

MARLIN: A Co-Design Methodology for Approximate Reconfigurable Inference of Neural Networks at the Edge

Original

MARLIN: A Co-Design Methodology for Approximate Reconfigurable Inference of Neural Networks at the Edge / Guella, Flavia; Valpreda, Emanuele; Caon, Michele; Masera, Guido; Martina, Maurizio. - In: IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. I, REGULAR PAPERS. - ISSN 1549-8328. - ELETTRONICO. - 71:5(2024).
[10.1109/tcsi.2024.3365952]

Availability:

This version is available at: 11583/2986451 since: 2024-02-29T11:26:20Z

Publisher:

IEEE

Published

DOI:10.1109/tcsi.2024.3365952

Terms of use:

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

Publisher copyright

(Article begins on next page)

MARLIN: a Co-design Methodology for Approximate Reconfigurable Inference of Neural Networks at the Edge

Flavia Guella¹ *Student member, IEEE*, Emanuele Valpreda¹ *Student member, IEEE*, Michele Caon¹ *Student member, IEEE*, Guido Masera¹ *Senior member, IEEE*, Maurizio Martina¹ *Senior member, IEEE*

Abstract—The optimization of neural networks (NNs) is necessary to enable their deployment on energy-constrained devices. State-of-the-art methods leverage approximate multipliers to execute NNs reducing the inference energy without heavily affecting the accuracy. However, previous works usually require a specialized hardware accelerator and are limited to fixed multipliers or reconfigurable ones with few approximation levels. This paper introduces MARLIN, a framework to deploy layerwise approximate NNs on PULP, a microcontroller with a RISC-V core. A multiplier architecture, with runtime selection of 256 approximation levels, is developed and integrated into the PULP cluster cores, enabling runtime configuration through control status register (CSR) instructions embedded within the code. The PULP toolchain is adapted to incorporate the approximation level selection within the instruction flow seamlessly. MARLIN leverages the genetic algorithm NSGA-II to search for the best configurations among thousands of approximate NNs. The framework is validated by simulating an approximate NN trained with the MNIST dataset on PULP. Moreover, MARLIN is used to optimize and approximate six ResNet models trained with the CIFAR-10 dataset. In particular, for ResNet-56, the most complex NN used in the experiments, the multiplication energy is reduced by 23.9% while retaining 99% of the accuracy of the exact model.

Index Terms—Approximate computing, neural networks, RISC-V, hardware acceleration, reconfigurable computing.

I. INTRODUCTION

IN recent years, there has been an increasing necessity to include neural networks (NNs) in embedded systems to deliver more advanced functionalities. Nonetheless, NNs superior task accuracy comes at the cost of high computational

complexity and memory requisites, thereby presenting challenges in deploying them on energy-constrained devices, such as microcontrollers (MCUs) [1], [2]. Over the past decade, dedicated hardware accelerators [3]–[9] have been developed to optimize energy efficiency and throughput during inference by reducing the data movement and the cost of arithmetic operations. The latter usually accounts for around a fourth of the total inference energy, as the convolutional and fully connected layers of NNs involve millions of multiplications and additions [10]. Nowadays, quantization-aware training can reduce the bitwidth of NN models to 8 bits or below with little or no loss in accuracy [3], [11]–[13]. It lowers the amount of memory traffic and the computational cost, allowing the deployment of NNs on low-power devices with no support for floating-point arithmetic. Layer-wise quantization leverages the different degrees of robustness and tolerance to error introduction of NN layers. Therefore, a runtime reconfigurable multiplier supporting operands with different bitwidths is necessary to leverage mixed-precision and layer-wise quantization [3], [4], [11], [12]. Similarly to quantization, approximation is another co-design technique mainly aimed at reducing the inference energy [14]. The basic idea is to reduce computational complexity and cost using operators that produce inexact results. However, as already pointed out, there is high variability in the sensitivity of single layers inside the same model and among different models. Therefore, designing a multiplier with different approximation levels is fundamental to ensure flexibility and adaptability. Several strategies have been explored in the literature to design hardware supporting layer-wise approximate NNs [15]–[17], leveraging retraining or parameters fine-tuning to reduce the accuracy degradation. However, previous works rely on specialized accelerators and support very few approximation levels [17], [18], limiting the possibility of finding the optimal error-accuracy trade-off; other works use several non-reconfigurable multipliers instances in the accelerator’s systolic array [15], [19], [20]. The latter approach increases the area overhead and prevents the same hardware from executing NNs with different parameters or NNs with different model architectures with the same energy efficiency or accuracy. Since Internet of Things (IoT) devices have limited area and power budgets [1], [2] and must be able to adapt to different workloads and performance targets, it is necessary to adopt a layer-wise approximation strategy that relies on a single multiplier offering several accuracy levels. Moreover, the architecture using this multiplier

Manuscript received XXXXXXXXXXXX XX, XXXX; revised XXXXXXXXXXXX XX, XXXX; accepted XXXXXXXXXXXX XX, XXXX. Date of publication XXXXXXXXXXXX XX, XXXX; date of current version XXXXXXXXXXXX XX, XXXX. This work is partially supported by TRISTAN project, nr. 101095947, which has received funding from the Key Digital Technologies Joint Undertaking and its members. This work is partially supported by NODES project nr. ECS00000036, which has received funding from the MUR – M4C2 1.5 of PNRR funded by the European Union - NextGenerationEU. This article was recommended by Associate Editor XXXXXXXXXXXX XXXXXXXXXXXX. (Flavia Guella, Emanuele Valpreda, and Michele Caon contributed equally to this work.) (Corresponding authors: Flavia Guella, Emanuele Valpreda.)

Flavia Guella, Emanuele Valpreda, Michele Caon, Guido Masera, and Maurizio Martina are with Department of Electronics and Telecommunications, Politecnico di Torino, Torino, 10129, Italia (email: flavia.guella@polito.it, emanuele.valpreda@polito.it).

Color versions of one or more of the figures in this article are available online at XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.

Digital Object Identifier XX.XXXX/XXXX.XXXX.XXXXXXXXXX

should be programmable and reconfigurable to support the deployment of different NNs, reusing the same hardware as efficiently as possible without requiring a redesign or recall of the IoT device. Such flexibility allows choosing which NN is suited for a particular scenario, prioritizing the battery life of the edge device or the accuracy of the predictions.

In this work, we present MARLIN, an automated layer-wise approximation co-design methodology that enables searching for the optimal energy-accuracy trade-off and deploying approximate NNs on a hardware accelerator. We leverage a runtime-reconfigurable parallel tree multiplier featuring 256 error levels, assigning low-power inaccurate multiplier configurations to layers with high error resilience and more accurate, albeit less efficient, setup to layers with low error tolerance. Moreover, we search for different network-level approximate configurations, i.e., NNs with the same model architecture but a different layer-wise approximation and, consequently, energy-accuracy trade-off, supporting different workload priorities. In order to rapidly deploy an approximate NN and prove the portability of this method to a known open-source hardware, we selected the PULP MCU [21] as the target IoT platform with RI5CY [22], a RISC-V core. The search of the approximate NNs configuration is done with a non-dominated sorting genetic algorithm (NSGA-II) [23]. This procedure is automated and executed offline, before the NN is mapped to any specific hardware. The configuration of the arithmetic units, using the search results, is done online. The contributions of this work are summarized as follows:

- A layer-wise approximation strategy that reduces the energy of arithmetic operations while retaining the original task accuracy, applicable to any NN topology with convolutional and fully connected layers, with no modification to the model architecture. During the optimization process, relying on NSGA-II, approximate NNs are evaluated by their error resilience and energy, assigning to each layer a different multiplier configuration.
- A runtime reconfigurable signed multiplier supporting 256 approximation levels.
- The post-synthesis simulation of the proposed methodology on a RI5CY core, adapted to include the proposed approximate multiplier, supporting the runtime selection of the accuracy. The entire deployment process of the approximate NN is automated and requires no specific knowledge of the optimization method.
- The code used in this work, the experimental data, and the instructions to replicate the results presented in this paper are available at <https://github.com/vlsi-lab/MARLIN>

This article is organized as follows: Section II introduces a background on NN accelerators and relevant works on approximate inference, discussing the differences with MARLIN, Section III details the co-design framework, the search strategy, the multiplier design, and the modified PULP-toolchain, Section IV presents the results, a comparison with the state-of-the-art, and discusses the trade-offs of reconfigurable approximate computing, and Section V summarizes the paper.

II. BACKGROUND AND RELATED WORKS

A. Hardware Accelerators and Mapping

NN accelerators are typically based on a hardware architecture that comprises a memory hierarchy and an array of interconnected processing engines (PEs) [4], [7]–[10], similar to the one depicted in Figure 1. The memory hierarchy usually comprises the system’s main memory (off-chip), global buffers (on-chip), and the registers within the processing engines. The off-chip and on-chip memories are connected through the system’s bus, whereas the PEs communicate through a network-on-chip. The execution of a NN is scheduled with a

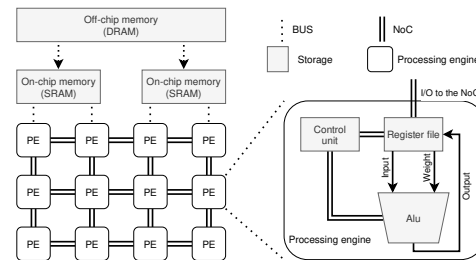


Fig. 1. Generic high-level model of a hardware accelerator for NN inference.

dedicated mapper that generates the instructions and partitions the resources, minimizing the energy and latency associated with the data movement. The complete cost of a multiplication considered by a mapper includes the energy and latency required to read all the operands, move them through the memory hierarchy, compute and store the result [9], [10], [24], [25]. The mappers used in [7]–[9] can be compared to the compilers used to generate machine code for processors such as RI5CY [22], which purpose is to optimize the performance and resource usage. The approximation methodology described in Section III is orthogonal to the mapping process as it only modifies the energy associated with the arithmetic operations and not the data movement. In [24] a tool that predicts the energy of approximate NN with a mapper based on [10] is presented validating the assumption that approximate computing only modifies the arithmetic energy without affecting the data movement. Therefore, MARLIN does not influence the mapping and could be easily integrated in [7]–[9], similarly to what has been done for DORY in Section III-E, as discussed in Section IV-D. Alternative hardware architectures for energy efficient NN inference at the edge are MCUs. ISA extensions and dedicated dot-product units inserted in the RISC-V pipeline are used in [5], [6], whereas in [26] a low-power neural processing unit is connected to the MCU through the bus. In this work, we selected PULP [5] as the target platform, a MCU-based low-power computing platform with a host CPU relying on the RI5CY or zero-riscy cores and equipped with a multi-core programmable cluster. The motivation of using PULP is twofold: the RTL and the toolchain are open-source and well documented [21], and, being a MCU-based platform, PULP truly represents a low-power IoT device with strict resource constraints [2]. Additionally, PULP is selected to produce a prototype that can be shared and adapted, without limiting the compatibility of this methodology to the PULP platform, as highlighted by the fact that the layer-wise approximation

178 strategy of Section III-B and the deployment process of
 179 Section III-E do not have any hardware dependencies except
 180 the multiplier. The PULP project includes libraries and tools
 181 to easily export a NN model written in PyTorch to C code
 182 compatible with the MCU. The MARLIN framework includes
 183 modified versions of the RISC-V core and software tools,
 184 adapted to include the approximation level selection through
 185 control status register (CSR) instructions, whose generation
 186 and compilation are added to the original PULP toolchain.

