Approximate Computing Architectures

Muhammad Abdullah Hanif, Vojtech Mrazek,
and Muhammad Shafique

Contents

Abstract

Approximate computing is an emerging computing paradigm for improving the efficiency of error-tolerant applications. It allows designers to trade a negligible amount of accuracy for significant efficiency gains. This chapter provides an

M. A. Hanif (✉) · M. Shafique
Division of Engineering, New York University, Abu Dhabi, UAE
e-mail: mh6117@nyu.edu; muhammad.hanif@tuwien.ac.at; muhammad.shafique@nyu.edu

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

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-15-6401-7_27-1

overview of approximate computing and how it can be exploited to offer improved efficiency while satisfying the user-defined accuracy/quality constraints. First, an overview of techniques for approximating arithmetic hardware modules is presented. Then, methodologies for efficient design space exploration of approximate modules and for building approximate accelerators are covered. Apart from hardware-level approximations, the chapter also discusses different software-level approximations and how they can be integrated with hardware approximations in a cross-layer design flow for building efficient systems.

Keywords

Approximate · Computing · Architecture · Deep neural network · DNN · Machine learning · Design method · Accelerator · Error-tolerant design · Image processing · Classification · Approximate circuits · Cross-layer

Approximate Computing

Emerging applications in the fields of cyber-physical systems (CPS) and Internet of things (IoT) have brought various challenges for the system design community. The major challenges include increases in the computational and memory requirements of applications, the growth in the number of computing devices, and the emergence of use cases with stringent energy/power constraints. All these challenges push for designing highly energy-efficient systems. Conventional techniques such as power gating and dynamic voltage and frequency scaling (DVFS) help in improving the energy efficiency of systems. However, they are insufficient to meet the growing demands of modern computing systems.

Studies by renowned research groups in the domain of energy-efficient computing systems have shown that a number of modern applications fall into the category of recognition, mining, and synthesis (RMS) applications that are (to some extent) resilient to errors (Nair 2014; Mishra et al. 2014; Esmaeilzadeh et al. 2012; Chippa et al. 2013). This error resilience is usually associated with one or more of the following factors:

1. Noise in real-world data
2. Inherent error masking characteristics of applications
3. Perceptual limitations of the users, e.g., a slight variation in the quality of an image/video (or audio) is usually unnoticeable by humans due to their psycho-visual/psychoacoustic limits
4. Absence of a unique nontrivial solution, e.g., web searches that result in slightly different but relevant links are usually equally acceptable

These factors can be exploited to relax the accuracy bounds of applications in order to achieve significant efficiency gains at the cost of minor accuracy/output quality loss.

Approximate computing (Xu et al. 2016; Shafique et al. 2016) is a computing paradigm that offers the opportunity to trade accuracy for improving the performance/efficiency of a system. This is mainly possible due to the extended design space that enables the designer to select designs having better efficiency compared to conventional designs while still meeting the user-defined accuracy constraints. Typical applications of approximate computing are in the areas of audio-visual data processing and machine learning, as slight variations in the output of these applications can be tolerated due to the intrinsic characteristics of the applications or perceptual limitations of the users.

Approximations can be applied at different layers of the HW/SW computing stack. At the software level, techniques like loop perforation (Sidiroglou-Douskos et al. 2011) and code simplification (Mohapatra et al. 2011) are commonly used, while, at hardware level, techniques like circuit approximation through voltage/frequency scaling (Chang et al. 2011) and functional approximations (Gupta et al. 2011) are widely used. Voltage/frequency scaling techniques can induce timing errors in the system, as in such cases the circuit operates at a lower voltage than the nominal value (Srinivasan et al. 2016). In functional approximation, the original circuit is replaced with a less complex substitute that exhibits almost the same functionality but improves nonfunctional circuit parameters such as power/energy consumption, latency, and area (Gupta et al. 2011).

This chapter provides an overview of approximate computing and how it can improve efficiency while satisfying the user-defined accuracy constraints. Figure 1 presents the overview of the chapter. First, section "Approximate Arith-

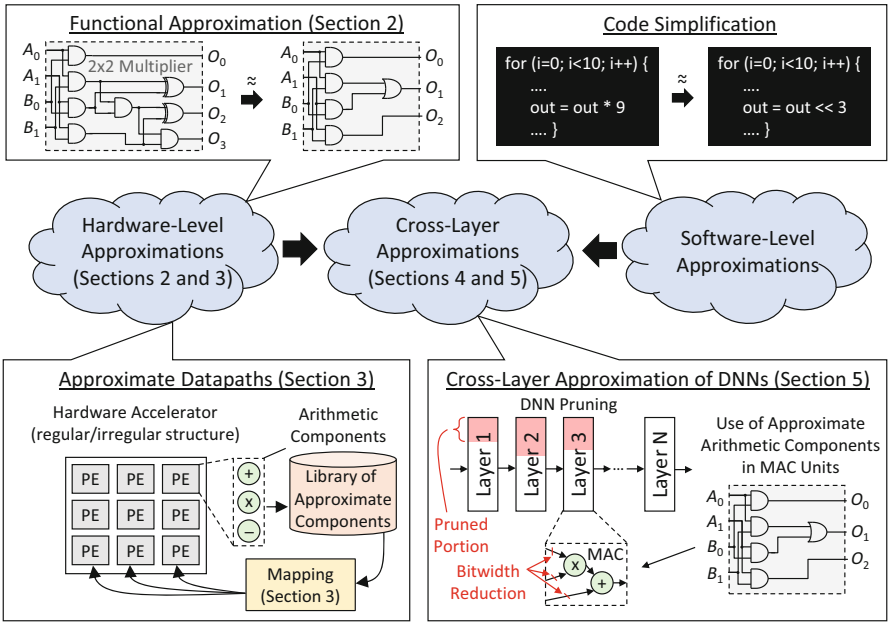


Fig. 1 Overview of the chapter

metic Components" covers techniques for designing approximate arithmetic modules (e.g., adders and multipliers). Then section "Design Methods for Building Approximate Hardware Accelerators: Case Studies for Error-Tolerant Applications" presents design methodologies for automatically generating approximate datapaths for application-specific systems. Section "Cross-Layer Approximations for Error–Tolerant Applications" presents a cross-layer design flow that integrates software-level and hardware-level approximations. Toward the end, section "Case Studies for Improving the Energy and Performance Efficiency of DNN Inference" highlights the effectiveness of cross-layer approximations for deep learning applications, and section "Conclusions" concludes the chapter.

Approximate Arithmetic Components

The use of approximate computing techniques introduces an error, called *approximation error*, which is a measure of the difference between the exact computing solution and the approximate one. This error should allow the designers to reduce the power consumption of the circuits. The approximation error can be introduced on various levels. This section covers various techniques for designing approximate arithmetic components such as adders or multipliers. These components may be used as basic building blocks in high-level approximation methods introduced in the following sections.

Design Methodologies for Approximate Components

This work focuses on approximate arithmetic circuits because they are frequently used in the key applications relevant for approximate computing. The methods for functional approximations can be divided into two categories: (1) manual and (2) automated.

The *manual (ad hoc) methods* are developed for a specific circuit component. In this chapter, examples of manual approximation of two key arithmetic circuits – adders and multipliers – are described. These circuits are widely approximated because they realize key operations in applications requiring low-power processing. MACs, although widely employed, are typically approximated by using separate multiplier and adder units instead of introducing an error to the complex MAC circuit, and thus these circuits are not discussed in the chapter. Designers of manually approximated circuits found some regularities in the design and modified the structure or the truth table of the circuit (Fig. 2a). On the other hand, *automated methods* use general-purpose circuit resynthesis and approximation techniques and enable approximation of arbitrary circuits. These methods start with an original (exact) circuit and, typically iteratively, modify its structure as shown in Fig. 2.

