2018, pp. 764–775. DOI: 10.1109/ISCA.2018.00069.

[38] Yvan Tortorella et al. “RedMule: A Mixed-Precision Matrix-Matrix Operation Engine for Flexible and Energy-Efficient On-Chip Linear Algebra and TinyML Training Acceleration”. In: *arXiv* (2023). 2301.03904.

[39] Luca Urbinati and Mario R. Casu. “A Reconfigurable 2D-Convolution Accelerator for DNNs Quantized with Mixed-Precision”. In: *Proc. Applications in Electronics Pervading Industry, Environment and Society (ApplePies).* Genoa, Italy: Springer Nature Switzerland, 2023, pp. 210–215. DOI: 10.1007/978-3-031-30333-3_27.

[40] Luca Urbinati and Mario R. Casu. “A Reconfigurable Depth-Wise Convolution Module for Heterogeneously Quantized DNNs”. In: *Proc. Int. Symp. on Circuits and Systems (ISCAS).* Austin, TX, USA: IEEE, 2022, pp. 128–132. DOI: 10.1109/ISCAS48785.2022.9937753.

[41] Michael Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices”. In: *IEEE Trans. VLSI Syst.* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[42] Angelo Garofalo et al. “XpulpNN: Enabling Energy Efficient and Flexible Inference of Quantized Neural Networks on RISC-V Based IoT End Nodes”. In: *IEEE Trans. Emerg. Topics Comput.* 9.3 (2021), pp. 1489–1505. DOI: 10.1109/TETC.2021.3072337.

[43] Guillaume Devic et al. "Highly-Adaptive Mixed-Precision MAC Unit for Smart and Low-Power Edge Computing". In: *Proc. 19th Int. New Circuits and Systems Conference (NEWCAS)*. Toulon, France: IEEE, 2021, pp. 1–4. DOI: 10.1109/NEWCAS50681.2021.9462745.

[44] Longwei Huang et al. "A Precision-Scalable RISC-V DNN Processor with On-Device Learning Capability at the Extreme Edge". In: *arXiv* (2023). 2309.08186.

[45] Risikesh RK, Sharad Sinha, and Nanditha Rao. "Variable Bit-Precision Vector Extension for RISC-V Based Processors". In: *Proc. 14th Int. Symp. on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Singapore: IEEE, 2021, pp. 114–121. DOI: 10.1109/MCSoC51149.2021.00024.

[46] Dongjoo Shin et al. "14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks". In: *Proc. Int. Solid-State Circuits Conference (ISSCC)*. San Francisco, CA, USA: IEEE, 2017, pp. 240–241. DOI: 10.1109/ISSCC.2017.7870350.

[47] Zicheng He et al. "Agile Hardware and Software Co-Design for RISC-V-Based Multi-Precision Deep Learning Microprocessor". In: *Proc. 28th Asia and South Pacific Design Automation Conference (ASPDAC)*. Tokyo, Japan: ACM, 2023, pp. 490–495. DOI: 10.1145/3566097.3567871.

[48] Xinyue Zhang, Zhaolin Li, and Qingwei Zheng. "Design of a configurable fixed-point multiplier for digital signal processor". In: *Proc. Asia Pacific Conf. on Postgraduate Research in Microelectronics & Electronics (PrimeAsia)*. Shanghai, China: IEEE, 2009, pp. 217–220. DOI: 10.1109/PRIMEASIA.2009.5397407.

[49] Rong Lin. "Reconfigurable parallel inner product processor architectures". In: *IEEE Trans. VLSI Syst.* 9.2 (2001), pp. 261–272. DOI: 10.1109/92.924037.

[50] Neil Weste and David Money Harris. "CMOS VLSI Design". In: 4th. Reading, MA, USA: Addison-Wesley, 2011.

[51] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proc. Conf. on Computer Vision and Pattern Recognition (CVPR)*. Salt Lake City, UT, USA, 2018, pp. 2704–2713.

[52] Hao Wu et al. "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation". In: *arXiv* (2020). 2004.09602.

[53] Colby R. Banbury et al. "MLPerf Tiny Benchmark". In: *arXiv* (2021). 2106.07597.

[54] Pasquale Davide Schiavone et al. "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. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Thessaloniki, Greece: IEEE, 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[55] Vincent Camus, Christian Enz, and Marian Verhelst. "Survey of Precision-Scalable Multiply-Accumulate Units for Neural-Network Processing". In: *Proc. Int. Conf. on Artificial Intelligence Circuits and Systems (AICAS)*. Hsinchu, Taiwan: IEEE, 2019, pp. 57–61. DOI: 10.1109/AICAS.2019.8771610.

