Hardware-Aware Evolutionary Approaches to Deep Neural Networks*

Lukas Sekanina and Vojtech Mrazek and Michal Pinos

Abstract This chapter gives an overview of evolutionary algorithm (EA) based methods applied to the design of efficient implementations of deep neural networks (DNN). We introduce various acceleration hardware platforms for DNNs developed especially for energy-efficient computing in edge devices. In addition to evolutionary optimization of their particular components or settings, we will describe neural architecture search (NAS) methods adopted to directly design highly optimized DNN architectures for a given hardware platform. Techniques that co-optimize hardware platforms and neural network architecture to maximize the accuracy-energy trade-offs will be emphasized. Case studies will primarily be devoted to NAS for image classification. Finally, the open challenges of this popular research area will be discussed.

* This is an Author Accepted Manuscript version of the following chapter: Sekanina, L., Mrazek, V., Pinos, M.: Hardware-Aware Evolutionary Approaches to Deep Neural Networks, published in Handbook of Evolutionary Machine Learning, edited by Wolfgang Banzhaf, Penousal Machado, Mengjie Zhang, 2024, Springer, reproduced with permission of publisher. The final authenticated version is available online at: https://doi.org/10.1007/978-981-99-3814-8_12

Lukas Sekanina
Brno University of Technology, Faculty of Information Technology, Czech Republic, e-mail: sekanina@fit.vutbr.cz

Vojtech Mrazek
Brno University of Technology, Faculty of Information Technology, Czech Republic, e-mail: mrazek@fit.vutbr.cz

Michal Pinos
Brno University of Technology, Faculty of Information Technology, Czech Republic, e-mail: ipinos@fit.vutbr.cz

1 Introduction

Previous chapters have shown that Evolutionary Algorithms (EA) can be utilized for neural architectures search (NAS) and for solving various hard optimization problems in the scope of machine learning applications. This chapter is devoted to the use of evolutionary algorithms in the task of discovering high-quality *implementations* of deep learning algorithms. We will focus on deep neural networks (DNN) and their distinct subclass – *convolutional neural networks* (CNN) – that are currently employed as machine learning engines in many challenging applications operated on different types of platforms, ranging from ultra-low-power edge devices via mobile phones to high-performance accelerators in data centers [68, 28, 8]. Hence, in addition to producing high-quality outputs, many of these implementations have to be energy efficient. This is achieved by developing specialized neural architectures and hardware inference accelerators for CNNs (and DNNs in general).

Section 2 introduces the principles of efficient processing of CNNs in specialized hardware. It describes two fundamental architectures of CNN accelerators that are currently used – *temporal architecture* which is typical for common processors and Graphic Processing Units (GPUs), and *spatial architecture*, often adopted in application-specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs). Particularly we emphasize the role of *mapping*, i.e. the strategy determining how a computational graph of (a potentially very complex) CNN is executed on limited hardware resources available on a chip. This chapter also deals with hardware simulators and fast predictors of hardware parameters of CNNs.

In Section 3, we start with a fully trained CNN model and discuss various optimizations that can be conducted by EAs to obtain its hardware implementation with desired properties. We focus on the evolutionary design of components (such as approximate multipliers and activation functions) of hardware accelerators, optimized precision scaling, evolutionary optimization of the CNN-to-accelerator mapping, and weight compression.

Section 4 is devoted to *hardware-aware NAS methods* and *NAS methods with hardware co-design*. Considering a given task (e.g., image classification) and target hardware, the *hardware-aware NAS* algorithms try to deliver the most suitable CNN architecture whose hardware implementation satisfies given constraints, e.g. on maximum latency. Note that the hardware platform is not directly optimized; it can be seen as a series of constraints for the NAS method. *NAS with hardware co-optimization* evolves CNN architecture and configuration of a configurable hardware accelerator in parallel, i.e., in addition to the space of CNN architectures, it optimizes hardware configuration (e.g., type of used resources, mapping strategies, buffer sizes, and compiler options). We survey the key evolutionary NAS methods addressing the above-mentioned approaches.

Finally, conclusions and open research challenges are presented in Chapter 5. Specifically, we address the problem of benchmarking of hardware-aware NAS methods, security & reliability issues, novel unconventional hardware platforms for DNNs, and design time reduction.

2 Hardware platforms for efficient processing of DNNs

The efficient processing of deep neural networks has been addressed in the literature in great detail, including a prominent book [69] and comprehensive surveys [8, 46, 51, 61]. This section aims to explain the basic concepts of efficient processing of CNNs and briefly survey hardware accelerators introduced for inference. First, we recall the principles and terminology of CNNs in Section 2.1. We distinguish two main architectures – temporal architecture (Section 2.2) and spatial architecture (Section 2.3). In Section 2.4, we will also deal with simulators and performance predictors developed to simplify the hardware accelerator design process.

2.1 Convolutional layers

A typical CNN consists of convolutional layers, pooling layers, fully connected layers, and some other less computationally intensive units. Convolutional layers are responsible for more than 90% of overall computation, dominating runtime and energy consumption of inference [69]. Figure 1 illustrates how the output, the so-called *output feature maps* (**O**), of a convolutional layer, are obtained. The *input feature maps* (**I**), holding either the input image or an intermediate result of a previous layer, are processed by applying a set of *filter weights* (**Weights**). As multiple input feature maps can be processed in parallel, multiple output feature maps are obtained, where **N** is their number; **N** also denotes the batch size. Table 1 summarizes all symbols used.

Table 1 Symbols used to describe convolution layers.

Symbol	Description
H	- Input Feature Map Height
W	- Input Feature Map Width
C	- Number of Input Channels
R	- Filter Height
S	- Filter Width
M	- Number of Output Channels
E	- Output Feature Map Height
F	- Output Feature Map Width
N	- Number of Input/Output Feature Maps
U	- Stride

A straightforward software implementation of the convolutional layer operation is depicted in Algorithm 1. The core operation is the so-called Multiply-And-Accumulate (MAC) operation. It multiplies a weight with an input activation and adds the product to a partial sum. Fig. 2 shows its basic circuit implementation, including the number of bits for each signal when a fixed-point (FX) number rep-

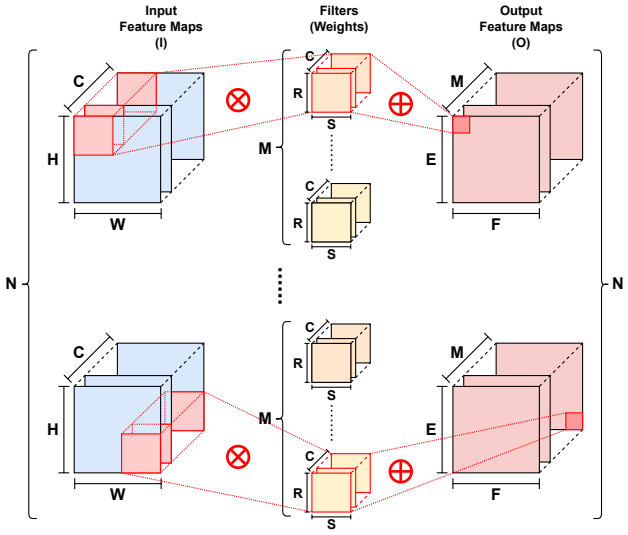

Fig. 1 Applying a set of filters on the input feature maps to calculate the output feature maps in convolutional layers. Symbols are defined in Table 1.

resentation is utilized. The $2N$-bit product is added to a $2N+M$ bit partial sum, where M depends on the number of weights. The result is quantized into N bits. The floating-point (FP) number representation is typically used for training.

Algorithm 1: Generalized convolution in CNNs

```
1  for ( n = 0; n < N; n++ ) {
2      for ( m = 0; m < M; m++ ) {
3          for ( x = 0; x < F; x++ ) {
4              for ( y = 0; y < E; y++ ) {
5                  O[n][m][x][y] = 0;
6                  for ( i = 0; i < R; i++ ) {
7                      for ( j = 0; j < S; j++ ) {
8                          for ( k = 0; k < C; k++ ) {
9                              O[n][m][x][y] +=
                                   I[n][k][Ux+i][Uy+j] × Weight[n][k][i][j];
10                 O[n][m][x][y] += B[m];
11                 O[n][m][x][y] = Activation(O[n][m][x][y]);
```

Computing the resulting output feature maps of convolutional layers and fully connected layers is usually transformed into matrix multiplications. The key factors determining how much time and energy will one inference require are the size of

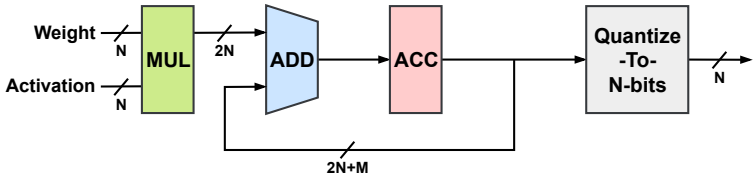

Fig. 2 A typical implementation of the Multiply&Accumulate (MAC) circuit utilizing the N-bit fixed point number representation for the weights.

these matrices, resources available on a given hardware platform, and a data flow control algorithm. Since a moderate CNN can have millions of parameters, these matrices do not fit the local memory of the accelerator. Hence, they must be processed at the block level, where a block is a submatrix that can be stored in local memory. These blocks are sent from the main (external) memory to local memory, then read and processed using arithmetic units (in MAC circuits). The partial results are stored in local memory or copied into the main memory when needed.

2.2 DNN accelerators: Temporal architecture

Hardware accelerators with *temporal architecture* employ a set of Arithmetic Logic Units (ALUs) with a fixed connection pattern and a hierarchical memory subsystem. General-purpose CPUs and GPUs are typical examples of temporal architectures. The ALU always receives data from local memory and returns the result to the same memory. As there is minimal support for data reuse, i.e., direct data sending between multiple ALUs, memory access becomes the main performance and energy bottleneck. Libraries such as MKL [25] and cuDNN [12] provide highly-optimized matrix multiplication and other algorithms for CPUs and GPUs.