187 *B. Approximate Neural Networks*

188 In [19], 600 non-reconfigurable approximate multipliers are
 189 tested with a multi-layer perceptron for MNIST and LeNet-5
 190 for the Street View House Numbers dataset. Each approximate
 191 neural network is executed using one of the 600 multipliers for
 192 every convolutional layer following five retraining steps, thus
 193 generating 600 different sets of weights for each model. In
 194 [16], [27] is suggested that hardware-aware retraining, while
 195 being a time-consuming, resource-intensive strategy, can miti-
 196 gate the effect of approximation. Mrazek et al. in [15] present
 197 ALWANN, a framework for the approximation of NNs where
 198 the assignment of each layer to an approximate multiplier,
 199 among the eight-bit ones in EvoApproxLib [28], is performed
 200 through the multi-objective genetic algorithm NSGA-II. The
 201 parameters of each approximate NN are fine-tuned (updated
 202 w.r.t. the starting model) without retraining. Therefore, for
 203 each approximate configuration, a new set of weights must
 204 be used for each convolutional layer. Similarly, Jain et al. in
 205 [20] present a methodology to map the layers of a NN on a
 206 group of systolic arrays, each composed of several instances
 207 of one approximate multiplier. The arrays are part of the same
 208 accelerator, with each region processing only one layer of the
 209 network. Contrary to [15], the weights are not updated; there-
 210 fore, the original weights can be used with different approxi-
 211 mate configurations. However, considering several static mul-
 212 tiplier architectures is not a scalable approach for a general-
 213 purpose processor due to the area overhead; this method also
 214 impacts flexibility in a custom array accelerator. Our work
 215 proposes a single runtime-reconfigurable multiplier with 256
 216 approximation levels, trading off the complexity of additional
 217 control logic with increased flexibility. In [17], Tasoulas et al.
 218 propose a methodology based on the modification of the bias
 219 parameter of each layer to alleviate the approximation error.
 220 Similarly to [15], this approach generates a new set of weights
 221 for each approximate NN. However, their multiplier features
 222 only three runtime adjustable approximation levels. Moreover,
 223 the reconfiguration is handled by chaining two bits to each
 224 weight stored in memory, increasing the storage requirements
 225 and energy associated with data movement. According to [29]
 226 and [30], the resilience of a NN model to adversarial attacks
 227 depends on the approximate multiplier adopted. Consequently,
 228 using different configurations, including an exact one, is sug-
 229 gested to achieve higher error tolerance in various scenarios.
 230 MARLIN applied to [30] would remove the constraint of
 231 using a single fixed approximate multiplier, enabling error
 232 compensation, used in [15], [17], [20] and this work to
 233 improve the accuracy. Moreover, MARLIN can eliminate the

234 limitations of [29], in which 13 different multipliers are used
 235 within each PE to support 13 approximation levels, enabling
 236 21.3 times more approximation levels with a thirteenth of the
 237 multipliers instances. Similarly to [18], we selected a RISC-
 238 V-based deployment platform, thus simplifying the software
 239 configuration of the approximate hardware. Nevertheless, in
 240 [18], the configuration signals, handled by a control unit
 241 external to the core, are generated by the user, thus relying
 242 on his expertise rather than on an automated mechanism.
 243 We overcome these limitations by embedding the runtime
 244 approximation control inside the instructions processed by the
 245 RISC-V core, leveraging the flexibility of the PULP platform.

246 **III. PROPOSED METHODOLOGY**

247 *A. Overview*

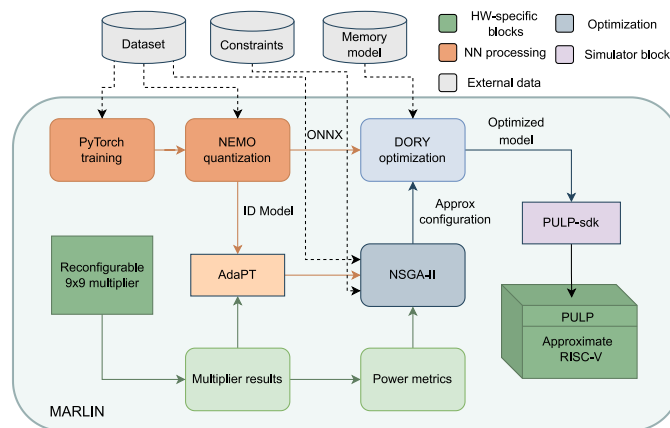


Fig. 2. MARLIN framework with hardware support specific for PULP SoC.

248 Our framework comprises different building blocks inter-
 249 acting with one another to determine a flow covering from the
 250 model definition up to its hardware deployment. From now
 251 on, we study the specific case of PULP platform, depicted in
 252 Figure 2. The first external input required is a valid dataset,
 253 such as MNIST or CIFAR-10, which defines the target task
 254 of the NN model. Given a specific application, there are
 255 often constraints on the minimum acceptable accuracy or the
 256 maximum tolerable energy consumption. Once the training
 257 dataset is provided with the specifications, a suitable NN
 258 model is identified and described with PyTorch [31]. At this
 259 stage, hyperparameters tuning is crucial to obtain consistent
 260 results and a model that is already robust to quantization
 261 errors. This phase implies choosing the number of training
 262 epochs, the type of optimizer, and other learning parameters.
 263 A standardized representation of the NN in the form of a
 264 data flow graph is required to port the model to PULP.
 265 For this reason, the trained NN is passed to NEMO [32],
 266 which transforms a floating-point model to an integer one in
 267 ONNX format. The precision of the model is fixed at this
 268 point in the procedure, with the added constraint that the bit-
 269 width of weights and activations cannot be above eight bits,
 270 either signed or unsigned. Up to this point of the procedure,
 271 the model has no knowledge of the approximation. On the
 272 hardware side, a reconfigurable inexact multiplier is designed

and instantiated in MARLIN with a 9x9 bits parallelism. As MARLIN's optimization software only requires, for each configurable approximation level, a look-up table (LUT) storing all the possible results for each couple of input operands and the average power to execute a single multiplication, the multiplier block is interchangeable. A single runtime reconfigurable approximate unit or several multipliers with fixed approximation levels could be integrated into the framework with little or no modification. Once this high-level description of the computational unit is available, the approximate model can be implemented and tested through the AdaPT library [33]. Any NN topology built with PyTorch's convolutional and fully connected layers can be easily included in MARLIN by overloading the layer definitions with the AdaPT ones without retraining or changing the model architecture. MARLIN solves the complex multi-objective problem of assigning an optimal approximation level to each layer through NSGA-II, described in Section III-B. It requires repeated simulations of the model with the selected configurations to evaluate their accuracy and power consumption. The obtained Pareto front will show different possible trade-offs between accuracy and power, corresponding to the two fitness functions NSGA-II tries to optimize. The last step that MARLIN performs on the software side is the C code generation to execute the model on the target hardware, presented in Section III-D. A modified version of DORY [34] is used to accomplish this task. This is the first part of MARLIN which requires knowledge of the actual hardware architecture, including a detailed high-level description of every memory level size and latency to perform memory tiling effectively. For this work, PULP was selected as the target platform among those supported. DORY receives as an input the ONNX model generated by NEMO and an additional node-by-node dictionary of the NN containing information of the approximation of each layer retrieved by NSGA-II. The modified DORY tool generates the C code for the provided approximate architecture with the received configuration. For this purpose, DORY has to be aware of the modifications the PULP platform undergoes. An approximate unit is added to the execution stage of the cluster cores to approximate all the relevant instructions in the computation of convolutional layers. It is based on the same reconfigurable multiplier, whose LUTs are used by NSGA-II and AdaPT. This unit is managed through a dedicated CSR in charge of activating, deactivating, and configuring its approximation level. In Section III-E is discussed how the modified DORY automatically inserts CSR write and set instructions in the C code of the model to enable the approximate unit when required. The final code runs on the PULP platform through PULP-SDK. The model can be executed by providing input data read from the external L3 memory while the weights are stored in L2 or L3 memory, depending on their sizes. The support of a real hardware RTL on which the model can run is crucial for the validation of the proposed co-design methodology, allowing accurate estimations of the metrics of interest on a complex system.

B. Genetic Search of Optimal Inter-Layer Approximation

In this work, we use NSGA-II, to solve the multi-objective problem of finding NN configurations with different trade-offs

between energy and accuracy. NSGA-II is a multi-objective genetic algorithm that evolves a population of solutions using non-dominated sorting and crowding distance assignment to classify and rank individuals based on their dominance and diversity. Crossover, i.e., recombination of different chromosomes, and mutation, i.e., variation of a gene, are applied to create offspring solutions, which are then integrated with the parent population. The selection process favors solutions from less crowded Pareto fronts and those with higher crowding distances, promoting the front exploration and providing a diverse set of non-dominated solutions [23]. The motivations for choosing the NSGA-II algorithm are its proven effectiveness in multi-objective optimization and relative ease of implementation and tuning compared to other alternatives such as reinforcement learning or Bayesian optimization. NSGA-II generates a set of optimized approximate NN configurations and selects for each one which approximation level is more suitable for each convolutional layer. Algorithm 1 details the