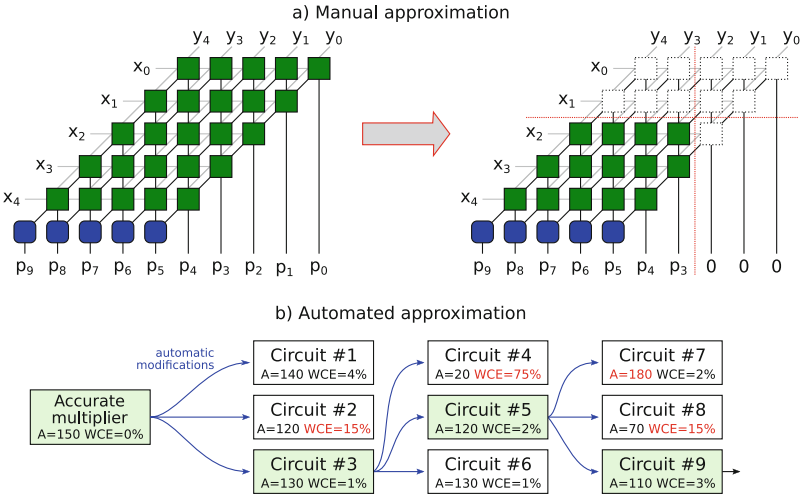


Fig. 2 Examples of two possible approaches for approximation of multiplier: (**a**) manual, where a designer found the rules for effectively omitting cells (Mahdiani et al. 2010), and (**b**) automated iterative approximation of a multiplier having the best area (A) and *worst-case error* (WCE) below 5%

Manual Approximation Methods

Adders: An adder performs the addition of two binary numbers. Two basic implementations are (1) ripple-carry adder (RCA), where the carry of each full adder is propagated to the next full adder, and (2) carry-lookahead adder (CLA), where several units working in parallel generate three signals ("sum," "propagate," and "generate") that are employed to quickly generate the carry-in signals. The CLA has significantly shorter delay than RCA. However, the area and power dissipation of CLA is larger than RCA. Many approximation principles for the adders implemented using one of these two schemes have been proposed in the literature (Jiang et al. 2017). The approximations can be classified into the following classes:

- *Speculative adders* were proposed by Lu (2004). In this architecture, the CLA structure is approximated using prediction of the carry for each sum bit.
- *Segmented adders*, where the addition is divided into n smaller subadders operating in parallel. These subadders have a fixed carry, and their delay is n-times shorter (Mohapatra et al. 2011). An advanced version divides the addition to the carry generation and sum generation, where each summation utilizes the information from the previous carry generation (Zhu et al. 2009).
- *Approximate carry select adders* consist of several subadders. Each subadder is made of two speculative adders – one with carry-in "0" and another with carry-in

"1". The carryout of the first adder is connected to a multiplexor in the next block selecting the output of one of two speculative adders (Du et al. 2012).
- *Approximate full adders* are implemented in LSBs of the adder. For example, the simple use of OR gates instead of full adders and ignoring carries in the LSB part can lead to enormous power and time savings (Mahdiani et al. 2010).

Multipliers: Compared to addition, multiplication is a more complex operation. Generally, it consists of stages of partial product generation, accumulation, and final addition. There are several accurate multiplier architectures. The manually approximated n-bit multipliers are usually derived from one of the following schemes: (1) an array multiplier, where the sum and carry signals are generated by n-bit adders in each of n rows and they are passed to the adders in the next row, and (2) Wallace (or Dadda) tree multipliers dividing the multiplication into layers, where the adders work in parallel without any carry propagation within the layer. The array multiplier is smaller than the tree multiplier but slower. The approximations can be implemented in the following parts of the multipliers (Jiang et al. 2017):

- *Approximation in generating partial products* modifies the submultipliers, which the multiplier is composed of. For example, Kulkarni et al. proposed an approximate 2×2-bit multiplier where only one single entry is altered ($3 \times 3 = 7$ and the remaining ones are correct) (Kulkarni et al. 2011). Larger multipliers are designed using this 2-bit multiplier as a building block.
- *Approximation in the partial product tree* modifies the structure of the multiplies. This approach is utilized in the broken-array multiplier (Mahdiani et al. 2010). This multiplier omits some rows and columns in the array multiplier. A straightforward truncation of LSBs in operands (e.g., the usage of accurate 6-bit multiplier instead of the 8-bit one) also modifies the partial product tree by omitting some partial product cells. The omitting approach can be done in an adaptive way. In the multiplier proposed by Kyaw et al. (2010), the LSB cell function is controlled by the MSBs of operands.
- *Approximation in counters or compressors* in the partial product tree utilizes the tree structure of the multiplier. The key operations in each level are compressions, where 3 bits or 4 bits are compressed to 2 bits (3:2 or 4:2 compressors). These circuits can be approximated, for example, by a substitution of full adders by approximate ones (Momeni et al. 2015).

Automated Approximation Methods

SALSA: The Systematic methodology for Automatic Logic Synthesis of Approximate circuits (SALSA) is an automated approach that turns the approximation synthesis to the standard synthesis task (Venkataramani et al. 2012). A virtual circuit containing an accurate solution, a candidate circuit, and decision circuit (with one output) is constructed. The output is active when the error bound constraint is violated. The *don't care states* are iteratively applied to the approximate solution.

These states are accepted if the output of the virtual circuit remains zero for all input combinations. Thereafter, a traditional don't care-based optimization technique is applied.

SASIMI: Another systematic approach, Substitute-And-SIMplIfy (SASIMI) (Venkataramani et al. 2013), tries to identify signal pairs in the circuit that show the same value with high probability and substitutes one for the other. These substitutions result in some logic to be eliminated from the circuit. In addition to that, the downsizing of gates on critical paths (simplification) may be enabled. Moreover, the connection of the signal pairs using a configurable substitution circuit provides a kind of quality configurable circuit that can dynamically operate at different accuracy levels depending on the application requirements.

ABACUS: In contrast with previous automated methods, Automated Behavioral Approximate CircUit Synthesis operates on the HDL level. It automatically generates approximate circuits directly from the behavioral-level description. In order to perform desired approximations, the method modifies the abstract synthesis tree (AST) using the following operators: (1) simplification of data types, (2) substitution of arithmetic operations by approximate operations, (3) transformation of arithmetic expressions, (4) substitution of variables with constants, and (5) loop transformations. In each iteration of the algorithm, the operations are randomly applied to the accurate circuits, while the error bound is checked after the application (Nepal et al. 2014). The search algorithm is based on a simple hill-climbing algorithm or multi-objective NSGA-II algorithm (Nepal et al. 2017).

AIG-Rewriting: Another automatic synthesis approach uses And-Inverter Graph (AIG)-based rewriting. The AIG is a widely employed representation in logic synthesis. The algorithm identifies the longest paths in the circuit. Then *cuts* are selected by performing cut enumeration on the selected paths. In a logic circuit represented by an acyclic graph, a cut of node n is a set of nodes of the network, called leaves, such that each path from primary inputs to n passes through at least one leaf (Mishchenko et al. 2006). Each cut is replaced by an approximate cut (typically by zero constant) to generate a new candidate circuit. If the candidate meets the error constraints, it is accepted to the next iteration (Chandrasekharan et al. 2016).

ASLAN: Automatic methodology for Sequential Logic ApproximatioN (ASLAN) performs synthesis of approximate sequential circuits. The algorithm tries to identify combinational blocks in a sequential circuit that are amenable to approximations. Then, existing combinational approximation techniques are utilized to obtain a series of approximate versions having different quality levels. A gradient/descent approach is used to iteratively approximate the entire sequential circuit, while the overall error bound is checked using a formal verification approach (Ranjan et al. 2014).

BLASYS: Another methodology for approximate circuit synthesis based on Boolean matrix factorization (BMF) is BLASYS (BMF-based Logic Approximate SYnthesiS). A heuristic algorithm cuts the original circuits to small subcircuits. The truth table of a subcircuit of the design is approximated using BMF to a controllable approximation degree. The results of the factorization are used to synthesize a less complex subcircuit. A subcircuit design-space exploration technique helps to identify the best order for subcircuit approximations. The first version of this methodology (Hashemi et al. 2018) targeted Hamming distance only. However, in the most recent version (Ma et al. 2019), different error metrics are available. This tool is available as an open source at https://github.com/scale-lab/BLASYS.

Evolutionary Algorithm-Based Methods: The logic synthesis is based on small iterative changes of the initial circuit and optimizing the so-called fitness value. Vasicek and Sekanina successfully employed this idea for approximate circuit design by introducing the error metric to the fitness function (Vasicek and Sekanina 2015).

The main advantage of evolutionary approximation is that the heuristic searching algorithm can easily handle arbitrary constraints by giving penalties to the fitness function. Some penalties can be introduced for exceeding error metric (e.g., worst-case arithmetic error, mean relative error [MRE], etc.). Moreover, the evolutionary approximation can handle any application-specific constraint like accurate multiplication by zero (Mrazek et al. 2016) or nonuniform input distribution (Vasicek et al. 2019) as well.

The evolutionary approximation was also used in the context of FPGAs. GRATER tool (Lotfi et al. 2016) employs a genetic algorithm to determine the precision of variables within an OpenCL kernel. By selectively reducing the precision, the number of parallel approximate kernels that can be mapped in the fixed area budget of an FPGA can be increased with respect to the original kernel implementations.

Error Metrics and Evaluation Analysis for Approximate Components

The quality of approximate combinational circuits is typically expressed using one or several error metrics. In addition to the error rate, the average-case as well as the worst-case situation can be analyzed. Among others, the mean absolute error (MAE) and the MRE are the most useful metrics that are based on the average-case analysis. Selection of the right metrics is a key step of the whole design. When an arithmetic circuit is approximated, for example, it is necessary to base the error quantification on an arithmetic error metric. For general logic circuits, where no additional knowledge is available and where there is not a well-accepted error model, Hamming distance or error rate is typically employed.

The following paragraphs summarize the error metrics that have been employed in literature to quantify the deviation between the outputs produced by a functionally

correct design and an approximate design. These metrics are divided into two categories. The category of arithmetic errors consists of metrics that compare integer values of the circuit outputs. The Boolean error metrics are classified as general errors.

Arithmetic Error Metrics

Let $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be an n-input m-output Boolean function that describes the correct functionality (the accurate function) and $f' : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be an approximation of it, both implemented by two circuits, namely, F and F'.