GPUs consist of hundreds to thousands of lightweight processing cores with a high-throughput memory subsystem organized into a high-performance single-instruction multiple data (SIMD) programmable stream architecture. Hence, GPUs are useful for the parallelization of matrix multiplication and other operations conducted on FP data types during DNN training and inference. GPUs range from small devices (e.g., NVIDIA Jetson Nano with 472 GFLOPS and 5-10 W) to high-performance nodes of supercomputers (e.g., NVIDIA V100 with 100 TFLOPS and 300 W).

Compared to GPUs, standard CPUs equipped with a few cores offer limited options for DNN acceleration. However, the data-level parallelism provided by SIMD instructions (SSE, AVX) are often exploited. A detailed survey of CPU-based DNN acceleration techniques is provided by Mittal at al. [47]. Low-cost microcontrollers (MCU) are typically employed to implement DNNs on low-power edge devices. Their instruction set can be enhanced with specialized instructions to accelerate MACs and, thus, convolutions. For example, RI5CY is an open MCU class RISC

V Core for energy-efficient processing of Quantized NNs (QNN) utilizing specialized SIMD instructions, fast dot product unit, and multiple FX data formats such as INT-2, INT-4, INT-8, INT-16, INT-32 [21]. To provide DNNs on low-power processors, GAP-8 programmable chip featuring an 8-core cluster composed of RISC-V processors, cache, and other components was developed [20].

As programming of CPUs and GPUs does not require hardware design skills, the DNN accelerators based on CPUs and GPUs are, in principle, more accessible to a broader spectrum of designers than specialized ASICs.

2.3 DNN accelerators: Spatial architecture

The spatial architectures, typically implemented in application-specific integrated circuits or in the field programmable gate arrays, employ an array of many locally communicating processing elements (PE), see Fig. 3. Each of them implements a MAC circuit, a small local memory (registers), and a controller. PEs are usually organized as pipelined systolic arrays optimized for fast execution of DNN operations. Hence, a PE can directly send its output to other PE(s), which leads to faster and energy-efficient computing of DNN operations, eliminating thus memory accesses. The used on-chip network determines connection options among the PEs. The execution time and energy of a DNN accelerator are thus primarily determined by the PE size, memory subsystem, on-chip network, and the so-called data flow organization.

Data flow is a general term covering the computation order and parallelization strategy applied in the accelerator. It defines the order of arithmetic operations and memory accesses to maximize data reuse. The term *mapping* refers to the dataflow strategy (i.e., computation order and parallelism strategy) coupled with the tiling strategy (selecting the size of input data with respect to available hardware resources such as buffers and PEs). A particular mapping of a given DNN on hardware resources and executing this mapping is implemented by the accelerator controller.

ASIC accelerators for DNNs (as surveyed [69, 8]) share the implementation ideas discussed in the previous paragraph. However, their various implementations differ in many aspects, including the fabrication technology, maximum operation frequency, bit precision, the size (of the PE array and on-chip memory), interconnection network, dataflow algorithm, support for weight compression, etc.

These accelerators often utilize the principles of *approximate computing* to provide the best trade-offs between inference accuracy and other objectives (performance, energy). Common approximation techniques are precision scaling, employing approximate arithmetic operations, network pruning, and approximate memory [45]. The most significant gains are obtained when the cross-layer approximation approach is adopted, involving software, architecture, and hardware, breaking thus conventional methods focused on optimizing each layer of abstraction independently [72]. Recent ASIC accelerators also employ the principles of *in-memory computing* to alleviate the data communication bottleneck between PEs and memory

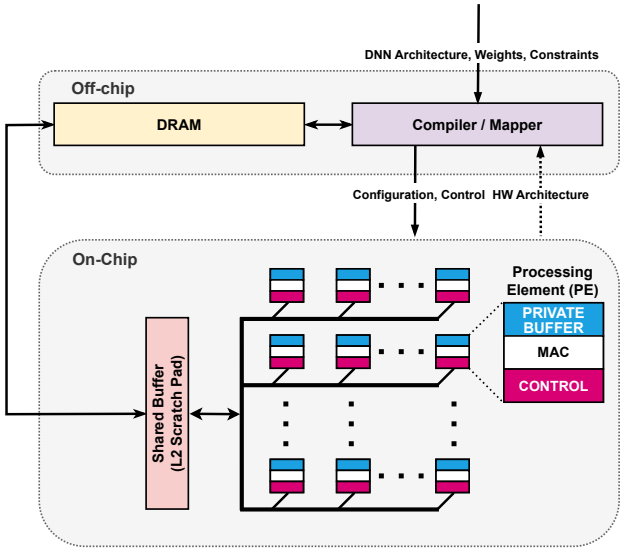

Fig. 3 Generic programmable hardware spatial accelerator for DNN inference. Processing elements form a two-dimensional systolic array enabling highly-efficient parallel computing and data reuse. The weights are stored in the external DRAM memory; some can be cached on the chip. Based on the DNN architecture description, hardware configuration, and other constraints, the compiler (mapper) generates the mapping of the DNN on hardware resources and its execution plan.

elements. In-memory computing aims to extend typical memory architecture with the capability of performing some arithmetic operations to accelerate data processing [66].

Google's Tensor Processing Unit (TPU) family is an example of DNN accelerators implemented as ASICs [28]. TPUv1, introduced in 2016, provided a systolic array of 256 $\times$ 256 8-bit FX multipliers allowing significantly accelerated matrix multiplications for CNN inference (with the peak performance 92 TOPS at 75 W). TPUv2 and TPUv3 offer increased performance and support FP operations which make them usable for DNN training. EdgeTPU, with a peak performance of 4 TOPs and 2 TOPs/W was developed for edge computing and smartphones. It is programmed using TensorFlow Lite models.

FPGAs have traditionally been seen as programmable arrays of logic blocks whose function is defined by means of look-up tables (LUT) and whose interconnection is based on programmable switches. However, modern FPGAs are heterogenous systems containing not only programmable logic but also embedded memories (BRAM), processors, programmable interfaces, and other specialized circuits (the so-called hard blocks) such as configurable digital signal processing (DSP) blocks. The DSP blocks are especially useful for accelerating the convolution operations of DNNs. Designers can currently choose from various models ranging from small Xilinx Zynq chips suitable for IoT nodes to complex systems on a chip such as Xil-

inx Versal integrating programmable logic for flexible parallel compute-intensive tasks, processors for sequential processing tasks, and vector processors for domain-specific parallel data processing [61, 46].

Table 2 summarizes the key parameters of selected platforms when programmed to accelerate the AlexNet inference. While ASIC provides the most energy-efficient DNN processing (see the Efficeincy column), Titan X GPU shows the highest performance (see the Performance column). The architecture of a given chip, fabrication technology, and operational frequency are primarily determining these properties.

Table 2 Performance and energy-efficiency of AlexNet on various platforms (composed using [11, 79]).

Platform	Chip	Freq. [MHz]	Precision	Perform. [inference/s]	Power [W]	Efficiency [inference/s/W]
ASIC	Eyeriss	200	FX16	34.7	0.3	124.8
FPGA	Kintex KU115	235	FX8	2252	22.9	98.3
FPGA	Kintex KU115	235	FX16	1126	22.9	49.2
FPGA	Zynq XC7Z045	200	FX8	340	7.2	47.2
FPGA	Zynq XC7Z045	200	FX16	170	7.2	23.6
GPU	Jetson TX2	1 300	FP16	250	10.7	23.3
GPU	Titan X	1 417	FP32	5120	227.0	22.6
CPU	Core-i7	3 500	FP32	162	73.0	2.2

2.4 Hardware simulators and performance predictors

Suppose we have a CNN model (i.e., a computation graph) and need to know the hardware parameters (e.g., latency and energy) of its potential implementation on a given accelerator. However, there are usually many options on how to configure a given accelerator, map the CNN on the available resources, and schedule the CNN processing.

To choose the most suitable implementation, a search has to be conducted in the space of possible mappings and hardware configurations. The objective is to minimize the inference time (latency), energy, or other parameters. From Table 3, presenting the cost of typical operations conducted on a chip (when a 45 nm technology is considered), one can conclude that (1) minimizing the access to external memory has to be optimized with the highest priority, and (2) optimizing the bit width saves some energy not only when arithmetic operations are conducted but also when data are moved to/from memory; moreover, shorter weights will reduce the memory size.

Hardware parameters of a given CNN implementation are usually obtained using the following methods:

Table 3 Energy needed to perform selected operations on a chip fabricated in 45 nm technology [69].

Operation	Type	Width	Energy
[-]	[-]	[bits]	[pJ]
Add	Integer	8	0.003
		16	0.005
		32	0.1
	Float	16	0.4
		32	0.9
Multiply	Integer	8	0.2
		32	3.1
	Float	16	1.1
		32	3.7
Read	SRAM	32	**5**
	DRAM	32	**640**

- Measuring a real hardware implementation provides exact values; however, it is time-consuming to build and measure real hardware.
- Simulation using precise hardware simulators can provide accurate results, but it is still time-consuming when conducted at the gate level.
- Analytical estimation consists of analyzing the CNN's computational graph and applying precomputed knowledge about the cost of particular operations on given hardware [6].
- Building a surrogate model capable of predicting a given hardware parameter. It requires selecting suitable features for the predictor and collecting annotated data. Various machine learning models have been utilized for this purpose, e.g., linear regression [71], neural network [80], Gaussian process [37], and Bayesian Ridge Regression [78].

Predicting the resulting latency using easy-to-obtain properties of DNNs, such as the number of weights or MACs, is highly unreliable [69]. However, a recent paper shows that one carefully constructed proxy model (predictor) is enough for hardware-aware NAS [37]. This is also documented by detailed benchmarking [31], disclosing that the inference latency and energy of CNN architecture on hardware are strongly correlated. It concludes that an energy constraint can be implicitly mapped to a corresponding latency constraint in NAS methods.