Algorithm 1 Approximation level selection with NSGA-II

```

1:  $\triangleright M$  is the quantized exact model
2:  $\triangleright axx\_mult$  is the approximate multiplier
3:  $\triangleright Ng$  is the number of generations
4:  $\triangleright Np$  is the population size
5:  $\triangleright Pc$  is the crossover probability
6:  $\triangleright Pm$  is the mutation probability
7:  $\triangleright \vartheta$  is the chromosome of  $L$  elements, in range  $[0, A]$ , storing the mult. configuration
8:  $\triangleright f_1(\vartheta), f_2(\vartheta)$  are the fitness functions to optimize
9:  $\triangleright P_n$  is the population at iteration  $n$ , with size  $Np$ 
10:  $\triangleright$  Initialization
11:  $L \leftarrow count(M.Conv)$   $\triangleright$  Number of Conv. layers in the model
12:  $A \leftarrow A_0$   $\triangleright$  Number of approximation levels for multiplier
13:  $P \leftarrow P_0$   $\triangleright$  Initial population vector randomly set
14:  $f(P_0) \leftarrow (f_1(P_0), f_2(P_0))$   $\triangleright$  Initial fitness evaluation
15:  $\triangleright$  Execution
16: for  $(n = 0; n < Ng; n++)$  do
17:    $Q_n \leftarrow Tournament(P_n, Pc, Pm)$ 
18:    $retrain\_models(M, axx\_mult, Q_n)$ 
19:    $f_1(Q_n) \leftarrow 1/accuracy(M, axx\_mult, Q_n)$ 
20:    $f_2(Q_n) \leftarrow energy(M, axx\_mult, Q_n)$ 
21:    $f \leftarrow (f_1, f_2)$ 
22:    $R_n \leftarrow P_n + Q_n$   $\triangleright$  Total population, size  $2Np$ 
23:   for each  $\vartheta$  in  $R_n$  do
24:      $Rank(\vartheta)$ 
25:      $F_i \leftarrow F_i \cup \vartheta$   $\triangleright F_i$  are the fronts
26:   end for
27:   for each  $\vartheta$  in  $R_n$  do
28:     for each  $\phi_k$  in  $f$  do
29:        $dis_\vartheta \leftarrow dis_\vartheta + Crowding\_distance(\vartheta, \phi_k)$ 
30:     end for
31:   end for
32:   Order  $R_n$  based on fronts and crowding distance
33:    $P_{n+1} \leftarrow$  best  $Np$  solutions in  $R_n$   $\triangleright$  Update iteration counter
34: end for
35: return  $\theta_{best}$   $\triangleright$  Optimum Pareto front is returned

```

NSGA-II search flow. Each chromosome has a dimension L , which is the number of layers composing the NN. The alleles of each gene are encoded as an integer number between 0 (exact level) and A , which is the number of approximation levels supported by the multiplier. Single-point crossover is used to combine chromosomes while maintaining inter-layer dependencies between approximate configurations, a strategy used in [20] to reduce the effect of computation errors on the NN accuracy without retraining. At the beginning of each iteration, Np approximate NNs are retrained with 10% of the training split (0.1 epoch). Then, the accuracy is evaluated with the validation dataset. Contrary to previous works [15], [17], the accuracy of candidate inexact NNs is not evaluated

361 immediately, but after a quick retraining with a fraction of an
 362 epoch. By leveraging partial retraining and validation, each
 363 NN configuration is evaluated by its resilience to computation
 364 errors and the retraining effort required to recover from
 365 such errors. Solutions with faster recovery will have higher
 366 validation accuracy than others that are less fit, using the
 367 same number of training samples, and therefore have an
 368 evolutionary advantage. Therefore, retraining (or fine-tuning)
 369 is used to compensate for the error and enhance the design
 370 space exploration. For what concerns the fitness evaluation in
 371 Algorithm 1, the inference energy is estimated as in [15], [17]
 372 by multiplying the number of multiplications of each layer
 373 of the model M with the average energy of the approximate
 374 multiplier when set to the approximation level defined by the
 375 corresponding gene of chromosome ϑ . After the evaluation
 376 of the two fitness functions, the algorithm continues with the
 377 mutation, crossover, tournament selection, ranking, and finally,
 378 the evaluation of the crowding distance and the generation
 379 of the new front. The cycle starts anew until all the N_g
 380 generations have been evaluated.

381 C. Reconfigurable Approximate Multiplier

382 In this work, we employ a single-cycle multiplier architec-
 383 ture to introduce minimum modifications in the control flow
 384 of the RISCY core. As our primary purpose is guaranteeing
 385 maximum versatility and ensuring portability to different hard-
 386 ware with minimum effort, a parallel multiplier architecture
 387 based on the Dadda reduction tree [35] is selected. The Dadda
 388 algorithm is employed to compress, through half-adders and
 389 full-adders, the matrix of partial products generated in the first
 390 step of the multiplication, following an as late as possible
 391 approach. It is preferred to the Wallace structure as it shows
 392 lower delay and complexity. The modified Baugh-Wooley al-
 393 gorithm [36] is employed to handle signed multiplication with
 394 minimum overhead in the size of the partial product matrix.
 395 Both techniques are general and scalable to arbitrary operand
 396 bit-width. A variation of the truncation mechanism proposed
 397 in [37] is identified as the target strategy to manage the
 398 dynamic setting of approximation and precision. It allows for
 399 easy support of approximate and exact configurations, both for
 400 full and reduced bit-width of the operands. Truncation relies
 401 on a masking signal to select specific columns of the partial
 402 product matrix to fix at zero to reduce the switching activity,
 403 hence dynamic power, of the logic gates in that section of
 404 the matrix, at the expense of an incorrect output. The number
 405 of reconfiguration levels is selected considering that when the
 406 approximation is extended to the most significant half of the
 407 result, the error becomes unbearable, as argued in [38] and
 408 [39]. An externally configurable masking signal noted as a , is
 409 introduced to manage the selection of the approximation level,
 410 as shown in Figure 3 for the case of 9-bit inputs and a on eight
 411 bits. Contrarily to [39], each mask bit a_j of a corresponds to a
 412 column of the matrix; there is no sharing of the configuration
 413 signals. From bit 2 onward, the probability of the output
 414 bit being one or giving a carry-out is higher than 50%, as
 415 intuitively proven by the fact that the number of bits stacked
 416 in the columns of the matrix is greater than two (Figure 3).

This high likelihood justifies the choice to gate the first row of
 the partial product matrix, from position 2 to 7, to one, when
 the corresponding bit of the approximation mask is active, and
 it is implemented through the OR with a_j complemented in
 Figure 3. All other bits are gated to zero, similarly to [37],
 using two-inputs AND gates for the masking. An additional
 feature of the designed architecture, which is uncommon to
 most approximate multipliers, is the capability of runtime
 reconfiguration of the precision of the operands. A precision
 masking signal p is introduced to perform data-gating on
 the partial product matrix using a mechanism similar to the
 approximation. The p mask is externally configured according
 to the precision of the expected result, thus allowing power
 saving when mixed-precision multiplications are required. The
 minimum supported bit-width for the input operands is fixed
 to two. The precision mask has the length of the output minus
 four. Figure 3 shows the precision signal on fourteen bits p_j
 covering the most significant part of the partial products. When
 a precision mask bit is set to zero the corresponding column
 of the matrix is entirely zeroed. The complete logic inserted in
 the generation of the partial product matrix to manage multiple
 approximation and precision levels is depicted in Figure 3.

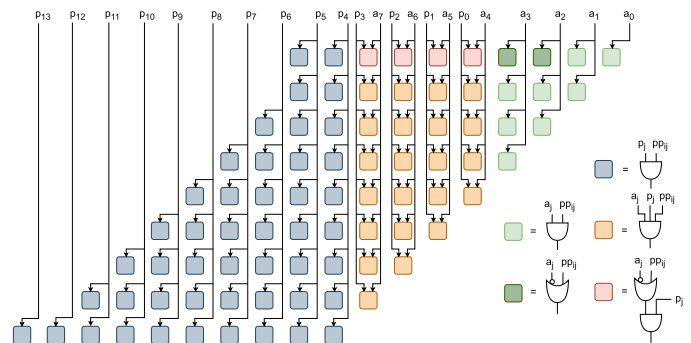


Fig. 3. Precision and approximation configuration management for the proposed multiplier. The approximation level is selected with the mask a , while the precision is selected with the mask p . a_j indicates the j^{th} bit of the approx_mask signal a , p_j the j^{th} bit of the precision_mask signal p , while pp_{ij} is the j^{th} bit of the i^{th} partial product evaluated according to modified Baugh-Wooley algorithm.

D. Reconfigurable Approximate RISC-V Core for PULP SoC

The RISCY core architecture instantiated in the PULP cluster must be adapted to enable reconfigurability in terms of approximation and precision. The first issue to tackle to introduce inexact operators in the core is how to expose them in the instruction set architecture (ISA) so that a programmer can effectively use them. A possible solution is the one proposed in [18]: adding custom instructions that, when decoded, configure the execution stage to use approximate operators. However, this approach requires an additional instruction for each inexact arithmetic operation supported by the hardware, defined with a new custom format capable of encoding the approximation level. Another drawback is related to the fact that, in this specific case, the C executable code is generated automatically by the DORY tool. If custom instructions were used, the user would have to replace the standard instructions with the custom ones, whenever necessary, analyzing the

generated C code line-by-line. The same could be achieved by making the compiler aware of the approximated instructions and where they are needed, but that would be time-consuming to implement and maintain. The methodology suggested in this work addresses flexibility and simplicity by defining a new custom CSR handling all the control and configuration of approximate operators. This approach is scalable; a single 32-bit register can manage all the new operations and does not occupy additional instruction encoding space. It also enables reconfiguration, as part of the register bits control the approximation and precision level. Finally, it is much more programmer-friendly as it does not require significant changes in the C code of the microcontroller, except for the addition of CSR instructions. In order to achieve these results, the proposed methodology to enable the online configuration of the approximate multiplier relies on the following steps. First, a CSR write instruction sets the `precision_mask` and `approx_mask` fields according to the specification of the layer. Secondly, before the computation starts, a CSR set instruction enables `approx_mac` and `approx_dot8`, which are disabled when the computation is over. Each CSR

Algorithm 2 PULP-NN matmul function pseudocode

```

1: ▷ ch_in/ch_out input/output channels of the Conv layer
2: ▷ k_x/k_y filter dimension along x/y
3: ▷ im2col is ch_in · k_x · k_y
4: ▷ col_cnt_im2col is im2col & 0x3
5: ▷ chan_left is ch_out & 0x3
6: ▷ Initialization
7: Load params. from stack and define variables ▷ 26 instr
8: ▷ Execution
9: for (i = 0; i < ch_out >> 2; i + +) do
10: Setup ▷ if first iteration 41 instr, else 6 instr
11:   for (j = 0; j < im2col >> 2; j + +) do
12:     Setup ▷ 15 instr
13:     Multiply and accumulate + save ▷ 15 instr · (im2col >> 2) + 9 instr
14:   end for
15:   if (im2col >> 2 == 0) then; Setup end if ▷ 12 instr
16:   while (col_cnt_im2col != 0) do
17:     Setup ▷ 4 instr
18:     Multiply and accumulate + save ▷ 15 instr · (im2col >> 2) + 2 instr
19:   end while
20:   Quantize and save results ▷ 54 instr
21: end for
22: Setup ▷ 10 instr
23: while (chan_left) do
24:   Setup ▷ if first iteration 23 instr, else 1 instr
25:   for (j = 0; j < im2col >> 2; j + +) do
26:     Setup ▷ 6 instr
27:     Multiply and accumulate + save ▷ 6 instr · (im2col >> 2) + 4 instr
28:   end for
29:   if (im2col >> 2 == 0) then; Setup end if ▷ 6 instr
30:   while (col_cnt_im2col != 0) do
31:     Setup ▷ 4 instr
32:     Multiply and accumulate + save ▷ 6 instr · (im2col >> 2) + 1 instr
33:   end while
34:   Quantize and save results ▷ 13 instr
35: end while
36: Save parameters and return ▷ 24 instr

```

instruction takes one clock cycle. In the general case, the last couple of CSR instructions are executed a number of times which depends on the tiler split performed by DORY. Their position in the code is optimized to produce minimum overhead in the control flow, considering the presence of MAC instructions that must produce the correct result. The usage of three instructions, rather than two, is forced by the specific organization of the template C files provided by DORY and PULP-NN C library [22]. The CSR instruction activating the approximate unit is the one executed more

frequently. It is located before the `matmul` function, whose pseudocode and number of assembly instructions are depicted in Algorithm 2. In a pessimistic estimation, a CSR set is performed once for each `matmul` function call, providing a quantitative measure of the reconfiguration overhead on the execution time. The GCC compiler is extended to account for the new `approx` CSR, whose fields are given in Figure 4. The

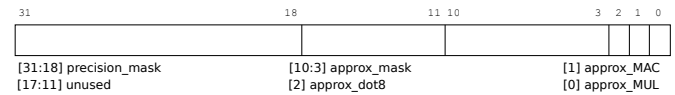


Fig. 4. The `approx` CSR configuration.