[56] Jinmook Lee et al. "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision". In: *IEEE J. Solid-State Circuits (JSSC)* 54.1 (2019), pp. 173–185. DOI: 10.1109/JSSC.2018.2865489.

[57] Kh Shahriya Zaman et al. "Custom Hardware Architectures for Deep Learning on Portable Devices: A Review". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.11 (2022), pp. 6068–6088. DOI: 10.1109/TNNLS.2021.3082304.

[58] Yunji Chen et al. "DaDianNao: A Machine-Learning Supercomputer". In: *Proc. 47th Int. Symp. on Microarchitecture*. Cambridge, UK: ACM/IEEE, 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[59] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *Proc. 43rd Int. Symp. on Computer Architecture (ISCA)*. Seoul, Korea (South): ACM/IEEE, 2016, pp. 243–254. DOI: 10.1109/ISCA.2016.30.

[60] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *Proc. 43rd Int. Symp. on Computer Architecture (ISCA)*. Seoul, Korea (South), 2016, pp. 367–379. DOI: 10.1109/ISCA.2016.40.

[61] Enrico Reggiani et al. "Mix-GEMM: An efficient HW-SW Architecture for Mixed-Precision Quantized Deep Neural Networks Inference on Edge Devices". In: *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*. Montreal, QC, Canada: IEEE, 2023, pp. 1085–1098. DOI: 10.1109/HPCA56546.2023.10071076.

[62] Victor Pan. "How to Multiply Matrices Faster". In: ed. by Victor Pan. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984. ISBN: 978-3-540-39058-9. DOI: 10.1007/3-540-13866-8_9.

[63] Amine Bermak and Dominique Martinez. “A compact multi-chip-module implementation of a multi-precision neural network classifier”. In: *Proc. Int. Symp. on Circuits and Systems (ISCAS)*. Vol. 3. Sydney, NSW, Australia: IEEE, 2001, 249–252 vol. 2. DOI: 10.1109/ISCAS.2001.921294.

[64] Arpan Suravi Prasad et al. “Siracusa: A 16 nm Heterogenous RISC-V SoC for Extended Reality With At-MRAM Neural Engine”. In: *IEEE Journal of Solid-State Circuits* (2024), pp. 1–15. DOI: 10.1109/JSSC.2024.3385987.

[65] Francesco Conti et al. “Marsellus: A Heterogeneous RISC-V AI-IoT End-Node SoC With 2–8 b DNN Acceleration and 30%-Boost Adaptive Body Biasing”. In: *IEEE J. Solid-State Circuits (JSSC)* 59.1 (2024), pp. 128–142. DOI: 10.1109/JSSC.2023.3318301.

[66] Biagio Peccerillo et al. “A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives”. In: *Journal of Systems Architecture* 129 (2022), p. 102561. DOI: 10.1016/j.sysarc.2022.102561.

[67] Christoffer Åleskog, Håkan Grahn, and Anton Borg. “Recent Developments in Low-Power AI Accelerators: A Survey”. In: *Algorithms* 15.11 (2022). DOI: 10.3390/a15110419.

[68] Albert Reuther et al. “AI and ML Accelerator Survey and Trends”. In: *Proc. High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2022, pp. 1–10. DOI: 10.1109/HPEC55821.2022.9926331.

[69] Albert Reuther et al. “Lincoln AI Computing Survey (LAICS) Update”. In: *arXiv* (2023). 2310.09145.

[70] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. “BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing”. In: *Proc. 28th Int. Conf. on Field Programmable Logic and Applications (FPL)*. Dublin, Ireland: IEEE, 2018, pp. 307–3077. DOI: 10.1109/FPL.2018.00059.

[71] Mohammad Hossein Askari Hemmat et al. “Quark: An Integer RISC-V Vector Processor for Sub-Byte Quantized DNN Inference”. In: *Proc. Int. Symp. on Circuits and Systems (ISCAS)*. Monterey, CA, USA: IEEE, 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181985.

[72] Md. Maruf Hossain Shuvo et al. “Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review”. In: *Proceedings of the IEEE* 111.1 (2023), pp. 42–91. DOI: 10.1109/JPROC.2022.3226481.

[73] Gianmarco Ottavi et al. “A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference”. In: *Proc. Computer Society Annual Symposium on VLSI (ISVLSI)*. Limassol, Cyprus: IEEE, 2020, pp. 512–517. DOI: 10.1109/ISVLSI49217.2020.000-5.