To illustrate the complexity of the mapping optimization, we consider ResNet-50 CNN [23], which should be implemented on Eyeriss and Simba accelerators. Tools such as Accelergy and Timeloop help in determining the most suitable mapping. Accelergy [75] is an early-stage energy and execution time estimation tool. It estimates hardware parameters of a CNN implementation on a given accelerator whose organization is described at the architecture level in the YAML language, i.e., using characteristics such as the number of PEs, memory size, on-chip network, and data

flow organization. Plug-ins for different fabrication technologies can be integrated. Timeloop [53] is a tool searching for the most suitable mapping of CNN to hardware accelerator (several search methods are available in the tool). It performs CNN layer-wise data tiling reflecting the memory hierarchy of the accelerator.

As an example, Fig. 4 shows a distribution of the energy efficiency of half a million randomly generated mappings for the first convolutional layer of the ResNet-50 network. Timeloop conducted this estimation for two accelerator models: Eyeriss, featuring 168 PEs, and Simba, equipped with 16 PEs. Both models were simulated using 45 nm fabrication technology and utilized external weight memory (LPDDR4). Note that the considered layer performs convolution over the $224\times224\times3$ input feature map, with 7×7 filters, stride 2, and 64 output channels. The energy efficiency of mappings exhibits substantial variations. These variations stem from the diverse options for tiling and scheduling, which are represented by different mappings. The more efficient mappings excel in utilizing buffer capacity, network subsystems, and loop ordering to maximize data reuse. Nevertheless, the optimal mapping is subject to change based on the workload. What may be an optimal mapping for one architecture could prove to be suboptimal or even invalid for another architecture.

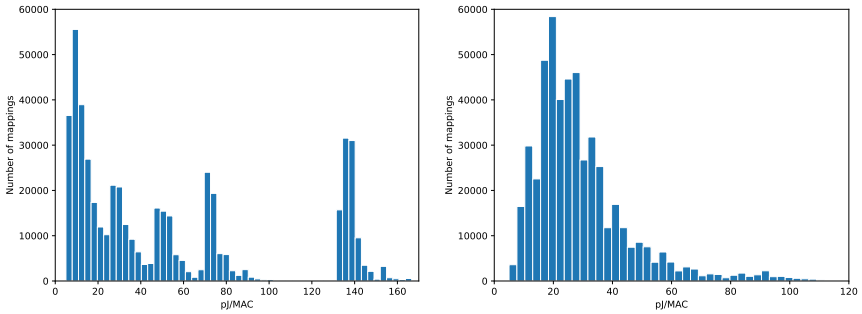

Fig. 4 The distribution of mappings showcasing energy efficiency (pJ/MAC) for the first layer of ResNet-50. The mappings were generated by Timeloop for the Eyeriss accelerator (left) and the Simba accelerator (right).

3 Evolutionary optimization in DNN hardware accelerators

Surveys [61, 44] document the gradually increasing interest in optimizing hardware implementations of fully trained CNNs (i.e., the inference accelerators), especially in the context of edge computing. In this chapter, we present approaches utilizing EAs for this purpose. Our primary focus will be on elucidating the rationale behind employing EAs in each design or optimization problem and discussing the methodologies employed.

3.1 Evolutionary design of components of hardware accelerators

Elementary components of CNN accelerators such as various arithmetic operations can be evolved and optimized to improve hardware parameters of these accelerators, and in some cases, the accuracy, too. Genetic programming (GP) is almost exclusively used as the evolutionary method. The fitness function reflecting the functionality of candidate designs is based on either (1) applying a candidate solution in a CNN, training the CNN, and interpreting the CNN's error as the fitness value, or (2) comparing the functionality of a candidate solution with a reference implementation of a given component on some data and determining the fitness as, e.g., a mean squared error. While approach (1), in principle, leads to more reliable solutions (the entire CNN is evaluated), approach (2) is computationally less expensive. In this context, typical targets for the evolutionary approach are multipliers, MACs, and activation functions.

3.1.1 Approximate multipliers

The inexact (approximate) multipliers provide inexact products; however, this inexactness can be tolerated because CNNs are often highly error-resilient [60]. In addition to reducing the bit width of multipliers used in MAC units, the approximation can be achieved by simplifying the logic equations specifying the product. The task is to design approximate multipliers showing good trade-offs between the accuracy and hardware parameters (such as energy and latency). In addition to many manual approximation methods, a fully automated circuit approximation methodology based on Cartesian genetic programming (CGP) has been developed [70, 50]. A common strategy is to optimize the multiplier with respect to an exact multiplier. In the fitness function, the error is expressed using error metrics such as the worst-case error or the mean absolute error. If a multiplier showing a good trade-off between the error and hardware parameters is discovered by CGP, it is used instead of the exact multipliers in one or several layers of a CNN. The CNN's accuracy is then determined, typically after a short fine-tuning. Based on the final accuracy, the evolved multiplier is accepted or rejected. This two-step design process is adopted because many candidate multipliers have to be generated, and evaluating each of them directly in the final CNN is very time-consuming.

Every candidate approximate multiplier $\widetilde{M}$, which is generated by a gate-level CGP, has two inputs (n and m bits) and produces a $n+m$ bit output [70]. The objective is to minimize the cost of the circuit (which highly correlates with power consumption) assuming that $\widetilde{M}$ shows the worst-case error (WCE) at most ε (Eq. 1):

$$F(\widetilde{M},\varepsilon) = \begin{cases} cost(\widetilde{M}) & \text{if } WCE(\widetilde{M}) \leq \varepsilon \wedge \\ & WCE_{zr}(\widetilde{M}) = 0 \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

The cost is estimated as the sum of the weighted areas of the gates used in the circuit. As the approximate multipliers are supposed to be used in neural networks, the requirement for accurate multiplying by zero ($WCE_{zr}(\widetilde{M}) = 0$) is integrated together with the WCE constraint in the fitness function. The validity of both conditions is checked using a single pass of exhaustive simulation of each candidate multiplier. At the end of evolution, the best-scored circuit is synthesized to get all its hardware parameters. The resulting approximate multiplier is also used in a given CNN to obtain classification accuracy, which is typically worsened in comparison with the CNN utilizing exact multipliers. However, the accuracy has usually recovered after retraining for a few epochs.

The case study reported in [50] deals with a situation in which all 8-bit multiplications of all convolutional layers of ResNet CNNs are replaced with one particular approximate implementation of the multiplier. Various evolved approximate 8xN-bit multipliers that are available in the EvoApproxLib [48] are tested. One operand (the activation) is always at 8 bits and the second operand (the weight) is on N bits, where $N = \{4,5,6,7,8\}$. Optimizing the bit width leads not only to smaller circuits but also to the reduced size of weight memory. Fig. 5 shows trade-offs between accuracy and energy of multiplication when ResNet-26 uses various approximate multipliers in its convolutional layers. For a small drop in accuracy, a 50% energy reduction of multipliers is obtained if a suitable approximate multiplier is used. Results are given for the 45 nm process and power supply voltage V_{dd} =1 V. The same approach was taken to design approximate MAC circuits in authors' work [9].

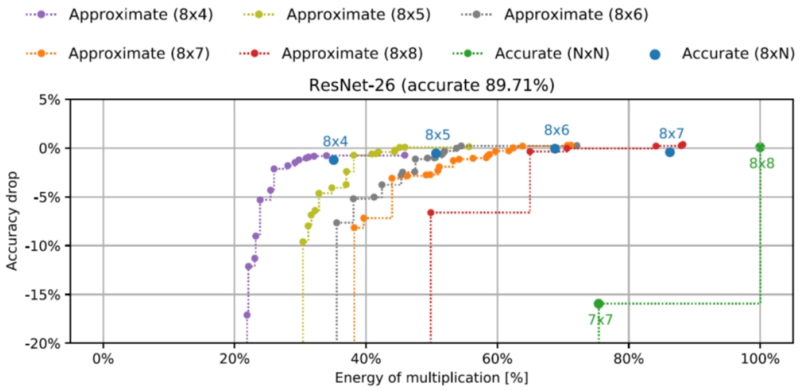

Fig. 5 The energy-accuracy trade-offs when all exact 8-bit multiplications of all convolutional layers of ResNet-26 CNN are replaced with an approximate multiplier taken from the EvoApproxLib library of 8xN-bit approximate multipliers (35 different multiplier implementations tested).

3.1.2 Activation functions

There are some frequently used activation functions such as ReLU or Sigmoid. However, better activation functions can be obtained for particular data sets and CNN architectures. EAs are thus employed to either deliver new functions (without considering their hardware implementation) [4, 65, 52] or optimize existing functions (with respect to resources) [58]. A common approach to evolve a new activation function is to employ a tree-based genetic programming (GP) which typically utilizes the function set consisting of hardware unfriendly functions (atan, tanh etc.). The fitness is based on evaluating the entire CNN in which a candidate activation function is embedded [4, 52].

For example, Lapid and Sipper [30] employs co-evolution to evolve activation functions for image-classification tasks using CGP. The function set comprises ten commonly used activation functions (ReLU, tanh, ELU, etc.) and 5 arithmetic operations ($+, -, \times$, minimum, and maximum). A cooperative coevolution algorithm evolves input-layer, hidden-layer, and output-layer activation functions (each having a separate population). An individual's fitness is determined by the activation function's ability to cooperate with members of the other populations (the fitness procedure is elaborated in [30]). On four classification datasets (MNIST, Fashion-MNIST, KMNIST, USPS) and two neural networks (a 7-layer MLP and an 8-layer CNN), the method was capable of improving the classification accuracy compared to the reference solution.

A hardware-aware evolutionary design of activation functions is conducted by Prashanth and Madhav [58]. Ordinary activation functions (sigmoid, tanh, Gaussian, ReLU, GeLU, Softplus) are considered as golden solutions, and their gate-level implementations are evolved for a given bit width using CGP. In the fitness function, a fully functional solution specified by a truth table is sought in the first step. Once it is obtained, its size is minimized in the second step.