The *worst-case arithmetic error*, sometimes denoted as *error magnitude* or *error significance* (Chan et al. 2013), is defined as

$$e_{wce}(f, f') = \max_{\forall x \in \mathbb{B}^n} |int(f(x)) - int(f'(x))|, \tag{1}$$

where $int(x)$ represents a function $int : \mathbb{B}^m \rightarrow \mathbb{Z}$ returning an integer value of the m-bit binary vector x. Typically, a natural unsigned binary representation is considered, i.e., $int(x) = \sum_{i=1}^{m} 2^i \cdot x_i$. The worst-case error represents the fundamental metric that is useful to guarantee that the approximate output differs from the correct output by at most error bound e.

In the literature, the *relative worst-case error*

$$e_{wcre}(f, f') = \max_{\forall x \in \mathbb{B}^n} \frac{|int(f(x)) - int(f'(x))|}{int(f'(x))} \tag{2}$$

is frequently employed to constrain the approximate circuit to differ from the correct one by at most a certain margin. Note that a special care must be devoted to the cases for which the output value of the original circuit is equal to zero, i.e., the cases when the denominator approaches zero. This issue can be addressed by either omitting test cases when $int(f(x)) = 0$ or biasing the denominator by 1. The first approach is usually employed in the manual approximation methods where the zero results are accurate (Jiang et al. 2017).

The *average-case arithmetic error* (also known as *MAE*) is defined as the sum of absolute differences in magnitude between the original and approximate circuit, averaged over all inputs:

$$e_{mae}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} |int(f(x)) - int(f'(x))|. \tag{3}$$

If the expression in the sum is replaced by the equation for relative error distance, the *mean relative error* is calculated:

$$e_{mre}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} \frac{|int(f(x)) - int(f'(x))|}{int(f'(x))}. \tag{4}$$

Note that the values produced by absolute error metrics e_{mae} and e_{wce} can be very large. Hence, these values can be expressed as a part of the output range using division by $2^m - 1$, i.e., the maximal output value. For example, the worst-case arithmetic error of 64 for an 8-bit output circuit (e.g., 4-bit multiplier) is equal to 25% error.

General Error Metrics

In addition to the arithmetic error metrics, there are metrics that are not related to the magnitude of the output of the correct or approximate circuit. These errors are typically used in approximation of general combinational circuits, where the weight of the output bits is unknown. In these circuits such as coders, decoders, and widely used benchmark circuits (e.g., ISCAS-89, ITC-99, etc.), the output value is not an arithmetic number, and arithmetic errors cannot be calculated (Vasicek and Sekanina 2014, 2015). However, the *error probability* is widely employed in arithmetic circuits as well.

The *error rate*, referred to as the *error probability*, represents the basic measure that is defined as the ratio of input vectors for which the output value differs from the original one:

$$e_{prob}(f, f') = 2^{-n} \cdot \left|\{\forall x \in \mathbb{B}^n : f(x) \neq f'(x)\}\right| \tag{5}$$

In many cases, it is also worth to consider the Hamming distance between $f(x)$ and $f'(x)$. The *worst-case Hamming distance*, denoted also as *bit-flip error* (Chen et al. 2014), is defined as

$$e_{bf}(f, f') = \max_{\forall x \in \mathbb{B}^n} \sum_{i=1}^{m} (f(x) \oplus f'(x))_i \tag{6}$$

and gives the maximum number of output bits that simultaneously output a wrong value. The average number of changed output bits, denoted as the *average Hamming distance*, can be expressed as follows:

$$e_{mhd}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} \sum_{i=0}^{m} (f(x) \oplus f'(x))_i . \tag{7}$$

Quality Evaluation

In the error-metric formulas, the enumeration of all possible input vectors is employed. For a larger number of inputs n, it is not feasible to enumerate $\mathbb{B}^n$. This issue can be solved by (a) enumerating a subset of $\mathbb{B}^n$ or (b) obtaining the exact value using a formal verification approach. The formal verification can be performed by exhaustive simulation (with maximal instruction level of SIMD parallelization) (Hrbacek and Sekanina 2014; Mrazek et al. 2018) or some formal verification technique. These techniques typically construct a virtual miter circuit (consisting of candidate circuit, golden solution, and comparison circuit). Reduced Ordered Binary Decision Diagrams (ROBDD) or SAT conjunctive normal form (CNF)

representation was employed in the area of approximate circuits. The ROBDDs can help the users to determine various error metrics (Hamming distance (Vasicek and Sekanina 2014) or mean or worst-case arithmetic error (Soeken et al. 2016; Vasicek et al. 2017)). However, SAT solving is more effective for complex circuits like multipliers. These solvers allow only to determine if the worst-case error is below some given threshold, but very complex circuits such as 32-bit multipliers or 128-bit adders can be approximated (Češka et al. 2017).

Design Methods for Building Approximate Hardware Accelerators: Case Studies for Error-Tolerant Applications

Approaches for creating approximate components were presented in section "Design Methodologies for Approximate Components". The components are typically organized in libraries (e.g., Shafique et al. 2015; Hanif et al. 2017; Mrazek et al. 2017). These libraries contain from tens to thousands of approximate implementations for each arithmetic operation (e.g., 8-bit multiplication, 16-bit addition, etc.); the user is provided with a broad set of implementation options to reach the best possible trade-off between QoR (quality of results) and energy (or other hardware parameters) at the accelerator level.

If the user wants to use the components in his application, they start with some accurate accelerator, where the accurate operations are replaced by corresponding approximate components. However, it is intractable to find an optimal combination of approximate circuits, even for an accelerator consisting of a few operations. Identifying the most suitable replacements of the arithmetic operations of the target accelerator with the approximate circuit is a complex task. In this chapter, two approaches to this task are presented. As it is a multi-objective optimization problem, there is no single optimal solution; rather, multiple ones typically exist.

The designers are primarily interested in approximate circuits belonging to the *Pareto frontier* that contains the so-called non-dominated solutions. Consider two objectives to be minimized, for example, the mean error and energy. Circuit C1 (Pareto) *dominates* another circuit C2 if (1) C1 is no worse than C2 in all objectives and (2) C1 is strictly better than C2 in at least one objective.

This problem resembles the binding step of high-level synthesis (HLS), whose objective is to (i) map elementary operations of the algorithm to specific instances of components that are available in the component library and (ii) optimize hardware parameters such as latency, area, and power consumption. In the context of approximate circuits, the principal difference and difficulty lie in the QoR evaluation at the accelerator level. Except for some particular cases (e.g., Mazahir et al. 2017a,b), it is in general unknown how the errors propagate if two or more approximate circuits are connected in a more complex circuit. A common approach is to estimate the resulting error using either analytic or statistical techniques, but it usually is a very unreliable approach as seen in Li et al. (2015). If the problem is simplified in such a way that the only approximation technique is truncation, then an optimal number of bits to be approximated can be determined (Sengupta et al. 2017).

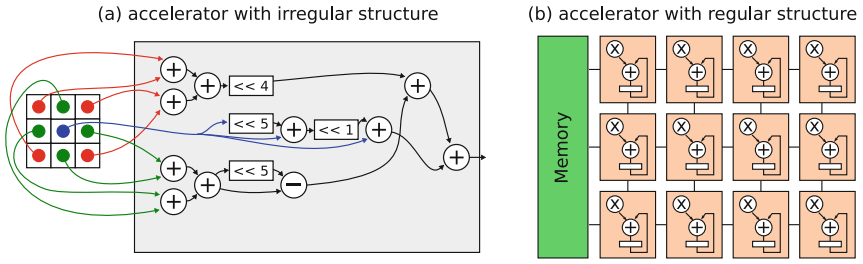


Fig. 3 Two types of accelerators: (**a**) with the irregular structure of fixed Gaussian filter having ten adders and one subtractor with different levels of approximation and (**b**) PE array for inference of neural network having PE employing the same adder and multiplier

Two major types of accelerators are discussed in the following sections. The first one maps every operation on every single hardware component (Fig. 3a). Typical examples of such irregular accelerators are image, video, or signal processing filter pipelines. The automated methodology shown in section "Image and Video Processing Applications" maps the approximate components to the operations. The second type of accelerator shares the hardware component for multiple operations (Fig. 3b). The sharing occurs, for example, in neural network inference acceleration. The layer operations are executed on a PE array where each processing element handles multiple different convolutions. In this case, additional constraints (e.g., only a few approximate PE arrays, order of the layers) must be satisfied. However, the structure of the neural network may be modified simultaneously. The approximation of neural networks is discussed in section "Deep Neural Networks (DNNs)".

Image and Video Processing Applications

Many different operations are employed in a typical image processing pipeline. In this section, three accelerators of different complexities that are typically used as benchmarks in image processing will be considered. In particular, Sobel edge detector (Sobel ED) (five operations), Gaussian filter with fixed coefficients (fixed GF) (11 operations), and generic Gaussian filter (generic GF) (17 operations) working on the 3×3 filter kernel were chosen.