Xpulp ISA extension [22] provides additional multiply-related instructions compared to the basic ones of RV32M. Some of these, such as MAC and SIMD dot products, are useful for NN inference. Here, the choice is to provide an approximate implementation only for instructions used in convolutional and linear layers. An in-depth analysis of the disassembly code produced by custom convolutions and a complete NN is performed to select them. The assembly instructions included in these benchmarks are approximated, together with others for which it is straightforward to extend support; they are all listed in Table I. In this first implementation, the approximate pipeline only computes 8-bit multiplications, even when the issued instruction expects a 32-bit or 16-bit operation. This restriction cannot cause any error assuming that NN layers quantization is always on 8 bits or below, which is the ordinary case. The instructions for which approximate support is provided are split into three subgroups, according to Table I. For each category, the `approx` CSR has a configuration bit; when this bit is set, all the instructions belonging to that group are executed in approximate mode. Besides the custom CSR, a unit responsible for inexact computation is inserted in the execution stage of the pipeline alongside the exact multiplier unit, as shown in the high-level block diagram of the approximate core in Figure 5. This design choice requires

TABLE I
APPROXIMATE INSTRUCTIONS MNEMONICS

MAC	MUL		DOT8	
p.mac	mul	p.mulsN	pv.dotup.b	pv.sdotup.b
p.macsN	p.muls	p.muluN	pv.dotusp.b	pv.sdotusp.b
p.macuN	p.mulu		pv.dotsp.b	pv.sdotsp.b

some modifications in the decoding phase of the instruction. Based on some control signals, the decoder has to activate either the correct or the inexact unit. The arithmetic block that is not selected for the instruction currently in the decode stage does not perform any operation in the next clock cycle as its inputs are not updated. Four instances of the designed multiplier are allocated in the reconfigurable approximate unit, as in Figure 6. They are all used in parallel to perform SIMD dot products on 8 bits, while only one is activated for MUL and MAC-related operations.

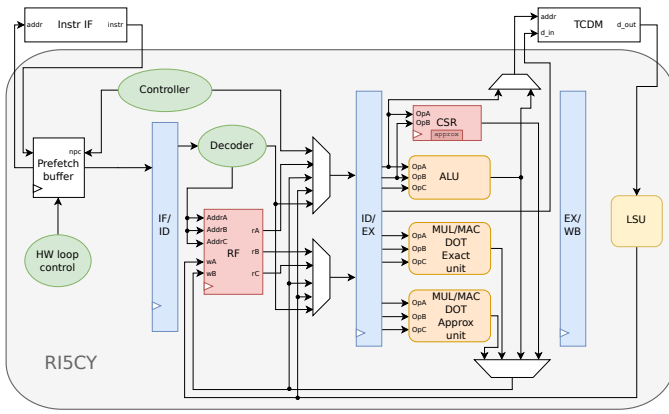


Fig. 5. RISCY pipeline with approximate operations support.

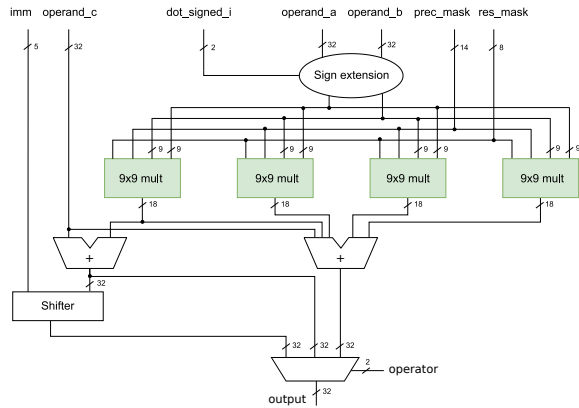


Fig. 6. RISCY approximate multiplier unit. The leftmost multiplier is in charge of MUL and MAC-related operations. The shifter is for immediate instructions with the N suffix in Table I. All multipliers work in parallel to execute dot8 operations and their result is fed to a 32-bit five-operands adder. The sign extension block handles the split of the 32-bit operands into four 8-bit chunks and their sign extension according to the decoded instruction.

E. Approximation in PULP Toolchain

Part of the PULP toolchain must be adapted to enable run-time approximation. Through NEMO and DORY libraries, the PULP platform offers software support to generate executable C code tailored to its architecture, starting from a PyTorch NN model. NEMO is a PyTorch complementary framework developed as a support tool to transform an already trained, full-precision NN into an integer one, performing the quantization and calibration of the model. DORY is an open-source tool for optimizing NNs mapping on PULP and other MCUs. Two building blocks of DORY, the configurable templates and PULP-NN back-end functions, are modified to automatically add the approximate CSR instructions in proper points of the C code, while the mapping optimization is unaffected. Besides an ONNX graph, the modified DORY uses as input a JSON file containing a layer-by-layer description of the quantization and approximation of the NN. Once these parameters become part of the DORY intermediate representation, they are used to fill hardware-specific template files with the correct CSR setting and generate the C code for the different layers. Relying on a JSON dictionary guarantees flexibility, as new items can be defined for each node in the network. Furthermore, it is

general; at this level, every type of approximate multiplier could be available, either reconfigurable or not. Moreover, as the precision information on the layer is kept separate from the approximation level, it is possible to configure the layer as exact but with reduced precision. This choice allows to save power by leveraging operations with reduced bit-width rather than with inexact computation.

IV. EXPERIMENTAL RESULTS

This section presents the computing setup used to conduct experiments to validate the proposed methodology, summarized in Figure 7. We synthesized and tested the modified RISCY with the reconfigurable multiplier and extracted relevant hardware metrics of the core and the arithmetic operator alone. Additionally, we compare our approach against state-of-the-art techniques that apply layer-wise approximation.

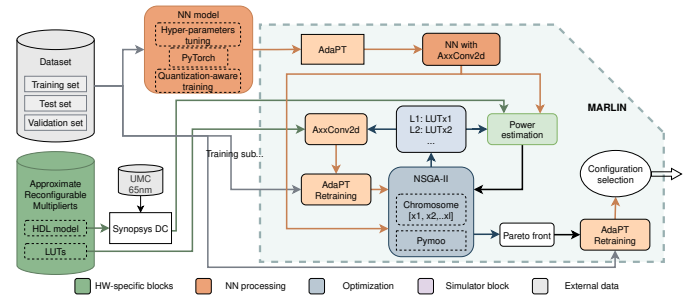


Fig. 7. MARLIN framework and computing setup for software simulation.

A. Multiplier Characterization

A 9-bit reconfigurable multiplier is designed according to the methodology in Section III-C to compute SIMD 8-bit dot products supporting all possible combinations of signed and unsigned operands in RISCY core. The multiplier is synthesized with Synopsys Design Compiler (DC) using the UMC 65 nm process technology library. The design is wrapped by a group of input and output registers to ease the enforcement of timing constraints. Similarly to [38], the target clock period is set to 2 ns, making the multiplier critical path delay compatible with the one it shows on the synthesized core. The `compile_ultra` command is issued to generate the gate-level netlist. Post-synthesis simulation is performed on 100000 random input samples to back-annotate the switching activity for accurate power estimation, as in [40]. Random input samples are a common approach in literature to characterize the power profile of approximate multipliers, [28], [37]. Table II shows the main hardware metrics and mean-relative error distance (MRED) obtained for the maximum precision configuration with different approximation levels.

$$MRED = \frac{1}{n} \sum_{i=1}^n \frac{|\hat{y}_i - y_i|}{|y_i|} \quad (1)$$

The MRED, a commonly used metric to estimate the performance of inexact arithmetic units [38]–[40], is defined in Equation (1), where n is the number of possible combinations of input values (i.e. $n = 2^{18}$, for a 9x9 multiplier), and \hat{y}_i

589 and y_i are the i^{th} approximate and exact results, respectively. 590 Error metrics are evaluated through exhaustive simulation on 591 the entire inputs dynamic [38], [40]. Table II presents the 592 same estimations, obtained with the same constraints and 593 testing conditions, for an exact 9x9 signed multiplier described 594 behaviorally in HDL and optimized by Synopsys DC, through 595 Synopsys DesignWare (DW) library. Our multiplier has an 596 area overhead of 28%, due to the additional logic for online 597 reconfigurability. This feature does not impact on the critical 598 path, as it remains under the 2 ns constraint. Finally, our 599 multiplier, in the exact configuration, saves 41.4% of power 600 compared to the one by Synopsys DW, while the highest approximate level saves up to 60.1%.

TABLE II
METRICS OF THE 9X9 SIGNED RECONFIGURABLE MULTIPLIER AND
COMPARISON WITH AN EXACT 9X9 MULTIPLIER

Design	Area [μm^2] (GE) ¹	Arrival time [ns]	Approx level	Total power [μW]	MRED
Exact	607.3 (422)	1.7	-	414.0	0
MARLIN	822.6 (572)	1.8	0	241.2	0
			127	183.0	0.07
			255	164.4	0.18

¹GE is the 2-input drive-strength-one NAND gate equivalent area.