3.1.3 Component selection and precision scaling

Suppose that an 8-bit CNN that has to be accelerated consists of more layers (neurons) than processing units available in the accelerator. Furthermore, approximate multipliers can be utilized in configurable processing units. Two tasks have to be solved together: (1) the assignment of the approximate multipliers to MACs of the processing units and (2) the assignment of the convolutional layers to the processing units. ALWANN is an optimization tool capable of selecting a suitable approximate multiplier for each processing unit in such a way that one approximate multiplier serves several layers, and the overall classification error and energy consumption are minimized [49]. The optimizations, including the multiplier selection problem, are solved by means of the NSGA-II algorithm in which the overall CNN accuracy and the energy consumed by the approximate layers are considered. Each candidate solution is uniquely defined by a pair of mappings (map1, map2), where map1 is a list of k integers in which each integer represents an approximate multiplier (taken

from EvoApproxLib) assigned to processing units $1 \ldots k$, and map2 is another list of l integers in which each integer determines the index of a processing unit that will be used to compute the output of the layer $1 \ldots l$. Additional restrictions may be applied depending on the chosen HW accelerator's structure. In order to altogether avoid the computationally expensive retraining of CNN, which is usually employed to improve the classification accuracy, a simple weight updating scheme is proposed that compensates for the inaccuracy introduced by employing approximate multipliers. ALWANN is evaluated for two architectures of CNN accelerators with approximate multipliers from the open-source EvoApproxLib library while executing three versions of ResNet on CIFAR-10. ALWANN saves 30% of energy needed for multiplication in convolutional layers of ResNet-50 while the accuracy is degraded by only 0.6% (0.9% for the ResNet-14).

Barone et al. [1] propose E-IDEA, an automatic application-driven approximation tool targeting different implementations (hardware and software). E-IDEA uses Clang-Chimera tool to analyze the Abstract Syntax Tree (AST) of the application's source code. Through the so-called mutators, approximations can be introduced at the source code level. The set of mutators includes loop-perforation mutators, precision-scaling mutators for floating-point arithmetic, a precision-scaling mutator for integer arithmetic, and a mutator supporting approximate arithmetic operator models of circuits being part of the EvoApproxLib library. An evolutionary approximation method based on NSGA-II tries to find the best approximation version of a given C/C++ code according to user-defined optimization objectives. A candidate solution is represented as a vector of integers; each of them corresponds to one parameter that can be modified. A set of matching rules specifies the positions in the source code at which a mutation can be applied. E-IDEA was used to approximate weighted sums computed within neurons to reduce hardware requirements and power consumption. Clang-Chimera was configured to truncate input operands and results of multiplications in the three convolutional and the two fully connected layers of the considered network (LeNet). Thus, the tool generates an approximate version of the considered CNN in which it is possible to configure the number of approximate bits to tune the introduced approximation degree for each multiplication involved in the weighted sum. Moreover, Clang-Chimera could select a suitable approximate multiplier from a library of approximate multipliers in these layers. NSGA-II optimized the CNN error on MNIST, aiming at reducing the circuit area. This allowed finding solutions to achieve more than 30% savings, with a negligible accuracy loss (0.48%) compared to the reference solution.

3.1.4 The CNN-to-hardware mapping optimization

In the GAMMA (Genetic Algorithm-based Mapper for ML Accelerators) framework, configurable hardware accelerators are considered, i.e., computation order, parallelizing dimensions, and tile sizes can be configured at compile-time [29]. For given constraints (the maximum number of parallelism levels and maximum tile sizes), GAMMA is searching for the most suitable mapping of a CNN layer (see

Alg. 1) on the hardware resources modeled in MAESTRO [42]. For a given CNN layer, hardware configuration (the number of PEs, local buffer size, global buffer size, latency, and bandwidth), and a mapping strategy, MAESTRO estimates the statistics such as latency, energy, runtime, power, and area.

The mapping is composed of several levels. Each level represents parallelism across a spatial dimension of the accelerator. Fig. 6 shows how a two-level mapping is encoded in the chromosome. The convolutional layer is specified using C input channels, K output channels, input activations of size $X \times Y$, output activations of size $X' \times Y'$, and filers of size $R \times S$. Each dimension is encoded using seven pairs of values. A pair of genes contains a CNN layer tensor notation (e.g., K, C) and its tile size. The ordering of pairs specifies the computation order. The first pair defines the parallelizing dimension. The L1-mapper describes the inner loop. The L2-mapper describes the outer loop, while containing PL1 number of instances of L1-mapper. The chromosome is used to create a candidate mapping which is then evaluated in MAESTRO.

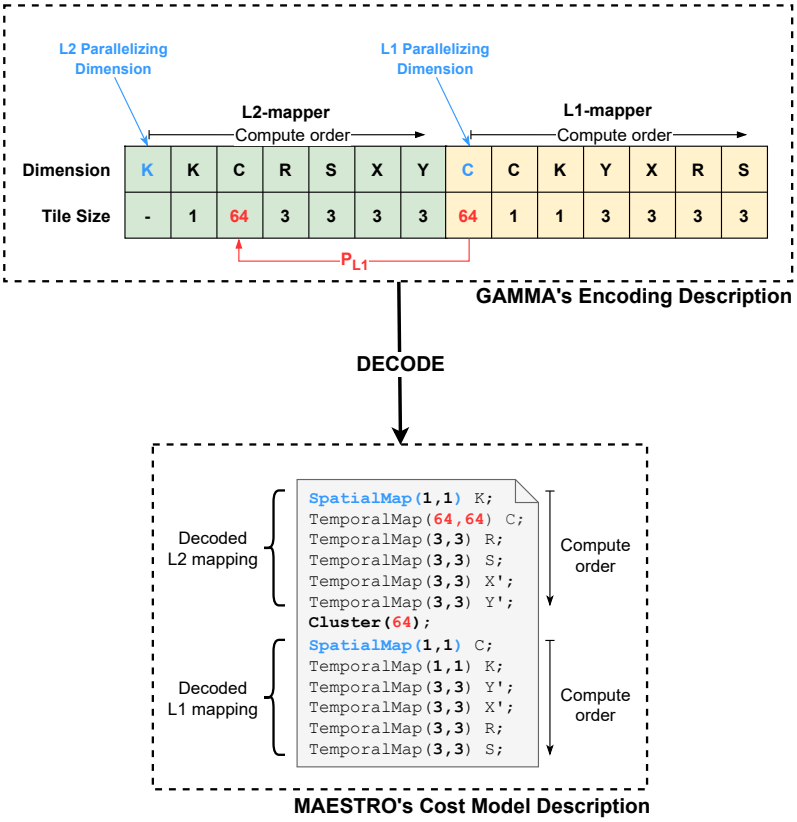

Fig. 6 GAMMA's encoding of a two-level mapper (top) and its decoded description for cost model (MAESTRO) of an NVDLA-like two-level mapper (bottom). Adopted from [29].

GAMMA supports several optimization algorithms, including a genetic algorithm with application-specific operators for mutation and crossover [29]. The fitness is defined as a reward value (e.g., latency, energy, or power) if a constraint on hardware resources is met. Otherwise (i.e., when evolved mapping requires more resources than the accelerator provides), a large penalty is assigned. GAMMA is evaluated on five CNN models with different complexity (VGG16, MobileNet-V2, ResNet-50, ResNet-18, MnasNet) and two platforms (TPU and Eyeriss) with a different number of hardware resources. Across CNN models and various hardware platforms considered, GAMMA finds solutions costing 5× to (1.2E+5)× less latency and 2× to (1.6E+4)× less energy.

In another method, AnaCoNGA, the quantization problem and hardware optimization problem are solved concurrently for a given (trained) CNN [19]. The hardware architecture search (HAS) is embedded into quantization strategy search (QSS), in a nested genetic algorithm formulation. For each potential quantization strategy proposed by QSS, the HAS loop efficiently optimizes the accelerator's parameters. In QSS, a multi-objective GA is used to tackle the multi-criteria optimization problem of maximizing accuracy and minimizing hardware-related costs. No hardware design takes place in this search. The quantization search space for a CNN has a size of Q^{2L}, where Q is the set of possible quantization levels for weights and activations, and L is the number of layers in the neural network. In the second GA (HAS), each individual's genome captures hardware parameters such as the PE size, the number of binary dot-products each PE can perform in parallel, and buffer sizes. The fitness criteria of this GA are the hardware design's execution performance (compute cycles and DRAM accesses) of a predetermined quantized CNN, as well as the number of FPGA resources (BRAMs and LUTs) it requires for its allocation. Note that an alternative approach to the nested formulation could be to combine HAS genomes with QSS genomes into one GA. However, this would result in a prohibitively complex and large search space which is difficult for GA.

In order to quickly evaluate candidate hardware designs, an analytical hardware model for the execution of CNN on a state-of-the-art accelerator (such as the Xilinx Z7020 SoC on the PYNQ-Z1 board) is created. AnaCoNGA is evaluated on ResNet-20 (using CIFAR-10 data set), ResNet56 (CIFAR-100), and ResNet18 (ImageNet). With AnaCoNGA, the accuracy of ResNet20 (on CIFAR-10) is improved by 2.88% compared to a uniform 2-bit CNN, and achieved a 35% and 37% improvement in latency and DRAM accesses, while reducing LUT and BRAM resources by 9% and 59% respectively, when compared to an edge variant of the accelerator [19].

3.1.5 Weight sharing and compression

Weight sharing enables to replace a group of similar weights by a single value. Instead of storing all the CNN weights, only a limited number of shared values and a codebook, where original weights are replaced by their corresponding indexes, are stored in the CNN memory. This approach is also known as *weight compression*. For example, if the original weights are encoded on 32 bits and there are 256 different

shared values, only the shared values and 8-bit indexes must be stored, reducing thus the memory footprint almost four times compared to the original weight set. Shared weight values are typically obtained with a clustering algorithm like K-means. The weight sharing idea can be applied at the level of the entire CNN or for each layer separately. The objective is to automatically select the optimal number of shared values per layer for the input CNN assuming that a range of possible numbers of shared values is provided. Introducing the shared values typically leads to a loss in accuracy. Hence, a suitable trade-off between the accuracy drop and compression rate is sought. Dupuis et al. [17] adopted a two-step approach to compute the shared weights. In the first step, the number of shared values is determined locally and separately for each layer by an exhaustive search The second step, based on the values obtained in the first step, tries to determine the most suitable combinations of shared weights across the entire CNN. As the number of combinations grows exponentially with the number of layers, the problem is solved by NSGA-II. The bottleneck is the accuracy evaluation because it requires evaluating CNN for each set of candidate (shared) weights. A proxy regression model was created using data obtained in the first step to accelerate the evaluation. The results carried out on recent CNN models, trained with the ImageNet dataset, show over $5\times$ memory compression at an acceptable accuracy loss without any retraining step.