Since there are hundreds to thousands of different approximate components for each operation and the complexity is exponential, the number is enormous. While the approximation of the five-operation accelerator is solvable by an exhaustive enumeration of all possible configurations, the accelerator consisting of 17 operations represents a nontrivial problem.

AutoAx Methodology

To address the approximate component binding problem, the authors proposed the AutoAx methodology (Mrazek et al. 2019a) that enables fast QoR and hardware cost

evaluation by means of machine learning algorithms and heuristic multi-objective searching algorithm.

The methodology requires the following inputs from the user: a hardware description of the chosen accelerator, corresponding software model, and training (benchmark) data. Hierarchical hardware as well as software models are expected in order to be able to replace relevant operations with their approximate versions and to evaluate how this change affects the QoR. Approximate circuits are taken from a library, in which each of them is fully characterized and many approximate implementations exist for each operation.

Let the accelerator contain n operations that can be implemented using some approximate circuits for the library. A *configuration* is referred to as a particular assignment of approximate circuits from the library to n operations of the accelerator. The goal of the methodology is to find a Pareto set of *configurations* where the design objectives to be optimized are QoR (e.g., SSIM, PSNR, etc.) and hardware cost (e.g., area, delay, power, or energy).

The whole process consists of three steps as illustrated in Fig. 4.

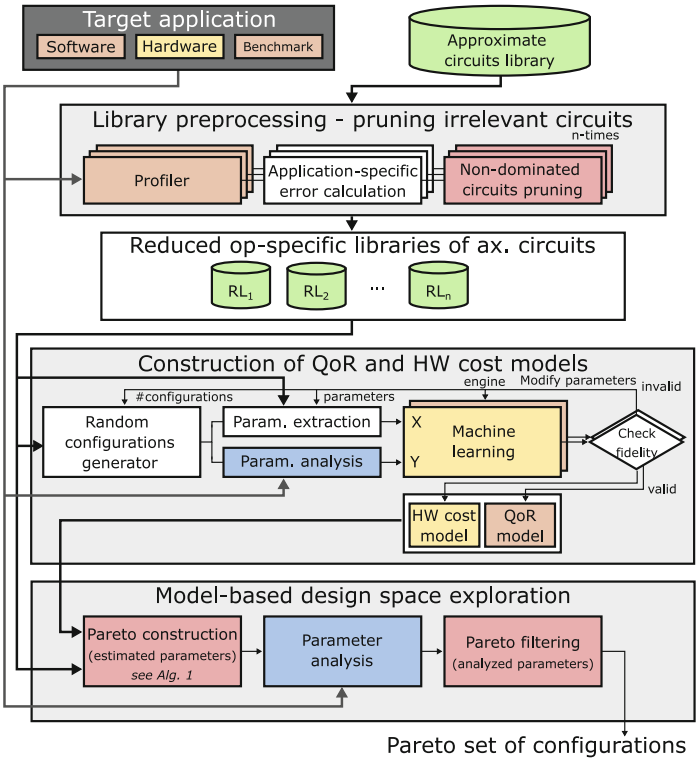


Fig. 4 Overview of the proposed *autoAx* methodology

Step 1: The library of the approximate circuits is preprocessed in such a way that clearly irrelevant circuits are removed. Irrelevant circuits are identified on the basis of their quality (measured with respect to a particular application) and hardware cost.

Step 2: Computational models enabling to estimate QoR and hardware cost are constructed by means of some machine learning algorithm. A small (randomly selected) subset of possible configurations is used for learning of the computational models.

Step 3: The Pareto frontier reflecting QoR and HW cost is constructed. To quickly remove as many low-quality solutions as possible, the construction algorithm employs the values estimated by the proposed models. The final Pareto front is then constructed using precisely computed QoR and hardware parameters by means of simulation and synthesis.

Library Preprocessing For each operation of the accelerator, a suitable subset of approximate circuits is separately identified in the library by means of benchmark data. For example, if the kth operation of the accelerator is 8-bit addition, then the objective of this step is to identify approximate 8-bit adders that form the Pareto front with respect to a suitable error metric (score) and hardware cost. The authors propose to base the selection on probability mass function (PMF) of the given operation which can be easily determined by simulation of the accelerator on benchmark data.

This process can be formalized as follows. Let I denote a set of all possible combination of values from the benchmark dataset that can occur on the input of kth operation $M(x_1, x_2, \dots)$, $x \in I$, $k = 1 \dots n$. Then, $D_k : I \rightarrow \mathbb{R}$ denoting the PMF of this operation is defined as $D_k(i_1, i_2, \dots) = Pr(x_1 = i_1 \wedge x_2 = i_2 \wedge \dots)$. This function is used to determine a score (weighted mean error distance) of an approximate circuit $\widetilde{M}$ implementing kth operation as follows: $\mathrm{WMED_k}(\widetilde{\mathrm{M}}) = \sum_{\forall i \in I} D_k(i) \cdot |M(i) - \widetilde{M}(i)|$. For each operation of the accelerator, this score is then used together with hardware cost to identify only those approximate circuits (i.e., 8-bit adders in our example) that are lying on a Pareto frontier.

Model Construction Since the synthesis and simulation are typically very time-consuming processes, it is intractable to use them to perform the analysis of hardware cost and QoR for every possible configuration of the accelerator. To address this issue, construction of two independent computational models is proposed – one for estimating QoR and a second for estimating hardware parameters. The estimation is based on the parameters of approximate circuits belonging to one selected configuration.

The models are constructed independently using a suitable supervised *machine learning algorithm* (regression problem). The learning process is based on providing example input–output pairs. In our case, each input–output pair corresponds with a particular configuration as shown in Fig. 5. One input is represented by a vector X,

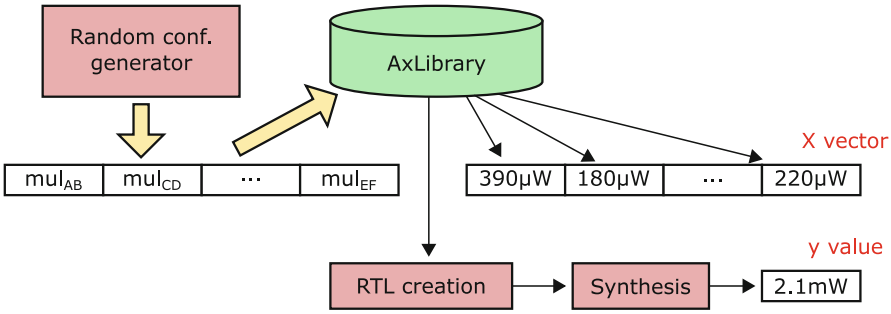


Fig. 5 Construction of training/testing set for ML model of hardware cost. The X-vector is extracted from the library (e.g., power, PDP [power–delay product] for HW cost, e_{mae}, e_{wce} for QoR), and the y-value is calculated using synthesis chain

which contains a subset of hardware or quality parameters of each approximate circuit realizing one of the operations as defined by the configuration. The output is a single scalar value y of QoR or hardware cost that is obtained by simulation and synthesis of the concrete accelerator with the given configuration. A training set typically containing from hundreds to thousands of configurations is generated for learning.

The goal of this step is to obtain high-quality models. A set of configurations different from the training set is used to determine the quality of the model and avoid *overfitting*, when the estimated values correspond too closely or exactly to training output values, and the model may, therefore, fail in fitting additional data. Typically, the accuracy is optimized by the machine learning algorithms. However, as the models are used for determining a relation between two different configurations, it is not necessary to focus on the accuracy. *Fidelity* (aka *monotonicity* (Bailey et al. 2007)) is considered as the optimization criterion that maximizes the fidelity of the model. The fidelity describes how often the estimated values are in the same relation ($<$, $=$ or $>$) as the real values for each pair of configurations. If the fidelity of the constructed model is insufficient, the parameters of the chosen learning algorithm should be tuned, or a different learning engine should be selected.

Model-Based Design Space Exploration In this step, the Pareto frontier containing those configurations that show the best trade-offs between QoR and hardware cost is constructed. In order to avoid time-consuming simulation and synthesis, the construction is divided into two stages. In the first stage, the computational models that were developed in the previous step are used to build a pseudo-Pareto set of potentially good configurations. In the second stage, based on the configurations forming the pseudo-Pareto set, a set of approximate accelerators is determined, fully synthesized, and analyzed by means of a simulator and benchmark data. A real QoR and real hardware cost is assigned to each configuration. Finally, these real values are used to construct the final Pareto set.