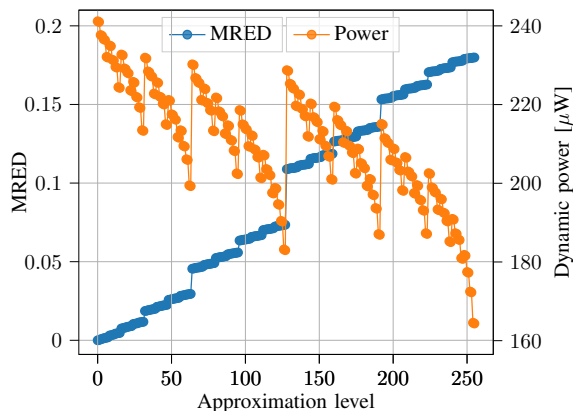


Fig. 8. MRED and dynamic power variation of the proposed multiplier with the approximation level for the full-precision 9-bit inputs configuration. The approximation level on the x-axis corresponds to the 1's complement of the approx_mask value.

602 Although the dynamic power variation with the approximation 603 setting does not follow a monotonic trend, as can be observed 604 in Figure 8, sub-optimal configurations are retained as they 605 come at no hardware cost in area, power and latency once the 606 8-bit approx_mask signal is inserted in the multiplier logic. 607 Every level has its error distribution, which can be optimal for 608 a specific NN layer. An optimal subset of the 256 configura- 609 tions might be selected offline depending on the dataset and 610 NN analysed. However, the proposed methodology aims to 611 be application-agnostic and versatile to several use-cases, thus 612 hardware and software provide support for the most generic 613 scenario. The jumps in power and MRED in Figure 8 can be 614 explained by looking at the logic in Figure 3. The MRED 615 shows higher steps when the approximation moves towards 616 columns on the left (e.g., from configuration 63 to 64, or from 617 191 to 192). On the contrary, more power is saved when more 618 columns, as far to the left as possible, are gated, reducing the

switching activity of the compressors in the Dadda Tree. This 619 condition in Figure 8 corresponds to approx_mask values 620 $\bar{a} = 2^j - 1, j \in [0, 1, 2, 3, 4, 5, 6, 7, 8]$. Consequently, when 621 approx_mask is set as above, that configuration will save 622 more power than the subsequent ones. Furthermore, it is a 623 Pareto optimal point since all the successive configurations 624 imply the next columns to the left to be approximated, thus 625 increasing the MRED.

TABLE III
COMPARISON BETWEEN MARLIN 8X8 SIGNED MULTIPLIER AND
STATE-OF-THE-ART APPROXIMATE MULTIPLIERS.

Multiplier	Conf	Exact conf	Online reconf	MRE	Area [μm^2] (GE)	Dynamic power [μW]
Ha [41]	-	✗	✗	0.102	461.2 (321)	170
Strollo [40]	-	✗	✗	0.053	496.8 (345)	181.6
Yang [42]	1	✗	✗	0.031	529.2 (368)	222.3
		✗	✗	0.041	516.2 (359)	206.6
		✗	✗	0.069	500 (348)	190.9
Mul8s [28]	1KV6	✓	✗	0	541.1 (376)	167.6
		✗	✗	0.089	378 (263)	116.3
		✗	✗	0.274	200.9 (140)	60.9
		✗	✗	1.347	126 (88)	36.6
de La Guia [37]	0	✓	✓	0	650.9 (452)	188.3
		✓	✓	0.374	650.9 (452)	130.3
		✓	✓	1.065	650.9 (452)	109.5
MARLIN	0	✓	✓	0	652.7 (454)	188.6
		✓	✓	0.236	652.7 (454)	130.6
		✓	✓	0.621	652.7 (454)	109.9

626 Table III compares different state-of-the-art techniques to 627 design approximate multipliers. All multipliers are 8-bit signed 628 and have been synthesized with a clock period of 2 ns and 629 characterized as for the 9x9 case. For a fair comparison, our 630 multiplier is rescaled on 8 bits. The design choice to fix some 631 bits of the partial product matrix at one rather than zero when 632 the corresponding columns are approximated improves the 633 MRED by up to 41.7% compared to [37] truncation approach, 634 with negligible area and power overhead. Compared to [40]– 635 [42], all based on approximate 4-2 compressors, our multiplier 636 has lower power consumption, for corresponding MRED, 637 while still covering a much wider error dynamic. Furthermore, 638 the approximate compressors approach, even when runtime 639 error tuning is implemented [38], [39], cannot provide an exact 640 configuration. This implies the need to pair an exact multiplier 641 with the approximate one to manage operations, such as 642 control flow ones, which must produce the correct output, 643 unacceptably increasing the total area. This consideration still 644 holds for Evoapproxlib multipliers [28]. Although, according 645 to Table III, [28] show better area and power metrics com- 646 pared to ours, the absence of reconfigurability precludes their 647 compatibility with MARLIN. As in [15], the implementation 648 of layer-wise approximation with [28] requires to instantiate as 649 many multipliers as the number of approximation levels, which 650 becomes inconceivable above certain values due to control, 651 area, and power overhead. In conclusion, our multiplier offers 652 several power-error trade-offs, spanning the MRED dynamic 653 of most of the 8x8 multipliers compared in Table III, standing 654 as the cheapest and most flexible solution to implement layer- 655 wise approximation. Our multiplier, unlike [28], [38], [40]– 656 [42], also enables runtime selection of the result bit-width to 657 add versatility and reduce power when full precision is not 658 required. For the 9-bit architecture, output precision from 18 659 bits down to 4 bits is supported through data-gating on the 660

661 most significant columns of the Dadda tree, with a maximum
 662 power saving of around 62%. In Table IV, we evaluate the
 663 trade-off of runtime precision setting with a ResNet-20 model
 664 executed with and without data-gating, and with different bit-
 665 widths. The average full precision power is evaluated with
 666 the multiplier configured to execute 9x9 bit multiplications,
 667 although input operands are quantized to lower precision. The
 668 average reconfigured power is evaluated when the masking
 signal p is set to match the input operand precision.

TABLE IV
RESNET-20 MULTIPLIER POWER WITH AND WITHOUT DATA-GATING

Activations precision	Weights precision	Avg mult pow full precision	Avg mult pow reconfigured	Absolute accuracy	Relative accuracy
8	8	239.15 μ W	237.93 μ W	91.50%	100%
6	4	227.85 μ W	170.09 μ W	91.43%	99.92%
4	4	225.85 μ W	125.35 μ W	89.87%	98.22%

669

670 B. Approximate RISC-V Core Characterization

671 The RI5CY core featuring the approximate extension is
 672 synthesized to extract area, delay, and power estimations using
 673 Synopsys DC and the UMC 65 nm library. The clock period
 674 is set to 5 ns. The timing constraints were relaxed with respect
 675 to the ones for the multiplier alone to also accommodate in
 676 the cycle time the four-operand 32-bit adder that follows it,
 677 as can be observed in Figure 6. As a matter of fact, the
 678 approximate multiplier instance in the core has a delay of
 679 around 1.9 ns, which makes its implementation mapping and,
 680 therefore, its energy contribution consistent with the analysis
 681 previously performed with a 2.0 ns clock period. The two
 682 cores with exact and reconfigurable approximate operators
 683 are both synthesized using the command: `compile_ultra`
 684 `-no_autoungroup -no_boundary_optimization`
 685 `-timing -gate_clock`. Table V compares area and timing
 values obtained. The delay constraints are satisfied by both

TABLE V
PERFORMANCE AND AREA COMPARISON FOR EXACT AND APPROXIMATE RI5CY AND THEIR RELATIVE MULTIPLIER UNITS

		Exact RI5CY	Approx RI5CY
Area [μ m ²] (GE)	Exact mult	14842.8 (10k)	-
	Approx mult	-	4737.2 (3k)
	Total	60621.1 (42k)	67006.8 (47k)
Timing [ns]	Exact mult	4.45	
	Approx mult	-	4.41

686 designs, and from the fourth column of table V it can be
 687 observed that our multiplier does not interfere with the micro-
 688 processor critical path which remains in the exact multiplier
 689 unit. The area overhead caused by the approximate unit alone
 690 is 7.8%, while the total overhead, considering the additional
 691 control part and the approx CSR, is 10.5%. The extra area
 692 cost is mainly due to the allocation of four reconfigurable
 693 multipliers in order to manage 8-bit dot products. We consider
 694 this overhead acceptable as this is the first prototype of this
 695 architecture; however, it could be significantly decreased if
 696 our four multipliers replaced the exact ones, which is possible
 697 since they also feature a non-approximate mode.
 698 The RTL model of the RI5CY is replaced by the gate-level

699

netlist for all cores in the PULP cluster to enable post-synthesis
 simulation and power estimation on a demonstrative use-case.
 A simple NN model is designed with PyTorch and used as a
 benchmark on MNIST dataset to verify the correct behavior
 of the entire framework and to collect power metrics. It
 comprises five convolutional layers, each followed by a ReLU
 activation function, and a final linear layer. The entire structure
 is depicted in Table VI. The model is trained for 30 epochs,

700
701
702
703
704
705
706

TABLE VI
CUSTOM NN ARCHITECTURE DESCRIPTION

Layer name	Output size	Kernel size	Output channels	# Mult
conv1	28x28	7x7	3	115224
conv2	28x28	5x5	8	470400
max pool	14x14	3x3	8	0
conv3	14x14	3x3	10	141120
conv4	14x14	3x3	16	282240
max pool	7x7	3x3	16	0
conv5	7x7	3x3	24	169344
max pool	3x3	3x3	24	0
linear	1x10	9x24	1	2160

with a batch size of 32, an initial learning rate of $3 \cdot 10^{-3}$, with
 a step factor of 0.3 every 5 epochs. Stochastic gradient descent
 with momentum 0.9 is used with a weight decay of 10^{-3} . The
 designed NN is run on the PULP platform for a single input
 image. For this model, the overhead of the CSR instructions
 execution can be quantified, according to Algorithm 2, in the
 worst case, which is the first convolutional layer, as one CSR
 set instruction every 403 instruction (0.25%). In the best case,
 which is the last convolution, the extra cost is 0.026%. This
 results in a negligible performance overhead due to the CSR
 switching, and thus of reconfiguration, on the overall processing.
 Through Siemens QuestaSim, the VCD dumps of the entire core
 and the approximate and exact multiplying units are collected.
 They are used as inputs for Synopsys Power Shell to extract
 power metrics based on the actual switching activity. Simulations
 are performed using the multipliers configurations obtained from
 the NSGA-II run that achieve accuracy over 90% with no retraining.
 For every configuration, an example for each of the ten possible
 categories is fed to the network, meaning numbers from zero to
 nine for MNIST. The exact post-synthesis RI5CY core is simulated
 with the same inputs, and the power of the accurate multiplier
 unit is collected; all results are averaged. Table VII reports,
 for the configurations listed in the first column, the NN test
 accuracy and, in the third column, the power of the multiplier
 unit (the exact one for the exact configuration in the first row,
 the approximate one for all other cases) averaged over the ten
 simulations. The fourth column contains the relative energy
 saving of the multiplier unit measured post-synthesis, the last
 column shows the relative energy saving estimated at the end of
 the NSGA-II search, as described in Section III-B. The first
 row of Table VII contains the power estimation of the exact
 RI5CY multiplier unit, where all operators, including the
 multipliers computing dot8 instructions, are described behaviorally,
 thus leaving the architecture selection to Synopsys DW. Consequently,
 for a meaningful comparison, the reference model for the NSGA-II
 relative estimation is the average energy consumed by a behavioral
 9-bit multiplier synthesized by Synopsys DC, with

707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745

746 the same settings of the approximate multiplier. Although
 747 the exact multiplier always executes housekeeping operations,
 748 these are a negligible fraction of its overall workload when all
 749 NN multiplications are mapped on it; thus, their contribution to
 750 the average power is minimal. Therefore, since the number of
 751 multiplications of the NN model is fixed, and so is the time
 752 the multiplier stays active, the relative energy is considered
 753 equivalent to the ratio between the average power of the
 754 approximate unit in the modified RISCY and that of the exact
 755 one in the unmodified core.