4 NAS considering the target hardware

NAS methods utilizing evolutionary algorithms to deliver a CNN architecture with a minimum error on test data were introduced in previous chapters of this book. *Hardware-aware NAS* methods extend this approach by considering other objectives such as latency, energy efficiency, and memory footprint with respect to a hardware platform implementing the neural network [63, 3]. Hardware-aware NAS methods can be seen as multi-objective optimization methods. Hence, in certain steps of the NAS algorithm, all relevant objectives must be evaluated, either by direct measurement on real hardware or estimated using software models (Section 2.4). A common approach to solve the multi-objective NAS problem adopted by the NAS community is either (i) to transform it into a single-objective one (using suitable constraints, prioritization, or aggregation techniques) and solve it with a common single-objective method or (ii) to employ a truly multi-objective approach (such as NSGA-II) [15].

A common practice is to model a candidate CNN using a directed acyclic graph encoded as a variable-length string. If only some hyperparameters are optimized then the chromosome is a fixed-length list of integers. All possible strings describing valid CNNs constitute the search space. To reduce the NAS time, the so-called *supernet* is often constructed first [55]. A supernet is an over-parameterized neural network built over a certain backbone CNN model, in which many options are supported for selected hyperparameters. The supernet is trained to solve a given problem. Its training is usually very costly as the supernet is more complex than any individual CNN. However, this cost can be amortized as many suitable subnet-

works, including their weights, can be extracted from the resulting supernet for a given specification (e.g., latency, accuracy, or energy constraint on given hardware) and used without repeating the expensive training process. The search for a suitable subnetwork, which can be conducted using an EA, is less expensive because it does not involve any training. A general limitation of this approach is that the supernet restricts the search space to its subnetworks.

Two major directions can be identified in the area of NAS methods explicitly targeting hardware implementations:

- *Hardware-aware NAS*, whose goal is to find the most suitable CNN model concerning a target hardware platform and the objectives to be optimized. Note that there is no additional search space to the neural architecture search space.
- *NAS with hardware co-optimization*, whose goal is to co-optimize CNN model and hardware configuration (such as amount and type of resources, dataflow strategies, buffer sizes, and compiler options). These methods work in three search spaces (weights, neural architectures, and hardware configurations) and must innovatively orchestrate several search algorithms to produce the best trade-offs between the accuracy and various hardware-relevant metrics.

These two directions will be discussed in the rest of this section.

4.1 HW-aware evolutionary NAS

An evident approach to optimizing the CNN architecture for given hardware is employing only hardware-friendly hyperparameters and operations, i.e., suitable convolution types, arithmetic operator implementations, quantization schemes, or memory access mechanisms with respect to the optimization objectives. For example, based on benchmarking 32 different operators, Hurricane [78] uses different subsets of operator choices for three hardware platforms. This way, the search space is narrowed toward CNN architectures suitable for a given hardware platform.

4.1.1 Classification of evolutionary HW-aware NAS methods

Table 4 showcases the key properties of selected evolutionary hardware-aware NAS methods. The search is conducted either at the *macro level* (i.e., the entire CNN is encoded in the chromosome) or the *subnetwork* level (also known as a *block* or a *cell*), in which only a subnetwork is optimized by an EA. The resulting subnetwork can be used multiple times in the final CNN. These NAS methods are often called a *micro-level* NAS. The search space can also be reduced to a few hyperparameters of a given pre-designed CNN architecture (see *hyperp* in the *Search Space* column in Table 4). Column *SuperNet* informs whether a supernet is used. The design objectives are listed in the *Objectives* column; the accuracy is not mentioned as it is always involved. The *Estimation Method* column tells us if at all and how particular

hardware parameters are estimated. We observe that latency (Lat) and Energy are often estimated rather than measured. If the accuracy (Acc) is estimated, then an NN-based predictor (surrogate) is almost always utilized for this purpose [73, 39]. The *Target device* informs about the target hardware platform(s). Finally, column *Data Set* lists the problems/data set(s) used for evaluation. It has to be noticed that, in addition to image classification, some other tasks are approached.

Some multi-objective NAS methods (e.g., [10, 18, 39, 38]) only optimize the number of FLOPs of weights, which are not highly correlated with the real hardware parameters such as latency and energy. We included these methods in Table 4 to cover the whole scope of methods in this area. The following paragraphs briefly present some recent hardware-aware NAS methods utilizing evolutionary algorithms.

4.1.2 Selected evolutionary NAS methods

The first evolutionary NAS methods such as [59] did not consider any hardware parameters during the evolution. Later, the NAS has become a truly multi-objective method.

The Lamarckian Evolutionary algorithm for Multi-Objective Neural Architecture DEsign (LEMONADE) [18] is a multi-objective NAS. It first selects a subset of architectures, assigning a higher probability to architectures that would fill gaps on the Pareto front for the objectives that can easily be evaluated (e.g., the number of parameters); then, it trains and evaluates only this subset to save computational resources during the architecture search. It proposes a Lamarckian inheritance mechanism that generates child networks that are warm-started with the predictive performance of their trained parents. This is accomplished by using (approximate) network morphism operators for generating children. Within 5 days on 16 GPUs, LEMONADE discovers architectures that are competitive in terms of predictive performance and resource consumption with hand-designed networks, such as MobileNetV2.

Schorn et al. [62] also employ a set of objective functions for the prediction of energy consumption, latency, and required bandwidth of DNNs on hardware, solely based on the topology of neural architecture to avoid the need for expensive simulations or training of candidate CNNs. Furthermore, they also consider error resilience as one of the objectives. Error resilience is seen as the robustness of the neural network classifier against perturbations in its neuron activation values. Such perturbations can be the result of random hardware faults, such as radiation-induced bit-flips. Hence, random bit-flip error simulations are used to evaluate the actual resilience of the obtained set of neural networks. Evolved CNNs achieve about a $6\times$ to $7\times$ lower data corruption rate at 0.5% bit error rate in the feature maps of the network in comparison with MobileNetV2.

GoldenNAS [41] introduces a novel dynamic channel scaling scheme to enable the channel-level search, a progressive space shrinking method to progressively shrink the search space toward target hardware, and an adaptive batch normaliza-

tion technique to enable the depthwise adaptiveness of CNNs under dynamic environments. GoldenNAS adopts the weight-sharing technique based on the supernet paradigm, where the supernet is derived from ShuffleNetV2. Multiobjective EA samples the supernet to obtain CNNs showing suitable trade-offs between accuracy and latency for various hardware platforms – GPU (Nvidia Quadro GV100), CPU (Intel Xeon Gold 6136), and edge device (Nvidia Jetson Xavier). Latency is modeled analytically. Note that supernet training for 100 epochs takes about 70 GPU hours, and each stage of the progressive space shrinking takes about 22 GPU hours. The evolutionary search process requires about 6 GPU hours to finish.

Lu et al. [37] utilizes latency monotonicity (i.e., the observation that the architecture latency rankings on different devices are often correlated on other devices) to reuse models from one proxy device on several target devices (several mobile and non-mobile devices tested in this paper). It avoids building a latency predictor for each target device. Hence, only one latency predictor based on an MLP with four layers is used. The search space is built up on MobileNet-V2 with multiplier 1.3, with the channel number in each block fixed. The search space consists of the depth of each stage, the kernel size of convolutional layers, and the expansion ratio of each block. The depth can be chosen from $\{2,3,4\}$, kernel size can be $\{3,5,7\}$, and candidate expansion ratios are $\{3,4,6\}$. There are five stages whose configurations can be searched. The one-shot NAS utilizes Once-For-All network [7] as a supernet. EA is searching for optimal architectures using one proxy device with 1000 individuals in the population and 50 generations for each latency constraint. The evolutionary search takes less than 30 seconds for each run.

APQ [73] also exploits the supernet. It utilizes a joint model architecture-pruning-quantization search. A mixed quantization is applied after extracting pruned sub-networks from the supernet. An energy/latency look-up table is used to provide the hardware feedback during the search.

4.2 NAS with hardware co-design

As emphasized by Lin et al. [35], NAS with hardware co-design opens a new search space – hardware configurations – to deeply co-optimize the CNN architecture and its hardware implementation with the aim of delivering the most suitable trade-offs between the accuracy and hardware parameters. In addition to the CNN architecture and weights, the hardware configuration is optimized, which can involve optimizing the bit widths, quantization levels, PE array size, buffer size, MAC circuit configuration (e.g. by utilizing approximate multipliers), data flow organization, tiling strategy, loop order, memory subsystem parameters, preferences for the high-level synthesis tools, etc.

Table 4 Hardware-aware evolutionary NAS methods. Titles of some data sets are abbreviated, e.g., C-10 for Cifar-10, C-100 for Cifar-100, ImgNet for ImageNet; the + symbol denotes that some additional data sets were omitted because of space limitations.