Algorithm 1 Pareto set construction

INPUT: RL – set of libraries, $RL = \{RL_1, RL_2, \cdots, RL_n\}$,
M_{HW} – HW costs model, M_{QoR} – quality model
OUTPUT: Pareto set $P \subseteq RL_1 \times RL_2 \times \cdots \times RL_n$

```
function HeuristicParetoConstruction(RL, M_QoR, M_C)
    Parent ← PickRandomlyFrom(RL_1 × RL_2 × ··· × RL_n)
    P ← ∅
    while ¬TerminationCondition do
        C ← GetNeighbour(Parent)
        e_QoR ← M_QoR(C)                          ▷ Estimate the quality of C
        e_HW ← M_HW(C)                            ▷ Estimate the HW costs of C
        if ParetoInsert(P, (e_QoR, e_HW), C) then
            Parent ← C
        else if StagnationDetected then           ▷ Parent not changed in last k iterations
            Parent ← PickRandomlyFrom(P)
        end if
    end while
    return P
end function
```

Although the first step reduced the number of possible configurations, the number of combinations may still be enormous especially for complex problems consisting of tens of operations. Therefore, the authors proposed an iterative heuristic algorithm (Algorithm 1) to construct the pseudo-Pareto set. The algorithm is a variant of stochastic hill climbing which starts with a random configuration (denoted as $Parent$), selects a neighbor at random (denoted as C), and decides whether to move to that neighbor or to examine another. The neighbor configuration is derived from $Parent$ by modifying a randomly chosen item of the configuration (i.e., another circuit is picked from the library for a randomly chosen operation). The quality and hardware cost parameters of C (e_{QoR} and e_{HW}) are estimated by means of appropriate estimation models. If the estimated values dominate those already present in Pareto set P, configuration C is inserted to the set, the set is updated (operation ParetoInsert), and the candidate is used as the $Parent$ in the next iteration. In order to avoid getting stuck in a local optimum, restarts are used. If the $Parent$ remains unchanged for k successive iterations, the $Parent$ is replaced by a randomly chosen configuration from P. The quality of the resulting Pareto set depends on the fidelity of the estimation models and on the number of allowed iterations. The higher fidelity, the better results. The number of iterations depends on the chosen termination condition. It can be determined by the size of P, execution time, or the maximum allowed number of iterations.

Results

The results are divided into two parts. Firstly, a detailed analysis of the results for the Sobel ED is provided to illustrate the principle of the proposed methodology. In the second part, only the final results are discussed due to the complexity of these problems and a limited space.

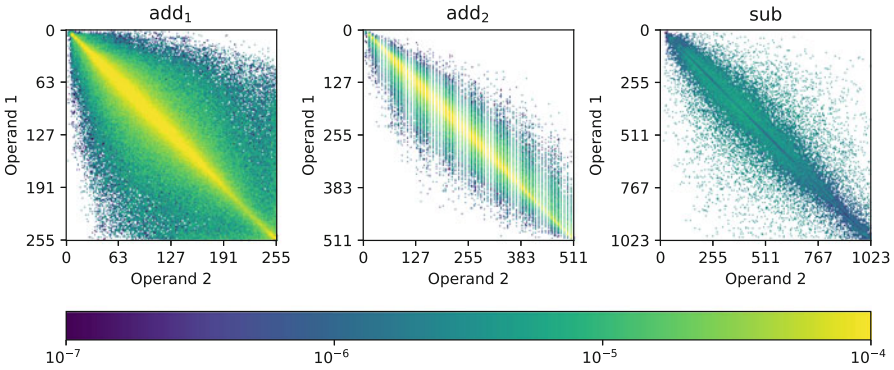


Fig. 6 PMF of operations in the Sobel ED

Sobel Edge Detector To eliminate irrelevant circuits from the library, a score is calculated for each circuit in the library. Firstly, the target accelerator is profiled with a profiler which calculates the PMF D_k for all operations (Fig. 6). Note that add_3 (resp. add_4) has almost identical PMF with add_1 (resp. add_2). Figure 6 shows that operand values (neighbor pixels) are typically very close. In the plot dealing with D_{add_2}, one can see regular white stripes caused by shifting of the second operand.

Using the obtained probabilities, the $WMED_k$ errors are calculated for all approximate circuits implementing kth operation. Then the components are filtered out. The process is guided by $area$ and $WMED_k$ parameters of the isolated circuits and keeps only Pareto-optimal implementations. At the end of this process, the number of circuits in reduced libraries is $|RL_{add_1}| = 35$, $|RL_{add_2}| = 32$, $|RL_{add_3}| = 37$, $|RL_{add_4}| = 33$, and $|RL_{sub}| = 36$.

The next step in the methodology is to construct models estimating SSIM and hardware parameters using parameters of the circuits belonging to one selected configuration. The $WMED$ of all employed circuits is employed as the input vector for the QoR model. For the hardware model, the input vector is $power, area$, and $delay$ of all circuits. Several learning engines are compared to identify the most suitable one for our methodology (1500 configurations for learning and 1500 configurations for testing were randomly generated using the reduced libraries).

The considered learning engines are the regression algorithms from *scikit-learn* tool for Python. Additionally, a naïve models are constructed for area ($M_a(C) = \sum_{\forall c \in C} area(c)$) and for SSIM ($M_{SSIM}(C) = -\sum_{\forall c \in C} WMED_k(c)$) to test if SSIM correlates with the cumulative arithmetic error and if the area correlates with the sum of areas of all employed circuits. These simple models are also considered in the comparisons.

Table 1 shows the fidelities for all constructed models when evaluated on the training and testing datasets. The best result for the testing datasets is provided by a random forest consisting of 100 different trees. The correlation between estimated and real area is shown in Fig. 7. The naïve models exhibit unsatisfactory results

Table 1 The fidelity of models for Sobel edge detector constructed by different learning engines

Learning algorithm	SSIM		Area	
	Train	*Test*	*Train*	*Test*
Random forest	99%	**96%**	97%	**92%**
Decision tree	**100%**	95%	**100%**	86%
K-neighbors	94%	94%	91%	89%
Bayesian ridge	90%	90%	91%	91%
Partial least squares	90%	90%	91%	90%
Lasso	90%	90%	91%	90%
Naïve model	–	90%	–	88%
AdaBoost	90%	90%	90%	88%
Least-angle	90%	90%	71%	72%
Gradient boosting	89%	89%	92%	91%
MLP neural network	86%	83%	92%	91%
Gaussian process	**100%**	71%	**100%**	55%
Kernel ridge	41%	42%	90%	90%
Stochastic gradient descent	24%	25%	75%	74%

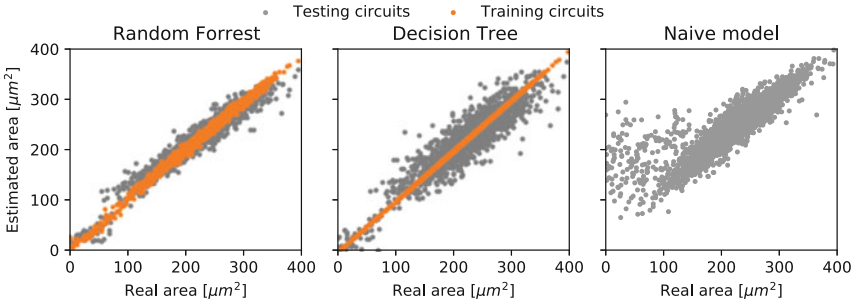


Fig. 7 Correlation of estimated area and real area obtained by synthesis tool for the selected learning engines used in Sobel ED experiment

especially for small resulting approximate accelerators. In the analysis of some of these cases in detail, the authors observe that the inaccuracy was typically caused by the last operation in the application (i.e., *sub*). As this operation shows a big error, it is significantly simplified by the synthesis tool, and as a consequence of that, many other circuits are removed from the circuit because their outputs are no longer connected to any component. Hence, the real area of these circuits was significantly smaller than the area calculated using the library. Due to this elimination, machine learning methods based on conditional structures (e.g., trees) exhibit better performance than methods primarily utilizing algebraic approaches (e.g., MLP NN).

The impact of input parameters on the model quality was analyzed. Including different error metrics such as the error variance did not improve the fidelity of QoR models. In contrast, omitting power and delay in hardware modeling led to 2% lower fidelities of these models on average.

The quality of the proposed heuristic algorithm that was used for Pareto frontier construction is evaluated now. Because of a low number of operations in Sobel ED, all possible configurations derivable from the reduced libraries RL_k (i.e., $4.92 \cdot 10^7$ configurations in total) can be evaluated. The proposed algorithm with a reasonable number of evaluations (10^5) could find the suboptimal solutions that are very close to the optimal ones. The proposed algorithm found solutions in three orders of magnitude closer to the optimal than the standard random search.

More Complex Pipelines The methodology was also applied to obtain approximate implementations of two versions of Gaussian image filter (fixed GF and generic GF). After profiling this accelerator and reducing the library of approximate circuits accordingly, random forest-based models of QoR and hardware parameters were created using 4000 training and 1000 testing randomly generated configurations. In the case of fixed GF, the fidelity of the area estimation model is 87% for hardware parameters and 92% for QoR. The fidelity of both models of generic GF is 89%. If the synthesis and simulations run in parallel, the detailed analysis of one configuration takes 10 s on average, and the model-based estimation of one configuration takes 0.01 s on average.

The Pareto construction algorithm evaluated 10^6 candidate solutions. On average, 39 iterations were undertaken to find a new candidate suitable for the Pareto front.