TABLE VII

POWER CONSUMPTION AND ENERGY SAVING OF THE MULTIPLYING UNIT WITH DIFFERENT LAYER-WISE CONFIGURATIONS FOR THE TARGET NN

Configuration	Test Acc [%]	Average RISCY mult power [μW]	Relative RISCY mult energy saving	Relative NSGA-II energy saving
Exact	98.7	10.46	0%	0%
[0, 0, 0, 0, 0]	98.7	2.606	75%	42%
[59, 31, 15, 12, 3]	98.7	2.509	76%	46%
[59, 31, 15, 31, 31]	98.5	2.474	76%	48%
[255, 63, 15, 31, 11]	97.8	2.412	77%	50%
[255, 126, 3, 63, 63]	91.3	2.403	77%	53%

756 The fourth column of Table VII shows that the obtained
 757 average energy saving on the multiplier unit is at least 75%
 758 when comparing the exact core and the approximate one with
 759 all multipliers configured as accurate (first and second row
 760 in Table VII). The maximum saving is 77%, which is more
 761 than 20% higher than the high-level estimation performed
 762 in NSGA-II. However, the advantage of adopting different
 763 approximate configurations is heavily reduced compared to
 764 the initial evaluation. This can be observed by rescaling the
 765 energy results in the fourth and fifth columns of Table VII
 766 with respect to the exact configuration of our multiplier. The
 767 highest estimated energy reduction, with respect to the model
 768 using the exact configuration of our multiplier for all layers,
 769 is 7.7%, with an accuracy loss of 7.4%, while the predicted
 770 saving was 20.1%. The cause of the gain drop, obtained by
 771 configuring the multiplier with a higher approximation, has to
 772 be addressed to the chosen task for the network. When the
 773 average power of the multiplier is estimated, 100000 random
 774 values with uniform distribution are used as inputs to the
 775 multiplier. However, the MNIST dataset is composed of black
 776 numbers on a vast white background, which means that the
 777 network inputs and, consequently, those of the multipliers are
 778 not uniformly distributed. Both input images and intermediate
 779 activations show a high concentration of zeros, contrary to
 780 the initial assumption. The main consequence of most values
 781 being zero is that data-gating, which is the primary source
 782 of power saving when approximation is applied, becomes
 783 ineffective as the operands are already zeros and have a low
 784 switching probability themselves. A higher sensitivity of our
 785 multiplier to zero input operands, compared to the DW one,
 786 could also explain the increased relative energy saving in the
 787 example with respect to the estimates used during the NSGA-
 788 II search. Even acknowledging the difference in the statistics
 789 of the operands, we run the optimization algorithm using the
 790 estimated average energy computed with uniformly distributed
 791 inputs, keeping it data-agnostic, as commonly done in state-of-
 792 the-art optimization frameworks, where performance metrics

793 are estimated independently from the statistics of the dataset
 794 [3], [10]–[12]. The latter enables our search algorithm to
 795 generalize to new data and thus demonstrate the efficacy of
 796 our method, an approach that was also used in [15]–[17], [24]
 797 to estimate the energy reduction with approximate multipliers.

C. Benchmark with CIFAR-10

798 We used 6 variations of the ResNet model architecture [43]
 799 to experiment with shallow and deep NNs for image classifica-
 800 tion, testing the effectiveness of MARLIN with CIFAR-10
 801 [44], a more challenging dataset than MNIST. We based the
 802 implementation of our ResNet models on the original paper
 803 and used the same model architecture and hyper-parameters,
 804 with 44k iterations instead of 64k, substituting the original
 805 multi-step scheduling of the learning rate with a cyclical
 806 scheduler [45], ranging between 10^{-1} and 10^{-4} . We opted for
 807 a cyclical learning rate instead of a stationary one to achieve
 808 faster convergence with fewer training iterations during the
 809 genetic search. We carried out separate quantization-aware and
 810 full precision training for the INT8 and FP32 models. All the
 811 experiments with approximate multipliers use the INT8 quan-
 812 tized models; the FP32 results are presented only to provide
 813 a comparison with full precision. We used scale quantization
 814 with the straight-through-estimator for the weights [13], while
 815 the activations are quantized using PACT [46]. Table VIII
 816 reports the NNs used in the experiments.

TABLE VIII
NEURAL NETWORKS USED IN THE EXPERIMENTS

Neural network	# Conv. layers	# Mult.	FP32 accuracy	INT8 accuracy	Design space size
ResNet-8	7	12.2M	85.33%	85.43%	$72 \cdot 10^{15}$
ResNet-14	13	26.4M	90.17%	90.32%	$20 \cdot 10^{30}$
ResNet-20	19	40.6M	91.77%	91.50%	$57 \cdot 10^{44}$
ResNet-32	31	68.9M	92.65%	92.58%	$45 \cdot 10^{74}$
ResNet-50	49	111.3M	92.88%	92.60%	$10 \cdot 10^{117}$
Resnet-56	55	125.5M	93.14%	93.11%	$28 \cdot 10^{131}$

817 The INT8 accuracy results are evaluated using the approxima-
 818 tion level 0 of the proposed multiplier, which provides exact
 819 results, for all the layers of each network. The multiplications
 820 reported in Table VIII are evaluated for the inference of one
 821 $32 \times 32 \times 3$ input image from the CIFAR-10 dataset. The design
 822 space size reported in the rightmost column is evaluated as
 823 the number of unique approximate layer-wise configurations,
 824 evaluated as A_x^L , with A_x being the number of approximation
 825 levels of the reconfigurable multiplier, in our case 256, and L
 826 the number of layers that can be approximated.
 827

828 For ResNet-8, ResNet-14, and ResNet-20, we ran the
 829 genetic search for 80 generations with a population of 70
 830 individuals, whereas for ResNet-32, ResNet-50, and ResNet-
 831 56, we increased the generations to 120. We set both the
 832 mutation probability P_m and the crossover probability P_c
 833 to 0.8. During the search phase, every approximate NN is
 834 retrained with 10% of the training set and the accuracy is
 835 evaluated with the validation set (5000 unseen images from the
 836 training set). We did not use the test set during the search phase
 837 as it would have biased the results and negatively affected the

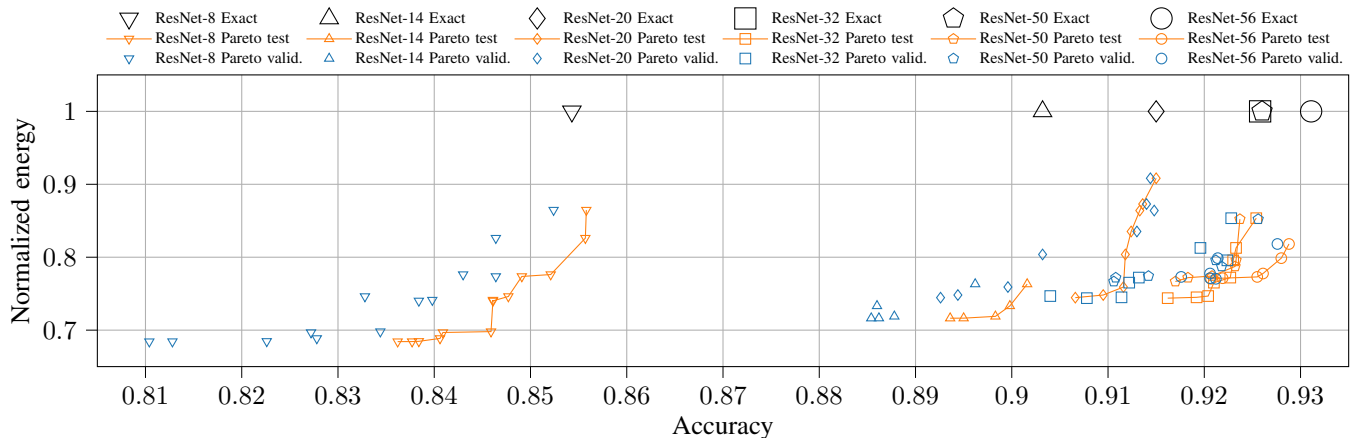


Fig. 9. Top 1% accuracy and normalized energy variation with different ResNet configurations. The *Pareto valid*. blue marks represent the validation accuracy evaluated during the genetic search, whereas the *Pareto test* orange marks represent the corresponding approximate configuration tested after the final retraining.

838 genetic algorithm. In this phase, the NN accuracy influences
 839 the evolution of each individual's configuration, i.e., the layer-
 840 wise approximation; therefore, to obtain NN models that can
 841 generalize on new unseen data and prove the effectiveness of
 842 our methodology, we removed any correlation with the final
 843 test results. Finally, once the Pareto front has been computed,
 844 the approximate NNs are retrained for one full epoch and
 845 evaluated using the test set.

846 The accuracy and energy results of only the dominant
 847 solution after the full retraining of the Pareto front are reported
 848 in Figure 9, providing a visual representation of the energy-
 849 accuracy trade-offs our methodology offers. For every NN
 850 under test, the absolute accuracy difference between *Pareto*
 851 *valid*. and *Pareto test* marks of each configuration is usually
 852 smaller than 1.5%, with even smaller values for deeper models.
 853 Therefore, the validation accuracy could represent a good ap-
 854 proximation of the final results, justifying the choice of using
 855 it in the proposed search strategy. Figure 10 depicts the utiliza-
 856 tion of approximation levels for each NN layer. It is possible
 857 to notice the presence of peaks around levels with an index
 858 equal to or smaller than $2^j - 1, j \in [0, 1, 2, 3, 4, 5, 6, 7, 8]$,
 859 corresponding to the Pareto optimal points of Figure 8. In
 860 Figure 10, it is shown how the majority of approximation
 861 levels are used in the Pareto front, justifying the choice to
 862 maintain power and MRED-dominated configurations as the
 863 most efficient levels might not be optimal ones to achieve a
 864 high task accuracy. Moreover, when concatenated appropri-
 865 ately, some approximation levels, even the Pareto-dominated
 866 ones, might mitigate the effect of computation errors on the
 867 final results, a strategy used in [20] to reduce the accuracy
 868 degradation. However, Figure 10 also highlights that some
 869 approximation levels are never used in this use case. Future
 870 development should add the possibility of pruning the search
 871 space removing unused solutions or those with the lowest uti-
 872 lization. In these experiments, to prove that our methodology
 873 is effective with an ample search space, low- and zero-usage
 874 solutions are deliberately kept to test the search algorithm in
 875 a worst-case scenario with the highest complexity.