Method	Ref.	Year	Search Space	Super Net	Objectives	Estimation Method	Target device	Data Set
Large-Scale	[59]	2017	macro		None	None	GPU	C-10, C-100
JASQNet	[10]	2018	cell		params	None	GPU	ImgNet, C-10
ECAD	[13]	2019	hyperp		Lat, Energy	simulator	FPGA	MNIST
ChamNet	[14]	2019	hyperp		Lat, Energy	Acc, Energy: GP predictor	GPU, DSP, Mobile	ImgNet
LEMONADE	[18]	2019	macro, cell		Params	None	GPU	ImgNet, C-10
NSGANetV1	[38]	2019	block		FLOPS	None	GPU	C-10, C-100
APQ	[73]	2020	block	Y	Lat, Energy	Acc: NN; Lat: LUT	ASIC	ImgNet
DeepMaker	[36]	2020	hyperp		Size	None	CPU, GPU, FPGA	MNIST, C-10, C-100
HNAS	[76]	2020	macro		Lat	Lat: LUT	GPU, Mobile	ImgNet
Hurricane	[78]	2020	macro	Y	Lat	Lat: Bayes. Regression	DSP, CPU, ASIC	ImgNet
MCUNet	[33]	2020	macro	Y	Lat, Mem, Flash	None	MCU	ImgNet, WWV, KWS
NasCaps	[43]	2020	hyperp		Lat, Mem, Energy	Lat: cycles; Energy: model	ASIC	C-10, MNIST, FMNIST+
NSGANetV2	[39]	2020	block	Y	Lat, MAC, Params	Acc: ML-surrogate	GPU	ImgNet, C-10, C-100+
OFA	[7]	2020	block	Y	Lat	Acc, Lat: NN	GPU, FPGA, Mobile	ImgNet
PONAS	[24]	2020	macro	Y	Params, FLOPS	LUT	GPU	ImgNet
Schorn et al.	[62]	2020	hyperp		Lat, Energy, FT	formula	GPU	C-10, GTSRB
SPOS	[22]	2020	blocks	Y	Lat, FLOPS	None	GPU	ImgNet
μNAS	[32]	2021	macro		Lat, Mem, MAC	Lat: MAC	MCU	C-10, MNIST, Chars74K+
HSCoNAS	[40]	2021	block	Y	Lat	Lat: formula	GPU, CPU	ImgNet
Prabakaran et al.	[57]	2021	macro		Mem	None	GPU, CPU	anomaly in ECG signals
Wang et al.	[74]	2021	block		FLOPS	None	GPU	IDS2012, ISCX VPN
NAS4RRAM	[77]	2021	cells		Energy	simulator	RRAM chip	C-10, C-100
NEMOKD	[67]	2021	hyperp		Lat	None	Movidius	C-10, C-100
GoldenNAS	[41]	2022	layer	Y	Lat	Analytical	Edge GPU, GPU, CPU	ImgNet
Lu et al.	[37]	2022	macro	Y	Lat	MLP	mobile, GPU, CPU	ImgNet

4.2.1 A single search algorithm

A straightforward approach is to add the hardware parameters to the chromosome which describes the CNN architecture, and extend thus the search space of the original NAS algorithm.

For example, Pinos et al. [56] evolved CNN architecture together with the selection of suitable approximate multipliers for particular CNN layers to reduce power consumption. The method, EvoApproxNAS, is based on CGP in which each node represents a network layer and a layer can use one of 35 multipliers. Fig. 7 shows resulting Pareto fronts from four independent experiments in which EvoApproxNAS utilized four different sets of multipliers in convolutional layers: 8×8-bit accurate (blue), $8\times N$-bit accurate (orange), 8×8-bit approximate (red), and $8\times N$-bit approximate (green). By combining all these Pareto fronts, one can observe those approximate multipliers allow EvoApproxNAS to reach the best trade-offs between accuracy and energy of multiplication for almost all investigated regions of parameters.

4.2.2 Two Search Algorithms

The approach presented in the previous section leads to a time-consuming search process due to the prohibitively huge joint space composed of the coupled yet different CNN architecture and hardware configuration spaces with extremely sparse optima.

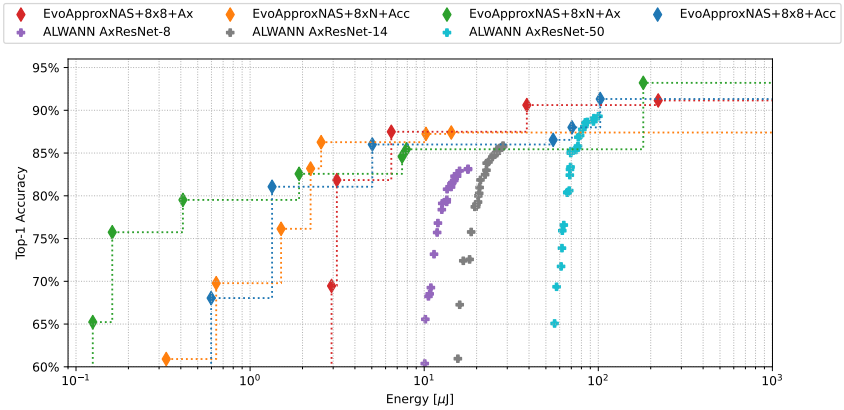

Fig. 7 The energy-accuracy trade-offs obtained by EvoApproxNAS on CIFAR-10 for four sets of multipliers that can be used by convolutional layers. Results are compared with various ResNet networks optimized with the ALWANN method [49].

To reduce the search cost, the problem is often decoupled. Two search algorithms are now employed. Algorithm A_1 is a common NAS and works in the space of CNN architectures. Algorithm A_2 then performs the search in the space of hardware configurations. It can again be based on an EA; however, other search techniques have been utilized in the literature [63]. The search algorithms can interact in different ways, for example:

1. A_1 samples a CNN model α. No training of α is performed.
2. A_2 is executed to find the most suitable hardware configuration c_{hw} (satisfying all hardware constraints imposed by the specification) for α.
3. If no suitable hardware configuration is obtained, α is discarded, and step (1) is taken again.
4. If c_{hw} satisfies all constraints, then α is trained and then tested on the test data to get its accuracy $Acc(\alpha)$.
5. Steps (1) to (4) are repeated until a suitable solution $(\alpha, c_{hw}, Acc(\alpha))$ is not reached.

This approach is especially useful if finding a suitable hardware configuration for α takes significantly less time than the training of α. On the other hand, if a super net is employed, it is not necessary to train candidate CNN architectures (subnets), and a search in the hardware configuration space can be conducted for architectures showing acceptable accuracy. From the NAS-hardware codesign methods, surveyed by Sekanina [63], Table 5 lists those utilizing an evolutionary approach for A_1 (the *NAS Method* column) or A_2 (the *HW opt. method*). Selected hardware parameters that are optimized are listed in the *Design parameters* column. The remaining columns have the same meaning as in Table 4. Note that the use of EAs is relatively unexplored in this new area as documented by only three items in Table 5.

QNAS [34] focuses on optimizing the parameters of a mixed-precision systolic-array-like architecture (the array size, buffer input/weight/output size) while searching the quantized neural architecture. It includes an EA-based hardware architecture search and a one-shot supernet-based quantized neural architecture search. First, a suite of neural architectures is sampled as a benchmark to find the hardware architecture that achieves the best performance on the benchmark. The hardware architecture is fixed, and the quantized neural architecture search (QNAS) is then performed to determine the neural architecture and quantization policy. The quantized neural network is composed of multiple ResNet blocks.

NAAS [35] holistically searches the neural network architecture and accelerator architecture, and unlike other methods (e.g., [26]), compiler mapping. The accelerator search space is defined by the number of processing elements, local memory size, global buffer size, memory bandwidth, and connectivity parameters. NAAS employs EA to optimize these parameters as well as the compiler mapping (the execution order and the tiling size). It introduces a special encoding, called importance-based encoding, for the accelerator space and the compiler mapping strategies to avoid enumerating all possible situations and representing them by indexes. First, NAAS generates a pool of accelerator candidates. For each accelerator candidate, a network architecture is sampled from a pre-trained network [7] that satisfies the pre-defined accuracy requirement. Since each subnet is well-trained, the accuracy evaluation is fast. Finally, the compiler mapping strategy is sought for the network candidate on the corresponding accelerator candidate.

Table 5 Evolutionary NAS methods with hardware co-design

Method	Ref.	NAS Method	Objective	HW opt. method	Design parameters	Target device	Data set
QNAS	[34]	EA, supernet	EDP	EA	#PE, mem. params	ASIC	ImgNet, C-10
NAAS	[35]	gradient, supernet	EDP	EA	#PE, mem. params., compiler mapping	TPU, ASIC	ImgNet, C-10
Pinos et al.	[56]	EA	Energy	in NAS	approximate multiplier type	ASIC	C-10, SVHN

Fig. 8 shows the impact of various approaches in optimizing the accuracy and Energy-Delay-Product (EDP) of ImageNet classifiers based on ResNet-50 and implemented on an Eyeriss-like chip. The original implementation (black point) of ResNet-50 (no NAS employed) is improved by a hardware search algorithm from QNAS [34] (green point). Additional improvement is provided by NAAS performing the hardware and compiler mapping co-search (orange point). The best trade-offs are reported for NAAS utilizing the hardware, compiler mapping, and CNN architecture co-search (blue points). These results (adopted from [35]) demonstrate that exploiting more design spaces can lead to better CNN implementations.

5 Conclusions and open challenges

We surveyed evolutionary approaches developed to optimize hardware implementations of CNNs and the NAS methods utilizing EAs. The optimization of vari-

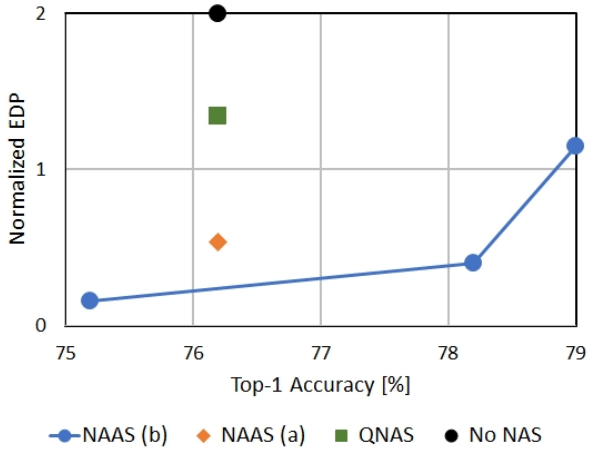

Fig. 8 Normalized EDP and top-1 accuracy (on ImageNet) obtained by NAS methods for CNNs running on an ASIC [35]: NAAS co-optimizing HW, compiler mapping and NN architecture (blue); NAAS co-optimizing HW and compiler mapping (orange); hardware search from QNAS (green); No NAS conducted (black). Adopted from [35].

ous components and implementation principles of hardware accelerators with EAs seems to be a useful strategy because the relationships among all internal variables in complex systems such as hardware accelerators are highly nonlinear and corresponding search spaces are hard to explore.