Table 2 shows the size of the design space after performing particular steps of the proposed methodology. For example, there are $7.15 \cdot 10^{63}$ configurations in the generic GF design space. The elimination of irrelevant circuits in the library reduced the number of configurations to $3.75 \cdot 10^{23}$. The number of configurations is enormous, and it would take 10^{17} years to analyze them. In contrast, the construction of 4000 random solutions for training of the models takes approximately 11 h, 10^6 iterations of the proposed Pareto construction algorithm employing the models takes 3 h, and the remaining 1000 configurations are analyzed in 3 h. Finally, approximately 100 configurations that are Pareto optimal in terms of area, SSIM, and energy are selected. In total, the proposed approach takes 17 h on a common desktop. Hypothetically, if the analysis would be used instead of the estimation model in the Pareto front construction, the analysis of 10^6 configurations would take 115 days.

Figure 8 compares resulting Pareto fronts obtained using the proposed methodology (orange line), the RS-based Pareto front construction algorithm (blue line),

Table 2 Size of the design space after performing particular steps of the proposed methodology

Application	# configurations			
	All possible	*Lib. preprocessing*	*Pseudo-Pareto*	*Final Pareto*
Sobel ED	$1.96 \cdot 10^{15}$	$4.92 \cdot 10^7$	335	62
Fixed GF	$7.35 \cdot 10^{34}$	$1.73 \cdot 10^{16}$	1166	132
Generic GF	$7.15 \cdot 10^{63}$	$3.75 \cdot 10^{23}$	946	102

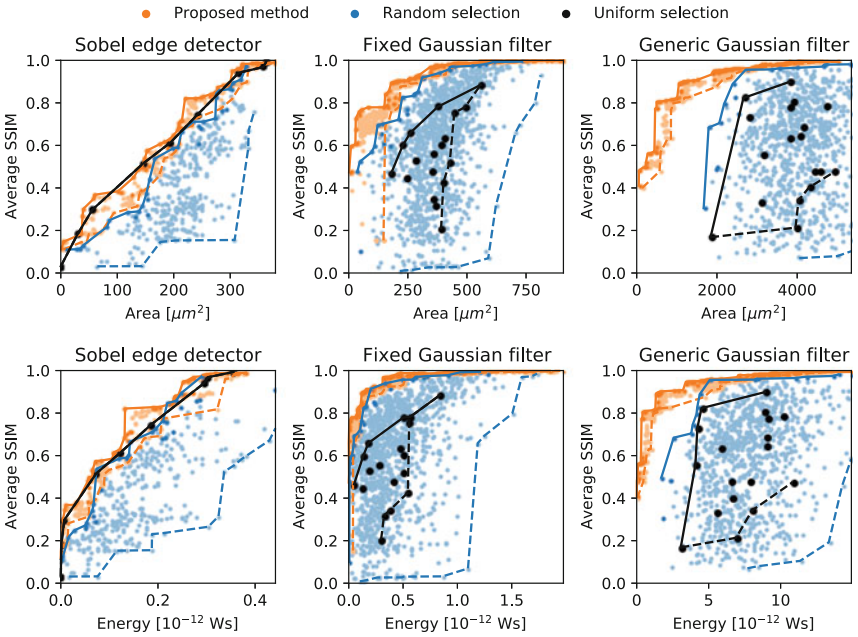


Fig. 8 Pareto fronts showing best trade-offs between SSIM, area, and energy obtained using three methods (orange, the proposed method; blue, random search; black, uniform selection) for three approximate accelerators

and the uniform selection approach (black line). The uniform selection approach is a manual selection method which one would probably take if no automated design methodology is available. In this method, particular approximate circuits are deterministically selected to exhibit the same error WMED (relatively to the output range). Figure 8 shows that this method provides relevant results only for accelerators containing a few operations. The randomly generated configurations (blue points) were obtained from a 3-h run of the random configuration generation-and-evaluation procedure. They are included in these plots in order to emphasize high-quality solutions obtained by the proposed method.

Deep Neural Networks (DNNs)

The neural networks have come to be an important part not only of supercomputers but even small embedded systems realizing machine learning *on the edge*. The structure of hardware accelerators is different in contrast to the typical signal processing pipeline introduced in the previous section. The accelerator is organized as an array of processing elements. An arbitrary approximate component cannot be assigned to any layer of DNN because the number of the *tiles* (parts of the PE array) is limited. A significant proportion of energy is consumed by the computational path consisting primarily of multiplications (25–50% (Judd et al. 2018)).

The energy cost of the computational path can be reduced using approximate computing because the DNNs exhibit error resilience property. The standard approach is to assign the approximate components to the layers while considering PE array construction constraints. The promising alternative approach is to construct the architecture with approximate components (neural architecture search) (Pinos et al. 2021), but this approach is computationally intensive. Therefore, the authors proposed ALWANN methodology (Mrazek et al. 2019b) that assigns the approximate components with the help of a multi-objective evolutionary algorithm.

ALWANN Methodology

ALWANN requires the following inputs from the user: already trained NN being subject of the approximation, a library of basic approximate components (adders, multipliers), and knowledge of the architecture of the final HW accelerator. Two HW-based architectures (as discussed in the previous section) are considered in this work: *pipelined* and *power-gated* arrays. For simplicity, the MAC units will be implemented using accurate addition and approximate multiplication, but approximate addition can be introduced as well in general. Let $L = \{L_1, L_2, \ldots\}$ be a set of indexes of convolutional layers of NN and M be a set of available approximate w-bit multipliers. The user should specify the number of different tiles $|T|$ the accelerator will consist of. Typically, $|T| < |L|$ and $w = 8$ is sufficient. Each tile's NFU consists of the array of the same MAC units. Each layer L_i is supposed to be executed on a single tile T_j.

The method outputs a set of AxNNs (modified original NN together with the corresponding configuration of the HW accelerator tiles) that are Pareto optimal with respect to the energy consumption and classification accuracy. The approximations are introduced to the original NN by *replacement* of the accurate convolutional layers by approximate ones together with *weight tuning*. Considering the structure of the HW-based accelerator, two tasks are solved simultaneously. The methodology looks for the *assignment of the approximate multipliers* to MACs in SA tiles $T = \{T_1, T_2, \ldots\}$, i.e., mapping $map_{TM} : T \rightarrow M$, and for the *assignment of the convolutional layers* to SA tiles, i.e., mapping $map_{LT} : L \rightarrow T$. The weights in each layer are updated according to the properties of a particular multiplier assigned to the tile which computes the output of the layer.

The overall architecture of the proposed framework is shown in Fig. 9. The framework expects that a fully specified NN is available (typically in *protobuf* format). If not already done, the NN is firstly quantized to avoid floating point MAC operations. The protobuf specification of the quantized NN is then edited, and all convolutional layers are replaced by approximate ones. This step is necessary to have the ability to specify which multiplier should be used to calculate the output of the MACs separately for each layer. To obtain a Pareto set of various AxNNs, the authors propose to use multi-objective genetic algorithm (NSGA-II) (Deb et al. 2002). The algorithm maintains a population of $|P|$ candidate solutions represented as a pair (map_{TM}, map_{LT}). The search starts from an initial population which

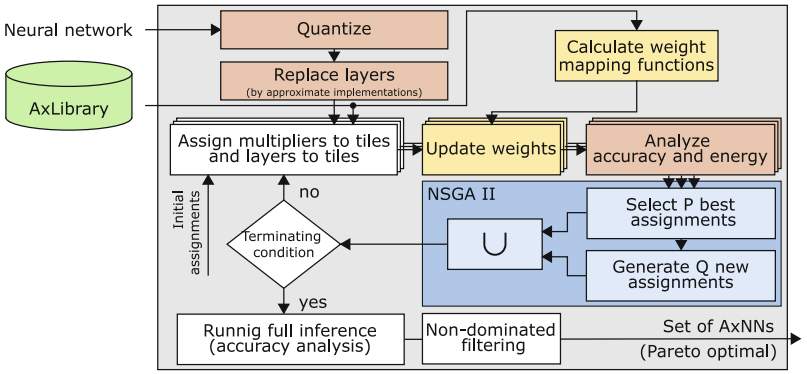


Fig. 9 Overall architecture of ALWANN framework

is generated either deterministically or randomly. The candidate solutions are iteratively optimized with respect to the accuracy of AxNN and energy required to perform one inference. For each candidate solution, a corresponding protobuf is created. This step includes the assignments of the multipliers to each approximate layer according to the map_{TM} and map_{LT} and refinements of the weights in each approximate layer depending on the chosen multiplier. Then, energy as well as quality of the obtained AxNN is evaluated on a subset of training data. The usage of the subset of training data reduces the testing time, and it simultaneously avoids overfitting. At the end of the optimization process when a terminating condition is met (typically the maximum number of allowed iterations is exceeded), the quality of the candidate solutions is evaluated using the complete training set. Solutions whose parameters are dominated by at least one other solution are filtered out.

In contrast to the AutoAx methodology for a generic pipeline, the introduced ALWANN approach does not employ ML models. The HW cost is estimated as a sum of energies because the chained approximation does not affect the overall HW cost in direct result sharing. Similarly, a fast evaluation of quality (classification accuracy) has been proposed. Since many approximate units work in parallel, this task can be performed on a GPU in a reasonable time. The common part of both methodologies is that they use a multi-objective genetic heuristic algorithm (a variant of NSGA-II).