876 Table IX compares our approach with ALWANN [15] to

TABLE IX
 COMPARISON WITH ALWANN [15] WITH 0.5% AND 1% RELATIVE
 ACCURACY DEGRADATION

Neural network	This work			ALWANN		
	Absolute accuracy	Relative accuracy	Energy	Absolute accuracy	Relative accuracy	Energy
ResNet-8 0.5%	85.21%	99.74%	77.62%	83.16%	99.88%	84.31%
ResNet-8 1%	84.59%	99.02%	69.80%	Same solutions as ResNet-8 0.5%		
ResNet-14 0.5%	89.98%	99.63%	73.32%	85.42%	99.85%	74.34%
ResNet-14 1%	89.50%	99.09%	71.64%	84.77%	99.09%	70.85%
ResNet-50 0.5%	92.14%	99.50%	80.67%	89.08%	99.92%	78.47 %
ResNet-50 1%	91.70%	99.03%	76.67%	88.58%	99.36%	70.02 %

877 understand how MARLIN stands against the state-of-the-art.
 878 To make a fair comparison, we compare approximate NNs
 879 with weights updated after a single epoch retraining for MAR-
 880 LIN and weight fine-tuning for ALWANN. Our method can
 881 achieve better results for shallower NNs such as ResNet-8 and
 882 maintain the same relative gains for ResNet-14, whereas it was
 883 not able to achieve higher energy efficiency than ALWANN for
 884 ResNet-50. Comparing the absolute accuracy of the NN mod-
 885 els, the approximate ResNet-14 within 1% relative accuracy
 886 degradation outperforms all the ResNet-50 models presented
 887 by ALWANN in top-1 accuracy and, by extension, in energy
 888 efficiency, as ResNet-14 has 76.3% fewer multiplications than
 889 ResNet-50. Our methodology is competitive, considering that
 890 the multipliers used in ALWANN have better area and power-
 891 MRED metrics, but are not reconfigurable [15], [28]. The
 892 main advantage of a reconfigurable multiplier against several
 893 arrays of fixed multipliers is that it is possible to improve
 894 the energy efficiency of arithmetic operations with lower area.
 895 This approach extended to a systolic array, would require a
 896 single array with the same multiplier architecture, whereas
 897 ALWANN requires N separate sub-arrays in order to support
 898 N approximation levels.

899 Table X compares MARLIN with the results presented in
 900 [17], without including absolute accuracy metrics, as they
 901 are not reported. We consider approximate NNs with one-
 902 epoch retraining for MARLIN, and approximate NNs with
 903 weight fine-tuning with and without additional bias for [17].
 904 Compared to the NNs with no additional bias, the approximate
 905 NNs configurations found with MARLIN require up to 13.1%

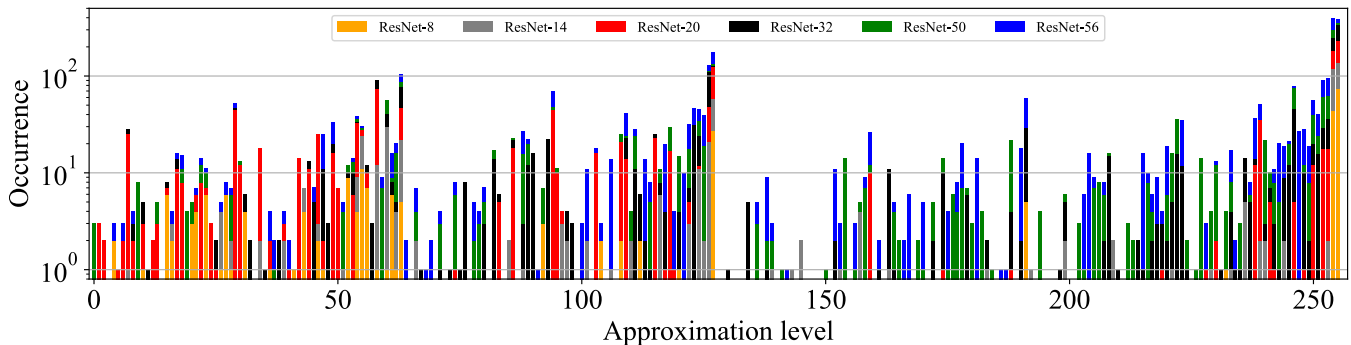


Fig. 10. Approximation levels utilization for all the configurations found for each ResNet model.

906 less energy for ResNet-20, up to 13% less energy for ResNet-
 907 32, and up to 15.1% for ResNet-56. When an additional error
 908 correction bias is added to the convolutional layer in [17],
 909 MARLIN can still achieve up to 9.8% less energy for ResNet-
 910 20, 8.6% for ResNet-32, and 7.3% for ResNet-56, without
 911 increasing the number of parameters and operations. Using
 912 more approximation levels proved to be an effective way to
 913 further reduce the inference energy, as we had 256 configura-
 914 tions against the 3 used in [17]. We justify these results with
 915 the traditional weight-update strategy used in this work and the
 916 presence of more approximate configurations. Retraining each
 917 configuration is slower and more computationally expensive
 918 than the multiplier-specific fine-tuning of [17], but allows a
 919 better adjustment of the NN parameters to compensate the
 computation errors, resulting in higher accuracy.

change, the number of computation cycles would also be
 unaffected. Table XI reports the area and the energy overheads
 of including and controlling the approximate multipliers in
 three accelerators, mapping the 19 convolutional layers of
 the ResNet-20 with 1% accuracy degradation and 74.46%
 energy of Table X. We used the power model in Timeloop
 [10] to evaluate the energy used to communicate to the PEs
 the approximation level of each layer, assuming one off-chip
 to on-chip memory transfer, and then #PEs transfers from
 the on-chip memory to the PEs' registers. The configuration
 energy of the entire NN is always below 0.002% of the total
 energy evaluated with Timeloop. The area overhead of 35%
 against exact multipliers can be negligible, considering that
 they account for less than 10% of the PE area in [8], [9].

TABLE X
 NORMALIZED ENERGY COMPARISON WITH [17] WITH 0.5%, 1%, AND 2%
 ACCURACY DEGRADATION

Neural network	Normalized energy		
	Ours	[17] w/o bias	[17] with bias
ResNet-20 0.5%	75.91%	86.1%	83.1%
ResNet-20 1%	74.46%	85.6%	83.0%
ResNet-20 2%	74.46%	85.1%	82.5%
ResNet-32 0.5%	77.21%	85.7%	81.7%
ResNet-32 1%	74.50%	85.7%	81.7%
ResNet-32 2%	74.39%	85.5%	81.4%
ResNet-56 0.5%	79.87%	94.0%	83.0%
ResNet-56 1%	77.12%	86.1%	83.0%
ResNet-56 2%	77.04%	86.1%	83.0%

TABLE XI
 MARLIN'S AREA AND COMMUNICATION OVERHEAD APPLIED TO OTHER
 HW ACCELERATORS

	Eyeriss [9]	DianNao [7]	Simba [8]
# PEs	256	256	1024
PE conf. comm. energy [pJ] (relative)	2138 0.001%	1997 0.001%	2369 0.002%
Mult. area exact [μm^2] (GE)	155468 (108k)	155468 (108k)	621875 (432k)
Mult. area approx. [μm^2] (GE) (relative)	210585 (146k) (+35%)	210585 (146k) (+35%)	842342 (586k) (+35%)

920
 921 **D. Compatibility with Other Accelerator Architectures**

922 Two important modifications are necessary to port MARLIN
 923 to other platforms: adapting the hardware architecture and the
 924 mapper to include the multiplier and the configuration instruc-
 925 tions. The former would require an additional 8-bit control
 926 signal from the PE control unit to set the approximation level.
 927 The elongated critical path delay due to the approximation
 928 logic could be a problem for some accelerators, but it is not
 929 for [8], [9], which have a critical path compatible with the
 930 proposed multiplier. For what concerns the mapper, recalling
 931 the discussion of Section II-A, a modification similar to what
 932 has been done in Section III-E can be implemented, inserting
 933 custom instructions to configure the multiplier, with negligible
 934 impact on the execution time. Since the scheduling does not

949 **E. Discussion**

950 The optimization approach of Section III-B allowed MAR-
 951 LIN to outperform previous works that relied on parameter
 952 fine-tuning [15], [17], leveraging partial retraining. MARLIN
 953 was run on a 32-thread Ryzen 5950X CPU with 64GB DDR4
 954 RAM and an Nvidia Quadro RTX A5000. The GPU was
 955 used only during the initial training of the FP32 and exact
 956 INT8 NNs presented in Table VIII, while the CPU was
 957 used to simulate the approximate convolutional layers during
 958 the training, validation, and test done during the search, as
 959 AdaPT only supports CPU computation [33]. The number of
 960 threads used during the computation was set to 16 for every
 961 experiment to compare how MARLIN execution time scales
 962 with different NNs depths. Table XII reports the execution time
 963 for the search phase and the training of the last Pareto front of
 964 Figure 9. On average, partial retraining is $\approx 6x$ faster than full
 965 retraining. An alternative implementation that leverages the
 966 GPU processing power, based on [47], is in development, with

the objective of reducing the search time with more complex NN models, allowing for a broader search space analysis. Compared to [15], the iteration time during the search phase is reduced by 72.8% for ResNet-8, 92.3% for ResNet-14, and 85.4% for ResNet-50. This speed-up is due to the increased utilization of CPU threads, as our training loop processes more images during each iteration compared to [15].

TABLE XII
MARLIN'S AVERAGE EXECUTION TIME WITH 16 THREADS

Neural network	Search phase		Final training	
	One iter.	Total	One iter.	Total
ResNet-8	6.8 sec.	10.6 hours	40.9 sec.	24.6 min.
ResNet-14	7.7 sec.	12 hours	79.4 sec.	35.7 min.
ResNet-20	19.6 sec.	30.5 hours	115.7 sec.	90.6 min.
ResNet-32	30.3 sec.	70.7 hours	189.0 sec.	81.9 min.
ResNet-50	47.1 sec.	109.9 hours	296.2 sec.	69.1 min.
ResNet-56	56.9 sec.	132.8 hours	329.2 sec.	93.3 min.