Introducing the hardware search space in NAS algorithms has led to more efficient implementations of CNNs on particular hardware platforms. However, several search algorithms working in the space of weights, neural architectures, and hardware configurations have to be coordinated, making the entire method complicated. Successful adoption of EAs in these applications requires utilizing not only modern multi-objective evolutionary design and optimization methods but also state-of-the-art (surrogate) modeling and simulation techniques to get reliable information about the underlying hardware quickly.

In the following sub-sections, we outline the challenges that are critical for the successful development in this area.

5.1 Benchmarking and reproducibility

As hardware-aware NAS methods are multi-objective, their fair assessment consists of evaluating multiple parameters of resulting implementations of CNNs and the design cost (time). It thus leads to an expensive construction and comparison of Pareto fronts in multidimensional spaces, which is often hard to perform because of incomplete information about some NAS methods. To support a fair benchmarking

methodology and accelerate the development of new NAS methods, the open-source data sets containing many pre-trained and evaluated CNNs from well-defined search spaces were introduced in the literature, e.g., NAS-Bench-201 [16]. In addition to the accuracy for each design point in the search space, hardware parameters (such as latency and power) are also precomputed for some hardware accelerators [31]. Hence, new NAS algorithms can quickly be developed and evaluated for predefined search spaces. We see a lot of space for further opening the whole field to a broader community of researchers and practitioners by sharing NAS implementation source codes, data generated by NAS methods, data measured on real accelerators, and data obtained from simulations of various configurations of hardware accelerators. This effort should also improve the reproducibility of results in this area.

5.2 Security and Reliability

In addition to optimizing the quality of service, performance, and power consumption, other objectives must be considered when hardware-accelerated DNNs are deployed in real-world systems. DNN systems are highly vulnerable to security and reliability threats at both the cloud and the edge. Security attacks include inserting random or crafted noise into the data, inserting malicious components into the system hardware, polluting inputs with imperceptible noise during inference, and monitoring system-side channels to deduce the underlying model [64]. Reliability issues include process variation during hardware fabrication, memory errors, and specific environmental conditions around the system that compromise reliability during training and inference. Shafique et al. [64] surveyed the threats and their respective countermeasures. One example of reliability-aware EA-based NAS – paper [62] – was discussed in Section 4. We expect a lot of research that could potentially utilize evolutionary algorithms in the areas of security and reliability of CNN accelerators.

5.3 Unconventional Hardware Platforms

Emerging technologies such as memristive crossbars or in-memory computing are investigated for CNN accelerators to reduce power consumption and other critical parameters [66, 2]. A very specialized simulator is usually developed to analyze the properties of these unconventional circuits and systems. The simulator can be connected with a NAS algorithm to find best-performing CNN-accelerator pairs. For example, PABO [54] uses NAS connected with a memristive crossbar-based CNN accelerator, where the CNN is mapped across the on-chip crossbar storage spatially. NAS4RRAM is a NAS method for optimizing CNNs and Resistive Random Access Memory (RRAM)-based accelerators [77]. NACIM [27] jointly explores device, circuit, and architecture design space and also takes device variation into account to

find the most robust neural architectures, coupled with the most efficient hardware design for an in-memory computing ASIC. In the future, more exotic hardware platforms for CNNs could be introduced (e.g., similar to the nanoparticle networks configured using evolutionary algorithms for solving simple problems [5]) to provide richer and deeper interaction of machine learning and configurable physical *materio*.

5.4 Design Cost

The evolutionary NAS method is a computationally expensive approach requiring many core hours producing considerable CO_2 emissions. We expect many new approaches to reduce the computation cost in all directions, including efficient search algorithms, network training algorithms, hardware simulation, and benchmarking strategies.

Acknowledgements This work was supported by the Czech science foundation project *Automated design of hardware accelerators for resource-aware machine learning* under number 21-13001S.

References

[1] Barone S, Traiola M, Barbareschi M, Bosio A (2021) Multi-objective application-driven approximate design method. IEEE Access 9:86,975–86,993
[2] Bavikadi S, Dhavlle A, Ganguly A, Haridass A, Hendy H, Merkel C, Reddi VJ, Sutradhar PR, Joseph A, Pudukotai Dinakarrao SM (2022) A survey on machine learning accelerators and evolutionary hardware platforms. IEEE Design & Test 39(3):91–116
[3] Benmeziane H, El Maghraoui K, Ouarnoughi H, Niar S, Wistuba M, Wang N (2021) Hardware-aware neural architecture search: Survey and taxonomy. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, International Joint Conferences on Artificial Intelligence Organization, pp 4322–4329, survey Track
[4] Bingham G, Macke W, Miikkulainen R (2020) Evolutionary optimization of deep learning activation functions. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, ACM, GECCO '20, p 289–296
[5] Bose SK, Lawrence CP, Liu Z, Makarenko KS, van Damme RMJ, Broersma HJ, van der Wiel WG (2015) Evolution of a designless nanoparticle network into reconfigurable boolean logic. Nature Nanotechnology 10:1048 – 1052
[6] Cai H, Zhu L, Han S (2019) Proxylessnas: Direct neural architecture search on target task and hardware. In: 7th International Conference on Learning Representations, ICLR, OpenReview.net

[7] Cai H, Gan C, Wang T, Zhang Z, Han S (2020) Once-for-all: Train one network and specialize it for efficient deployment. In: 8th International Conference on Learning Representations, ICLR, OpenReview.net
[8] Capra M, Bussolino B, Marchisio A, Shafique M, Masera G, Martina M (2020) An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. Future Internet 12(7):113
[9] Ceska M, Matyas J, Mrazek V, Sekanina L, Vasicesk Z, Vojnar T (2022) Sagtree: Towards efficient mutation in evolutionary circuit approximation. Swarm Evol Comput 69:100,986
[10] Chen Y, Meng G, Zhang Q, Zhang X, Song L, Xiang S, Pan C (2018) Joint neural architecture search and quantization, URL https://arxiv.org/abs/1811.09426, 1811.09426
[11] Chen YH, Krishna T, Emer JS, Sze V (2017) Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits 52(1):127–138, DOI 10.1109/JSSC.2016.2616357
[12] Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E (2014) cuDNN: Efficient primitives for deep learning, URL http://arxiv.org/abs/1410.0759
[13] Colangelo P, Segal O, Speicher A, Margala M (2019) Artificial neural network and accelerator co-design using evolutionary algorithms. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–8, DOI 10.1109/HPEC.2019.8916533
[14] Dai X, Zhang P, Wu B, Yin H, Sun F, Wang Y, Dukhan M, Hu Y, Wu Y, Jia Y, Vajda P, Uyttendaele M, Jha NK (2019) ChamNet: Towards efficient network design through platform-aware model adaptation. In: 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp 11,390–11,399, DOI 10.1109/CVPR.2019.01166
[15] Deb K (2009) Multi-Objective Optimization Using Evolutionary Algorithms, Wiley
[16] Dong X, Yang Y (2020) NAS-Bench-201: Extending the scope of reproducible neural architecture search. In: International Conference on Learning Representations
[17] Dupuis E, Novo D, O'Connor I, Bosio A (2022) A heuristic exploration of retraining-free weight-sharing for CNN compression. In: 27th Asia and South Pacific Design Automation Conference, ASP-DAC, IEEE, pp 134–139
[18] Elsken T, Metzen JH, Hutter F (2019) Efficient multi-objective neural architecture search via Lamarckian evolution. In: 7th Int. Conference on Learning Representations, ICLR 2019, OpenReview.net
[19] Fasfous N, Vemparala MR, Frickenstein A, Valpreda E, Salihu D, Höfer J, Singh A, Nagaraja NS, Voegel HJ, Vu Doan NA, Martina M, Becker J, Stechele W (2022) AnaCoNGA: Analytical HW-CNN co-design using nested genetic algorithms. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp 238–243
[20] Garofalo A, Rusci M, Conti F, Rossi D, Benini L (2019) PULP-NN: A computing library for quantized neural network inference at the edge on RISC-V

based parallel ultra low power clusters. In: 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), pp 33–36

[21] Garofalo A, Tagliavini G, Conti F, Rossi D, Benini L (2020) XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions. In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp 186–191, DOI 10.23919/DATE48585.2020.9116529

[22] Guo Z, Zhang X, Mu H, Heng W, Liu Z, Wei Y, Sun J (2019) Single path one-shot neural architecture search with uniform sampling. CoRR abs/1904.00420, URL http://arxiv.org/abs/1904.00420

[23] He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition, URL http://arxiv.org/abs/1512.03385, 1512.03385

[24] Huang S, Chu W (2020) PONAS: progressive one-shot neural architecture search for very efficient deployment. CoRR abs/2003.05112, URL https://arxiv.org/abs/2003.05112, 2003.05112

[25] Intel (2021) Intel-optimized math library for numerical computing. URL https://software.intel.com/en-us/mkl

[26] Jiang W, Yang L, Sha EHM, Zhuge Q, Gu S, Dasgupta S, Shi Y, Hu J (2020) Hardware/software co-exploration of neural architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39(12):4805–4815, DOI 10.1109/TCAD.2020.2986127

[27] Jiang W, Lou Q, Yan Z, Yang L, Hu J, Hu XS, Shi Y (2021) Device-circuit-architecture co-exploration for computing-in-memory neural accelerators. IEEE Transactions on Computers 70(4):595–605, DOI 10.1109/TC.2020.2991575

[28] Jouppi NP, Young C, Patil N, Patterson D (2018) A domain-specific architecture for deep neural networks. Commun ACM 61(9):50–59

[29] Kao SC, Krishna T (2020) Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In: Proceedings of the 39th International Conference on Computer-Aided Design, ACM, ICCAD '20

[30] Lapid R, Sipper M (2022) Evolution of activation functions for deep learning-based image classification. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, ACM, GECCO '22, p 2113–2121

[31] Li C, Yu Z, Fu Y, Zhang Y, Zhao Y, You H, Yu Q, Wang Y, Hao C, Lin Y (2021) HW-NAS-Bench: Hardware-aware neural architecture search benchmark. In: 9th International Conference on Learning Representations, ICLR 2021, OpenReview.net