Representation of Candidate AxNNs Each candidate solution is uniquely defined by a pair (map_{TM}, map_{LT}). The authors propose to use an integer-based encoding. The first part map_{TM} is encoded using $|T|$ integers where each integer corresponds with index i of multiplier $M_i \in M$. Similarly, the second part is encoded using $|L|$ integers where each integer determines index i of a tile $T_i \in T$ that will be used to compute the output of the corresponding layer. Depending on the structure of the chosen HW accelerator, additional restrictions may be applied. For example, for pipelined architecture, a rule that the tiles are assigned consequently can be constrained.

Evaluation and Experiments

To evaluate ALWANN, TensorFlow framework was extended to support approximate quantized layers. The extension has been published as open source at https://github.com/ehw-fit/tf-approximate. The tool flow is shown in Fig. 10. At the beginning, the common `QuantizedConv2D` layers are replaced with newly introduced `AxConv2D` layers. The remaining part follows the scheme already described in section "ALWANN Methodology". For the evaluation, ResNet networks (v1 with non-bottleneck blocks) (He et al. 2015) were chosen and trained to recognize images from CIFAR-10 dataset. The library of approximate multipliers consists of all 36 eight-bit fully characterized multipliers from the publicly available EvoApproxLib library (Mrazek et al. 2017).

Figure 11 shows the quality of AxNNs obtained using ALWANN from the original ResNet-8. The results are compared with three configurations of AxNNs mentioned in the previous section, especially to *uniform structures* widely used in the recent literature. The proposed method delivers significantly better AxNNs compared to the manually created AxNNs. The uniform structure (all layers approximated) widely used in the literature (see, e.g., Sarwar et al. 2018; Mrazek

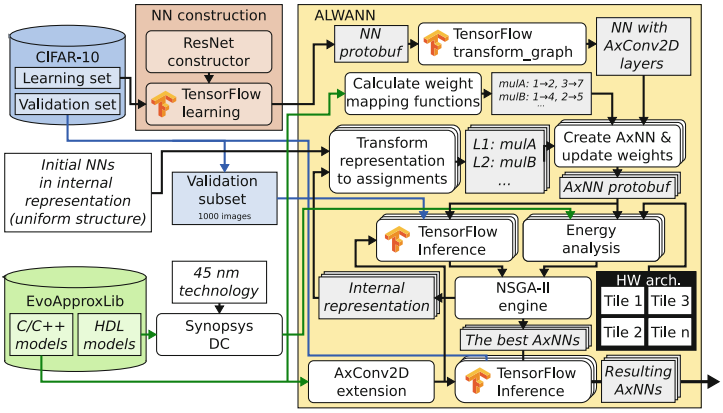


Fig. 10 Our tool flow for retraining-less approximation of ResNet neural network

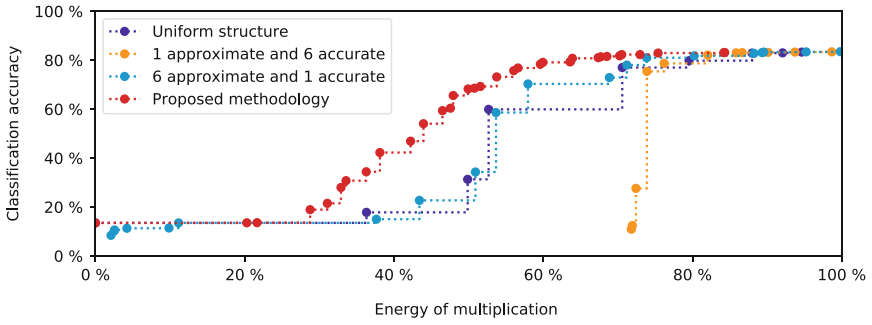


Fig. 11 Comparison of AxResNet-8 approximate neural networks constructed by means of proposed algorithm and NNs having a regular structure

et al. 2016) achieves results comparable to AxNNs with all but one approximated layers. In contrast to that, AxNN with one approximate layer leads to significantly worse results because of small energy saving. The proposed method provides **better trade-offs between the accuracy and energy consumption** in comparison with the uniform NN architectures reported in the state-of-the-art works.

A bottleneck of the algorithm was the expensive simulation of approximate multipliers on CPU. Although the multipliers were cached, our single core application has 10× lower performance than vectorized accurate multiplication. Since one inference of full dataset took 54.5 min, 7.5 days were needed for the construction of the approximate neural network. This problem was addressed in Vaverka et al. (2020) by employing approximate operations on a GPU. The speed was improved more than 200×, and the most complex 50-layer NN can be approximated in less than 2 h on a single GPU.

Overall Results Table 3 gives some parameters of the best AxNNs constructed using the proposed tool. The following parameters are reported for each network: relative accuracy and total and relative energy of convolutional operations. The relative values are calculated with respect to the original quantized (8-bit) ResNet. The quality of the obtained AxNNs for ResNet-50 is very promising. If a target application is able to tolerate 1% accuracy drop (from 89.15% to 88.1%), for example, more than 30% of energy can be saved. The evaluation across different

Table 3 Parameters of selected AxNNs implementing dataset CIFAR-10. The relative values are compared to accurate 8-bit neural network, and total energy is related to the energy of one accurate multiplication E_M

AxNN	Accuracy	Relative accuracy	Relative energy	Total energy [$\times E_M$]
AxResNet-50	*89.15%*	*100.00%*	*100.00%*	*120.27 M*
	89.30%	100.17%	83.29%	100.17 M
	89.08%	99.92%	78.47%	94.37 M
	88.69%	99.48%	77.97%	93.77 M
	88.58%	99.36%	70.02%	84.21 M
	88.10%	98.82%	69.12%	83.13 M
	87.77%	98.45%	67.36%	81.02 M
	85.00%	95.34%	57.74%	69.45 M
AxResNet-14	*85.55%*	*100.00%*	*100.00%*	*35.33 M*
	85.87%	100.37%	80.32%	28.38 M
	85.42%	99.85%	74.34%	26.27 M
	84.77%	99.09%	70.85%	25.04 M
	83.82%	97.98%	64.64%	22.84 M
AxResNet-8	*83.26%*	*100.00%*	*100.00%*	*21.18 M*
	83.16%	99.88%	84.31%	17.86 M
	81.79%	98.23%	70.23%	14.87 M
	79.11%	95.02%	59.95%	12.70 M
	75.71%	90.93%	56.04%	11.87 M

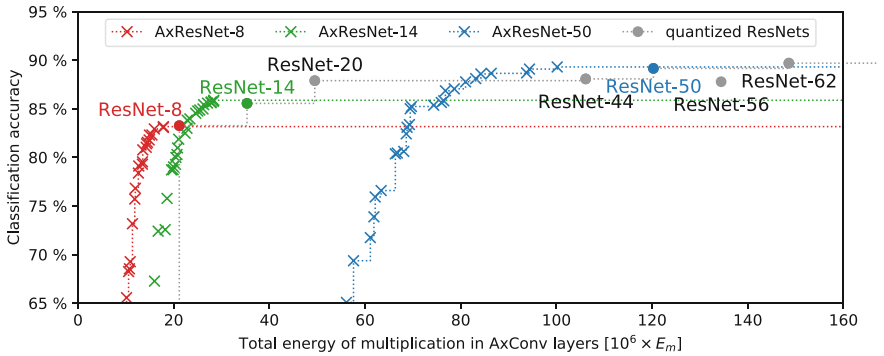


Fig. 12 Comparison of proposed AxNNs (crosses) with accurate quantized NNs (points) – the energy reports the energy of multiplications in the convolutional layers, while E_m is energy of one multiplication. Gray points represent quantized networks that were not approximated (complexity reduction)

architectures shows that it is not advantageous to use AxNNs having more than 4% (2% for AxResNet-14) degradation of accuracy for AxResNet-50, because AxResNet-14 (AxResNet-8) exhibit the same quality but lower energy.

Complete overview of the best obtained AxNNs having accuracy higher than 65% is provided in Fig. 12. In addition to the parameters of the AxNNs for three ResNet architectures discussed so far, the parameters of all possible ResNet architectures up to 62 layers (see the dots) are included, namely, ResNet-20, ResNet-44, ResNet-56, and ResNet-62, that have been trained in the same way as the remaining ResNet NNs. These NNs have been obtained by reducing the number of layers by multiples of six, i.e., at block boundaries. In total, seven different ResNet architectures are included. As evident, our method is able to produce significantly more design points; more than 40 points are produced from a single ResNet. Moreover, majority of the design points are unreachable by simple reduction of the number of layers (see the blue crosses vs. dot symbols). Considering the computational complexity, each ResNet instance must be trained separately. For complex structures, training of a new structure can take several days or weeks on computer clusters.

Comparison with State of the Art (SoA) Table 4 compares the proposed approach with the state-of-the-art approaches for reducing the energy of NNs that have been evaluated on CIFAR-10 dataset. Table 4 includes reported energy reduction and accuracy degradation. The requirement for retraining, uniformity of the architecture, and complexity of NN are also provided. In contrast with multiplier-less multiplication where only four different architectures were proposed (Sarwar et al. 2018), our approach allows to find a new design points with high granularity without retraining. Besides that, our approach enabled the authors to find AxNNs with low energy exhibiting low accuracy, e.g., <80%. Even these solutions can be beneficial, for example, as one of initial stages of some progressive chain classifier (Choi and Venkataramani 2019).