[74] Théo Dupuis et al. "Sparq: A Custom RISC-V Vector Processor for Efficient Sub-Byte Quantized Inference". In: *Proc. Northeast Workshop on Circuits and Systems (NEWCAS)*. Edinburgh, United Kingdom: IEEE, 2023, pp. 1–5. DOI: 10.1109/NEWCAS57931.2023.10198172.

[75] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.

[76] Matheus Cavalcante et al. "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI". In: *IEEE Trans. VLSI Syst.* 28.2 (2020), pp. 530–543. DOI: 10.1109/TVLSI.2019.2950087.

[77] Víctor Soria-Pardos et al. "Sargantana: A 1 GHz+ In-Order RISC-V Processor with SIMD Vector Extensions in 22nm FD-SOI". In: *Proc. 25th Euromicro Conference on Digital System Design (DSD)*. Maspalomas, Spain: IEEE, 2022, pp. 254–261. DOI: 10.1109/DSD57027.2022.00042.

[78] Chuanning Wang et al. "A Scalable RISC-V Vector Processor Enabling Efficient Multi-Precision DNN Inference". In: *arXiv* (2024). 2401.16872.

[79] Eric Flamand et al. "GAP-8: A RISC-V SoC for AI at the Edge of the IoT". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Milan, Italy: IEEE, 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445101.

[80] Francesco Conti et al. "PULP: A Ultra-Low Power Parallel Accelerator for Energy-Efficient and Flexible Embedded Vision". In: *Journal of Signal Processing Systems* 84.3 (2016), pp. 339–354. DOI: 10.1007/s11265-015-1070-9.

[81] Davide Nadalini et al. "Reduced Precision Floating-Point Optimization for Deep Neural Network On-Device Learning on MicroControllers". In: *arXiv* (2023). 2305.19167.

[82] Donghyeon Han et al. "Energy-Efficient DNN Training Processors on Micro-AI Systems". In: *IEEE Open Journal of the Solid-State Circuits Society* 2 (2022), pp. 259–275. DOI: 10.1109/OJSSCS.2022.3219034.

[83] Ivan Miro-Panades et al. "SamurAI: A 1.7MOPS-36GOPS Adaptive Versatile IoT Node with 15,000× Peak-to-Idle Power Reduction, 207ns Wake-Up Time and 1.3TOPS/W ML Efficiency". In: *Proc. Symposium on VLSI Circuits*. Honolulu, HI, USA: IEEE, 2020, pp. 1–2. DOI: 10.1109/VLSICircuits18222.2020.9163000.

[84] Davide Rossi et al. "Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode". In: *IEEE Journal of Solid-State Circuits* 57.1 (2022), pp. 127–139. DOI: 10.1109/JSSC.2021.3114881.

[85] Gianmarco Ottavi et al. "Dustin: A 16-Cores Parallel Ultra-Low-Power Cluster With 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode". In: *IEEE Trans. Circuits Syst. I Regul. Pap. (TCAS-I)* 70.6 (2023), pp. 2450–2463. DOI: 10.1109/TCSI.2023.3254810.

[86] Angelo Garofalo et al. "Darkside: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training". In: *IEEE Open Journal of the Solid-State Circuits Society* 2 (2022), pp. 231–243. DOI: 10.1109/OJSSCS.2022.3210082.

[87] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.

[88] Partha Pratim Ray. "A review on TinyML: State-of-the-art and prospects". In: *Journal of King Saud University - Computer and Information Sciences* 34.4 (2022), pp. 1595–1623. DOI: 10.1016/j.jksuci.2021.11.019.

[89] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv* (2018). 1806.08342.

[90] Xiao Sun et al. "Ultra-Low Precision 4-bit Training of Deep Neural Networks". In: *Proc. Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Online: Curran Associates, Inc., 2020, pp. 1796–1807.

[91] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *Proc. Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*. San Francisco, CA, USA: IEEE, 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[92] Alessandro Pappalardo et al. "QONNX: Representing Arbitrary-Precision Quantized Neural Networks". In: *arXiv* (2022). 2206.07527.

[93] Aakanksha Chowdhery et al. "Visual Wake Words Dataset". In: *arXiv* (2019). 1906.05721.

[94] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *Proc. 13th European Conference on Computer Vision (ECCV)*. Zurich, Switzerland: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1.

[95] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *arXiv* (2017). 1704.04861.

[96] Alex Krizhevsky. *Learning multiple layers of features from tiny images.* Tech. rep. Toronto, ON, Canada: University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[97] K He et al. "Deep Residual Learning for Image Recognition". In: *Proc. Conf. on Computer Vision and Pattern Recognition (CVPR).* Las Vegas, NV, USA, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[98] Pete Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *arXiv* (2018). 1804.03209.