[32] Liberis E, Dudziak u, Lane ND (2021) μNAS: Constrained neural architecture search for microcontrollers. ACM, EuroMLSys '21, p 70–79

[33] Lin J, Chen WM, Lin Y, Cohn J, Gan C, Han S (2020) MCUNet: Tiny deep learning on iot devices. In: 34th Conference on Neural Information Processing Systems (NeurIPS 2020), pp 1–12

[34] Lin Y, Hafdi D, Wang K, Liu Z, Han S (2019) Neural-hardware architecture search. In: 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)

[35] Lin Y, Yang M, Han S (2021) NAAS: Neural accelerator architecture search. In: 2021 58th ACM/ESDA/IEEE Design Automation Conference (DAC)

[36] Loni M, Sinaei S, Zoljodi A, Daneshtalab M, Sjödin M (2020) DeepMaker: A multi-objective optimization framework for deep neural networks in embedded systems. Microprocessors and Microsystems 73:102,989, DOI https://doi.org/10.1016/j.micpro.2020.102989

[37] Lu B, Yang J, Jiang W, Shi Y, Ren S (2021) One proxy device is enough for hardware-aware neural architecture search. Proc ACM Meas Anal Comput Syst 5(3)

[38] Lu Z, Whalen I, Boddeti V, Dhebar Y, Deb K, Goodman E, Banzhaf W (2019) NSGA-Net: Neural architecture search using multi-objective genetic algorithm. In: Proc. of the Genetic and Evolutionary Computation Conference, ACM, GECCO '19, p 419–427

[39] Lu Z, Deb K, Goodman E, Banzhaf W, Boddeti VN (2020) NSGANetV2: Evolutionary multi-objective surrogate-assisted neural architecture search. In: Computer Vision – ECCV 2020, Springer, Cham, pp 35–51

[40] Luo X, Liu D, Huai S, Liu W (2021) HSCoNAS: Hardware-software co-design of efficient DNNs via neural architecture search. In: DATE 2021, 2103.08325

[41] Luo X, Liu D, Huai S, Kong H, Chen H, Liu W (2022) Designing efficient DNNs via hardware-aware neural architecture search and beyond. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41(6):1799–1812

[42] MAESTRO (2021) An open-source infrastructure for modeling dataflows within deep learning accelerators. URL http://maestro.ece.gatech.edu/

[43] Marchisio A, Massa A, Mrazek V, Bussolino B, Martina M, Shafique M (2020) NASCaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In: 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp 1–9

[44] Mazumder AN, Meng J, Rashid HA, Kallakuri U, Zhang X, Seo JS, Mohsenin T (2021) A survey on the optimization of neural network accelerators for micro-ai on-device inference. IEEE Journal on Emerging and Selected Topics in Circuits and Systems 11(4):532–547, DOI 10.1109/JETCAS.2021.3129415

[45] Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4):1–33, DOI 10.1145/2893356

[46] Mittal S (2020) A survey of FPGA-based accelerators for convolutional neural networks. Neural Computing and Applications 32(32):1109–1139

[47] Mittal S, Rajput P, Subramoney S (2022) A survey of deep learning on cpus: Opportunities and co-optimizations. IEEE Transactions on Neural Networks and Learning Systems 33(10):5095–5115

[48] Mrazek V, Hrbacek R, et al (2017) Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Proc. of DATE'17, pp 258–261

[49] Mrazek V, Vasicek Z, Sekanina L, Hanif AM, Shafique M (2019) ALWANN: Automatic layer-wise approximation of deep neural network accelerators

without retraining. In: Proc. of the IEEE/ACM International Conference on Computer-Aided Design, IEEE, pp 1–8

[50] Mrazek V, Sekanina L, Vasicek Z (2020) Libraries of approximate circuits: Automated design and application in CNN accelerators. IEEE Journal on Emerging and Selected Topics in Circuits and Systems 10(4):406–418, DOI 10.1109/JETCAS.2020.3032495

[51] Murshed MGS, Murphy C, Hou D, Khan N, Ananthanarayanan G, Hussain F (2021) Machine learning at the network edge: A survey. ACM Comput Surv 54(8), DOI 10.1145/3469029, URL https://doi.org/10.1145/3469029

[52] Nader A, Azar D (2021) Evolution of activation functions: An empirical investigation. ACM Trans Evol Learn Optim 1(2)

[53] Parashar A, Raina P, Shao YS, Chen YH, Ying VA, Mukkara A, Venkatesan R, Khailany B, Keckler SW, Emer J (2019) Timeloop: A systematic approach to dnn accelerator evaluation. In: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 304–315

[54] Parsa M, Ankit A, Ziabari A, Roy K (2019) PABO: Pseudo agent-based multi-objective bayesian hyperparameter optimization for efficient neural accelerator design. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp 1–8, DOI 10.1109/ICCAD45719.2019.8942046

[55] Pham H, Guan MY, Zoph B, Le QV, Dean J (2018) Efficient neural architecture search via parameter sharing. In: Proc. of the 35th International Conference on Machine Learning, ICML 2018, PMLR, vol 80, pp 4092–4101

[56] Pinos M, Mrazek V, Sekanina L (2022) Evolutionary approximation and neural architecture search. Genetic Programming and Evolvable Machines 23(3):351–374

[57] Prabakaran BS, Akhtar A, Rehman S, Hasan O, Shafique M (2021) BioNetExplorer: Architecture-space exploration of bio-signal processing deep neural networks for wearables. IEEE Internet of Things Journal pp 1–10, DOI 10.1109/JIOT.2021.3065815

[58] Prashanth HC, Madhav R (2022) Evolutionary standard cell synthesis of unconventional designs. In: Proceedings of the Great Lakes Symposium on VLSI 2022, ACM, GLSVLSI '22, p 189–192

[59] Real E, Moore S, Selle A, Saxena S, Suematsu YL, Tan J, Le QV, Kurakin A (2017) Large-scale evolution of image classifiers. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70, JMLR.org, ICML'17, p 2902–2911

[60] Sarwar SS, Venkataramani S, Ankit A, Raghunathan A, Roy K (2018) Energy-efficient neural computing with approximate multipliers. J Emerg Technol Comput Syst 14(2):16:1–16:23

[61] Sateesan A, Sinha S, G SK, Vinod AP (2021) A survey of algorithmic and hardware optimization techniques for vision convolutional neural networks on FPGAs. Neural Process Lett 53(3):2331–2377

[62] Schorn C, Elsken T, Vogel S, Runge A, Guntoro A, Ascheid G (2020) Automated design of error-resilient and hardware-efficient deep neural networks. Neural Comput Appl 32(24):18,327–18,345

[63] Sekanina L (2021) Neural architecture search and hardware accelerator co-search: A survey. IEEE Access 9:151,337–151,362, DOI 10.1109/ACCESS.2021.3126685

[64] Shafique M, Naseer M, Theocharides T, Kyrkou C, Mutlu O, Orosa L, Choi J (2020) Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead. IEEE Design & Test 37(2):30–57, DOI 10.1109/MDAT.2020.2971217

[65] Sipper M (2021) Neural networks with à la carte selection of activation functions. SN Comput Sci 2(6):470, DOI 10.1007/s42979-021-00885-1, URL https://doi.org/10.1007/s42979-021-00885-1

[66] Staudigl F, Merchant F, Leupers R (2022) A survey of neuromorphic computing-in-memory: Architectures, simulators, and security. IEEE Design & Test 39(2):90–99

[67] Stewart R, Nowlan A, Bacchus P, Ducasse Q, Komendantskaya E (2021) Optimising hardware accelerated neural networks with quantisation and a knowledge distillation evolutionary algorithm. Electronics 10(4)

[68] Sze V, Chen Y, Yang T, Emer JS (2017) Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE 105(12):2295–2329

[69] Sze V, Chen Y, Yang T, Emer JS (2020) Efficient Processing of Deep Neural Networks. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers

[70] Vasicek Z, Sekanina L (2015) Evolutionary approach to approximate digital circuits design. IEEE Transactions on Evolutionary Computation 19(3):432–444

[71] Velasco-Montero D, Fernandez-Berni J, Carmona-Galan R, Rodriguez-Vazquez A (2020) Previous: A methodology for prediction of visual inference performance on IoT devices. IEEE Internet of Things Journal 7(10):9227–9240

[72] Venkataramani S, et al (2020) Efficient AI system design with cross-layer approximate computing. Proceedings of the IEEE 108(12):2232–2250, DOI 10.1109/JPROC.2020.3029453

[73] Wang T, Wang K, Cai H, Lin J, Liu Z, Wang H, Lin Y, Han S (2020) APQ: Joint search for network architecture, pruning and quantization policy. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp 2075–2084, DOI 10.1109/CVPR42600.2020.00215

[74] Wang X, Wang X, Jin L, Lv R, Dai B, He M, Lv T (2021) Evolutionary algorithm-based and network architecture search-enabled multiobjective traffic classification. IEEE Access 9:52,310–52,325, DOI 10.1109/ACCESS.2021.3068267

[75] Wu YN, Emer JS, Sze V (2019) Accelergy: An architecture-level energy estimation methodology for accelerator designs. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp 1–8

[76] Xia X, Ding W (2020) HNAS: Hierarchical neural architecture search on mobile devices, URL https://arxiv.org/abs/2005.07564, 2005.07564

[77] Yuan Z, Liu J, Li X, Yan L, Chen H, Wu B, Yang Y, Sun G (2021) NAS4RRAM: Neural network architecture search for inference on rram-based accelerators. Science China Information Sciences 64:160,407

[78] Zhang LL, Yang Y, Jiang Y, Zhu W, Liu Y (2020) Fast hardware-aware neural architecture search. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp 2959–2967, DOI 10.1109/CVPRW50498.2020.00354

[79] Zhang X, Wang J, Zhu C, Lin Y, Xiong J, Hwu Wm, Chen D (2018) DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp 1–8

[80] Zhou Y, Dong X, Akin B, Tan M, Peng D, Meng T, Yazdanbakhsh A, Huang D, Narayanaswami R, Laudon J (2021) Rethinking co-design of neural architectures and hardware accelerators, URL https://arxiv.org/abs/2102.08619, 2102.08619