A limitation in finding the optimal trade-off between energy and accuracy is the dimension of the search space, which determines the search time and requires a carefully tuned search strategy. This problem is also found in mixed precision layer-wise quantization, in which the search space is q^{2L} , with q quantization levels for weights and activations, for L layers [11], [12]. Future work should focus on pruning the search space after a number of experiments (i.e., NSGA-II generations) to reduce its size, possibly reducing the time to converge. A further improvement over layer-wise approximation can be proposed by looking at past works on quantization. In AutoQ [3] channel-wise quantization is used to reduce the inference energy with less accuracy degradation than [11], [12], proving that NNs resilience to quantization errors has an intra-layer dependency besides the inter-layer one. Therefore, achieving an optimal energy-accuracy trade-off is possible by extending approximate computing in the channel dimension. In AutoQ [3], a bit-serial accelerator is required to support channel-wise quantization, whereas with MARLIN the only necessary modification, to enable it with the proposed RISC-V core, would be to adapt the CSR instructions inserted by our modified version of DORY. Nonetheless, the main challenge would be the efficient exploration of a wider search space.

V. CONCLUSION

In this paper, we presented MARLIN, a layer-wise approximation methodology leveraging a single multiplier architecture that can be configured runtime with 256 approximation levels to achieve an optimal trade-off between the inference energy and the task accuracy. In this work, a prototype based on a modified RISC-V core is proposed to test our methodology on a low-power IoT platform. The PULP toolchain has been adapted to automatically include the runtime approximation level selection alongside the instructions executed while processing convolutional layers. MARLIN can evaluate thousands of different NNs, leveraging NSGA-II to find the optimal configuration by generating a Pareto front that contains a set of layer-wise approximate NNs with reduced inference energy, without a significant accuracy loss.

REFERENCES

- M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current progress, research challenges, and future roadmap," in *58th ACM/IEEE Design Automation Conf. (DAC)*, pp. 1303–1306, 2021.
- M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for AI-enabled IoT devices: A review," *Sensors*, vol. 20, no. 9, p. 2533, 2020.
- Q. Lou, F. Guo, M. Kim, L. Liu, and L. Jiang, "AutoQ: Automated kernel-wise neural network quantization," in *Intl. Conf. on Learning Representations*, 2019.
- N. Fafous *et al.*, "AnaCoNGA: Analytical HW-CNN co-design using nested genetic algorithms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 238–243, 2022.
- A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Trans. of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.
- A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 186–191, 2020.
- T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, p. 269–284, 2014.
- Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM Intl. Symposium on Microarchitecture*, p. 14–27, 2019.
- Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.
- A. Parashar *et al.*, "Timeloop: A systematic approach to dnn accelerator evaluation," in *IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.
- K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *IEEE/CVF Conf. on Computer Vision and Pattern Recognition*, pp. 8604–8612, 2019.
- N. Fafous *et al.*, "HW-FlowQ: A multi-abstraction level HW-CNN co-design quantization methodology," *ACM Trans. Embed. Comput. Syst.*, vol. 20, sep 2021.
- H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *CoRR*, vol. abs/2004.09602, 2020.
- H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate arithmetic circuits: A survey, characterization, and recent applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020.
- V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, 2019.
- X. He, W. Lu, G. Yan, and X. Zhang, "Joint design of training and hardware towards efficient and accuracy-scalable neural network inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 810–821, 2018.
- Z.-G. Tasoulas, G. Zervakis, I. Anagnostopoulos, H. Amrouch, and J. Henkel, "Weight-oriented approximation for energy-efficient neural network inference accelerators," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4670–4683, 2020.
- İ. Taştan, M. Karaca, and A. Yurdakul, "Approximate CPU design for iot end-devices with learning capabilities," *Electronics*, vol. 9, no. 1, 2020.
- M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, "Improving the accuracy and hardware efficiency of neural networks using approximate multipliers," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 317–328, 2020.
- P. Jain, S. Huda, M. Maas, J. E. Gonzalez, I. Stoical, and A. Mirhoseini, "Learning to design accurate deep learning accelerators with inaccurate multipliers," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 184–189, 2022.
- PULP platform. Accessed: Jan. 23, 2023 [Online]. Available: <https://pulp-platform.org>.
- M. Gautschi *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multi-objective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[24] M. Pinos, V. Mrazek, and L. Sekanina, "Prediction of inference energy on cnn accelerators supporting approximate circuits," in *26th Intl. Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 45–50, 2023.

[25] E. Manor and S. Greenberg, "Using hw/sw codesign for deep neural network hardware accelerator targeting low-resources embedded processors," *IEEE Access*, vol. 10, pp. 22274–22287, 2022.

[26] E. Manor and S. Greenberg, "Custom hardware inference accelerator for tensorflow lite for microcontrollers," *IEEE Access*, vol. 10, pp. 73484–73493, 2022.

[27] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *IEEE/ACM Intl. Symposium on Low Power Electronics and Design (ISLPED)*, pp. 27–32, 2014.

[28] V. Mrazek, L. Sekanina, and Z. Vasicek, "Libraries of approximate circuits: Automated design and application in CNN accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 406–418, 2020.

[29] E. Attoofian, "Increasing robustness against adversarial attacks through ensemble of approximate multipliers," in *IEEE Intl. Conf. on Networking, Architecture and Storage (NAS)*, pp. 1–8, 2022.

[30] A. Siddique and K. A. Hoque, "Is approximation universally defensive against adversarial attacks in deep neural networks?," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, p. 364–369, 2022.

[31] PyTorch. Accessed: Mar. 11, 2023 [Online]. Available: <https://pytorch.org/>.

[32] F. Conti, "Technical report: NEMO DNN quantization for deployment model," 2020.

[33] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris, and J. Henkel, "AdaPT: Fast emulation of approximate DNN accelerators in PyTorch," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022.

[34] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs," *IEEE Trans. on Computers*, vol. 70, no. 8, pp. 1253–1268, 2021.

[35] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[36] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. on Computers*, vol. C-22, no. 12, pp. 1045–1047, 1973.

[37] M. de la Guia Solaz, W. Han, and R. Conway, "A flexible low power DSP with a programmable truncated multiplier," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 59, no. 11, pp. 2555–2568, 2012.

[38] T. Yang, T. Ukezono, and T. Sato, "A low-power high-speed accuracy-controllable approximate multiplier design," in *23rd Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pp. 605–610, 2018.

[39] F.-Y. Gu, I.-C. Lin, and J.-W. Lin, "A low-power and high-accuracy approximate multiplier with reconfigurable truncation," *IEEE Access*, vol. 10, pp. 60447–60458, 2022.

[40] A. G. M. Strollo, E. Napoli, D. De Caro, N. Petra, and G. D. Meo, "Comparison and extension of approximate 4-2 compressors for low-power approximate multipliers," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3021–3034, 2020.

[41] M. Ha and S. Lee, "Multipliers with approximate 4-2 compressors and error recovery modules," *IEEE Embedded Systems Letters*, vol. 10, no. 1, pp. 6–9, 2018.

[42] Z. Yang, J. Han, and F. Lombardi, "Approximate compressors for error-resilient multiplier design," in *IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 183–186, 2015.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conf. on computer vision and pattern recognition*, pp. 770–778, 2016.

[44] A. Krizhevsky, G. Hinton, et al., "Learning multiple layers of features from tiny images," *University of Toronto*, 2009.

[45] L. N. Smith, "Cyclical learning rates for training neural networks," in *IEEE winter Conf. on applications of computer vision*, pp. 464–472, 2017.

[46] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," 2018.

[47] E. Trommer, B. Waschneck, and A. Kumar, "High-throughput approximate multiplication models in PyTorch," in *26th Intl. Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 79–82, 2023.



Flavia Guella received the B.S. and M.S. (both with summa cum laude) in electronics engineering from Università degli Studi di Palermo in 2020, and Politecnico di Torino in 2023, respectively. She is currently pursuing the Ph.D. program in Electronics and Communications Engineering at Politecnico di Torino, under the supervision of Prof. Maurizio Martina and Guido Masera. Her research interests include co-design methodologies for the efficient deployment of neural networks on low-power systems and approximate computing.



Emanuele Valpreda received the B.S. and M.S. (summa cum laude) degrees in electronic and communications engineering from Politecnico di Torino, in 2017 and 2019 respectively. Now he is a Ph.D. student with the Electronics and Telecommunications Department on energy efficient neural network acceleration for edge computing, under the supervision of Prof. Maurizio Martina and Guido Masera. His current research interests include neural network compression, reliability and approximate computing.



Michele Caon is a Ph.D. student with the Electronics and Telecommunications Department, Politecnico di Torino, under the supervision of Prof. Maurizio Martina and Guido Masera. He received his B.S. and M.S. (summa cum laude) at Politecnico di Torino in 2017 and 2019. His research interests are innovative digital, integrated, programmable computing circuits and systems. He is currently working on near-memory computing circuits embedded in heterogeneous systems-on-chip.



Guido Masera (SM'07) received the Dr.-Ing. (summa cum laude) and Ph.D. degrees in electronic engineering from Politecnico di Torino, Italy. He is a Professor with the Electronics and Telecommunications Department, Politecnico di Torino, since 1992. His research interests include several aspects in the design of digital integrated circuits and systems, with a special emphasis on high-performance architectures for communications, forward error correction, image and video coding, cryptography and hardware accelerators for machine learning. He has more than 200 publications, two patents and was a designer of several ASIC components. Dr. Masera is an Associate Editor of MDPI Electronics and a former Associate Editor of the IEEE Transactions on Circuits and Systems I, IEEE Transactions on Circuits and Systems II and the IET Circuits, Devices & Systems.



Maurizio Martina received the Dr.-Ing. and Ph.D. degrees in electronic engineering and electronic and communications engineering from Politecnico di Torino, Italy, in 2000 and 2004, respectively. He is a Professor with the Electronics and Telecommunications Department, Politecnico di Torino, since 2014. His research interests include computer architecture and VLSI design of digital integrated circuits for image and video coding, forward error correction, cryptography and artificial intelligence. He has more than 100 publications and holds two patents. He served as an Associate Editor of the IEEE Transactions on Circuits and Systems—I and as a Guest Editor of several special issues, including BioCAS 2017 special issue in IEEE Transactions of Biomedical Circuits and Systems and ISCAS 2023 special issue in IEEE Transactions on Circuits and Systems-II. He has been part of the organizing and technical committee of several IEEE conferences, including BioCAS 2017, AICAS 2020 and PRIME 2023.