[99] Yundong Zhang et al. "Hello Edge: Keyword Spotting on Microcontrollers". In: *arXiv* (2017). 1711.07128.

[100] Yuma Koizumi et al. "Description and Discussion on DCASE2020 Challenge Task2: Unsupervised Anomalous Sound Detection for Machine Condition Monitoring". In: *arXiv* (2020). 2006.05822.

[101] Yuma Koizumi et al. "ToyADMOS: A Dataset of Miniature-Machine Operating Sounds for Anomalous Sound Detection". In: *Proc. Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA).* New Paltz, NY, USA: IEEE, 2019, pp. 313–317. DOI: 10.1109/WASPAA.2019.8937164.

[102] Li Zhang et al. "A fine-grained mixed precision DNN accelerator using a two-stage big–little core RISC-V MCU". In: *Integration* 88 (2023), pp. 241–248. DOI: https://doi.org/10.1016/j.vlsi.2022.10.006.

[103] Alessio Burrello et al. "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs". In: *IEEE Trans. Comput.* 70.8 (2021), pp. 1253–1268. DOI: 10.1109/TC.2021.3066883.

[104] TensorFlow. *TensorFlow Hub.* https://tfhub.dev. Accessed on: Mar 9, 2024.

[105] TensorFlow. *TensorFlow Lite example apps.* https://www.tensorflow.org/lite/examples. Accessed on: Mar 9, 2024.

[106] Intel. *Neural Compute Application Zoo.* https://movidius.github.io/ncappzoo/. Accessed on: Mar 9, 2024.

[107] Intel. *OpenVINO Model Zoo.* https://docs.openvino.ai/2023.2/model_zoo.html. Accessed on: Mar 9, 2024.

[108] AMD Xilinx. *Vitis AI Model Zoo.* https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Vitis-AI-Model-Zoo. Accessed on: Mar 9, 2024.

[109] NVIDIA. *NVIDIA NGC Catalog.* https://catalog.ngc.nvidia.com/models. Accessed on: Mar 9, 2024.

[110] NVIDIA. *Jetson Model Zoo.* https://elinux.org/Jetson_Zoo#Model_Zoo. Accessed on: Mar 9, 2024.

[111] Luca Urbinati. *qkeras-mod*. https://github.com/LucaUrbinati44/qkeras-mod.git. Accessed on: Mar 9, 2024.

[112] RISC-V. *MLPerf Tiny benchmark GitHub repository*. https://github.com/mlcommons/tiny/tree/master/benchmark. Accessed on: Mar 9, 2024.

[113] Mohammad Amir Mansoori and Mario R. Casu. "Multi-objective Framework for Training and Hardware Co-optimization in FPGAs". In: *Proc. Applications in Electronics Pervading Industry, Environment and Society (ApplePies)*. Genoa, Italy: Springer Nature Switzerland, 2023, pp. 273–278. ISBN: 978-3-031-30333-3.

[114] Mentor Graphics. "Catapult Synthesis User and Reference Manual". In: (2018).

[115] Kasem Khalil et al. "Designing Novel AAD Pooling in Hardware for a Convolutional Neural Network Accelerator". In: *IEEE Trans. VLSI Syst.* 30.3 (2022), pp. 303–314. DOI: 10.1109/TVLSI.2021.3139904.

[116] RISC-V. *Releases - Riscv/Riscv-Bitmanip*. https://github.com/riscv/riscv-bitmanip/releases/. Accessed on: Mar 9, 2024.

[117] Edward Manca. "Design of a Novel Precision Scalable Multiplier to Improve Quantized Neural Network Computation on a Low-Power RISC-V Processor". MA thesis. Turin, Italy: Politecnico di Torino, 2023. URL: https://webthesis.biblio.polito.it/27724/.

[118] ETH Zurich and University of Bologna. *Ibex: An embedded 32 bit RISC-V CPU core*. https://ibex-core.readthedocs.io/en/latest/. Accessed on: Mar 9, 2024.

[119] Davide Giri et al. "Accelerator Integration for Open-Source SoC Design". In: *IEEE Micro* 41.4 (2021), pp. 8–14. DOI: 10.1109/MM.2021.3073893.

[120] Stewart Greenhill et al. "Bayesian Optimization for Adaptive Experimental Design: A Review". In: *IEEE Access* 8 (2020), pp. 13937–13948. DOI: 10.1109/ACCESS.2020.2966228.

[121] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. "Predictive Entropy Search for Multi-objective Bayesian Optimization with Constraints". In: *Neurocomputing* 361 (2019), pp. 50–68. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.06.025.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.