Table 4 Comparison of automated NN approximation methods: architectural parameters, energy and accuracy reduction reported on CIFAR-10

Approach	Retrain./Unif./Depth	Energy/Accuracy
Venkataramani (Venkataramani et al. 2014)	Yes/no/low	−22%/−0.5%
		−26%/−2.5%
Sarwar (Sarwar et al. 2018)	Yes/yes/high	−33%/−1.8%
He (He et al. 2015)	Yes/yes/high	$-12\%/-1.2\%_{\ 50\rightarrow 44}$
		$-71\%/-4.0\%_{\ 50\rightarrow 14}$
		$-48\%/-2.7\%_{\ 14\rightarrow 8}$
ALWANN (Mrazek et al. 2019b)	No/no/high	$-30\%/-0.6\%_{\ \text{AxRN-50}}$
		$-30\%/-0.9\%_{\ \text{AxRN-14}}$
		$-30\%/-1.7\%_{\ \text{AxRN-8}}$

Cross-Layer Approximations for Error-Tolerant Applications

Section covered effective techniques for hardware-level approximations. However, improvements can be achieved through software-level approximations as well. Therefore, this section presents a methodology for combining software- and hardware-level approximations to achieve significant improvements in the performance of a system by leveraging the error resilience characteristics of the application. After presenting a generic methodology for cross-layer approximation in section "Methodology for Combining Hardware- and Software-Level Approximations", section "Cross-Layer Methodology for Optimizing DNNs" covers a methodology specifically designed for optimizing DNN-based systems.

Methodology for Combining Hardware- and Software-Level Approximations

To design highly resource-efficient systems by exploiting the error resilience of the applications, it is necessary to employ approximations at both the software and the hardware levels. Figure 13 presents a cross-layer methodology for designing such systems where approximations are systematically employed across the hardware and the software stacks (Shafique et al. 2016). First, different approximation possibilities are explored at individual levels to short-list a set of Pareto-optimal points. This set can be a result of only a single type of approximation (e.g., functional approximation of computational modules) or multiple types (e.g., functional approximation of computational modules and voltage-scaling in on-chip memory) from the same level. In case of multiple types, efficient design space exploration methodologies are required to find the configurations that offer the best quality-efficiency trade-off. Once the points at individual levels are selected, they are forwarded for a cross-layer design space exploration to select a combination that offers the best efficiency while meeting the user-defined quality constraints. The

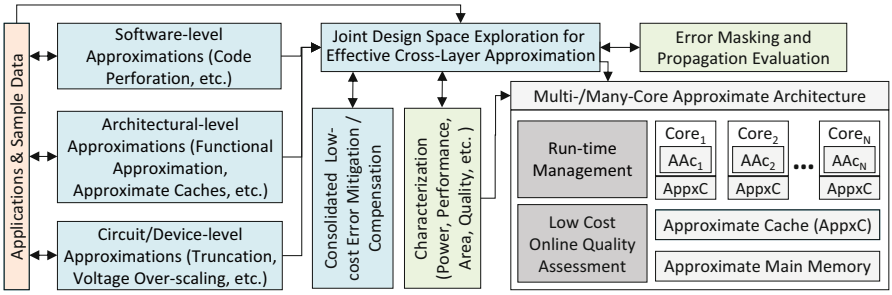


Fig. 13 A cross-layer approximation methodology for designing highly efficient systems (Shafique et al. 2016)

joint exploration is supported by fast error estimation methodologies that take into consideration error masking and propagation properties of approximations to estimate the joint effect of different approximations on the output quality. Low-cost error compensation modules can be employed to compensate for a portion of the quality loss. A set of points are then forwarded to the characterization stage for estimating the performance characteristics of the designs, e.g., power/energy and area. These characteristics are then used together with quality estimates to identify the optimal configurations that offer the best quality-efficiency trade-off. The system can also be equipped with an online approximation management module that can configure the system based on the user requirements and/or run-time conditions to maximize efficiency gains.

Note that most of the works in the domain of approximate computing are focused toward designing techniques for a specific layer of the computing stack, and only a limited amount of research has been carried out on cross-layer methodologies that systematically employ approximations at all the abstraction layers to achieve optimal quality-efficiency trade-off. This is mainly because there are several critical challenges in realizing an effective cross-layer methodology such as the one shown in Fig. 13. A few of these challenges are listed below:

- Designing methodologies/analytical models for evaluating the error masking and propagation characteristics of approximations, specifically for projecting them across layers of the computing stack.
- Designing techniques for efficiently estimating the overall performance characteristics of an approximated system that has different types of approximations deployed at different layers.
- Developing methods for low-cost consolidated error detection and correction for cross-layer approximations.
- Developing low-cost systems for online quality assessment and resource management. Such systems are mainly for applications that have high run-time variations and require modules for dynamically orchestrating the approximation knobs to achieve best efficiency while meeting the user-defined quality constraints.

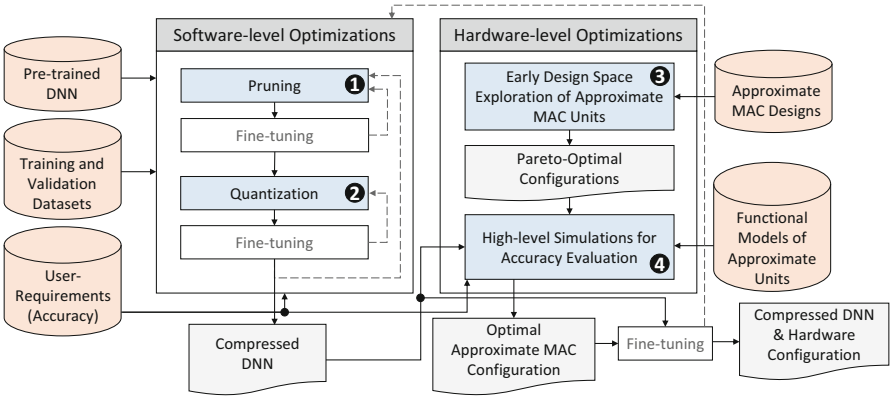


Fig. 14 A cross-layer optimization flow for DNNs (Hanif and Shafique 2021)

Cross-Layer Methodology for Optimizing DNNs

DNNs are widely being used in many applications due to their state-of-the-art performance (LeCun et al. 2015). Studies have shown that they are (to some extent) resilient to errors in intermediate computations. This property of DNNs can be exploited through different types of approximations to reduce their execution cost and enable their deployment on resource-constrained devices. Toward this, various software-level and hardware-level approximation/optimization techniques have been proposed. At the software level, pruning and quantization are employed to reduce the complexity of the network and computations (respectively), and at the hardware level, customized hardware accelerators and approximate arithmetic modules are employed (as also shown in section "Deep Neural Networks (DNNs)"). These techniques can be combined in a systematic manner to achieve high efficiency gains. Figure 14 presents a cross-layer methodology that combines pruning and quantization techniques with hardware-level optimizations (Hanif and Shafique 2021). The methodology consists of the following steps:

- **Pruning:** At the software level, the most effective technique for optimizing DNNs is pruning. It involves removing the ineffectual weights from the network to reduce the complexity of DNNs. Based on its effectiveness, the cross-layer methodology employs pruning as Step 1. An iterative pruning technique is mainly employed that reduces the number of parameters in multiple iterations, where each iteration is (optionally) followed by partial retraining to compensate for the accuracy loss. The weights to be removed are selected based on their saliency, which can be estimated using L1-norm/L2-norm or by using a complex back-propagation algorithm. The number of weights removed in each iteration and the amount of retraining after each iteration are two key hyper-parameters that can impact the compression and/or accuracy of the resultant network and, therefore, have to be selected carefully. The iterations are performed till the accuracy of the network drops below the user-defined accuracy constraint, and

the network from the second to the last iteration is forwarded to the next step for further optimization.

- **Quantization:** The precision of DNN data structures impacts the memory requirements and the complexity of the computational modules. Quantization is employed to represent weights and activations using low-precision fixed-point format. It not only reduces the memory requirements for the inference stage but also helps in simplifying the hardware modules, e.g., MAC units. Therefore, the methodology employs quantization in Step 2 to further compress the network and simplify the logic units at the hardware level. The quantization process can be coupled with retraining to compensate for the accuracy loss due to quantization errors in the computations. Moreover, pruning and quantization can also be combined in a single unified process (Tung and Mori 2018). However, such methods require sophisticated optimization algorithms to efficiently explore the combined design space and propose an effective solution.
- **Hardware Approximations:** Specialized hardware accelerators are used for energy-efficient processing of data in real-world systems. These accelerators can be equipped with approximate units to further boost the efficiency gains. Toward this, Step 3 of the methodology explores the potential of hardware-level approximations, e.g., functional approximation of adders and multipliers. This step performs design space exploration of approximate modules to find the most suitable configurations that offer high efficiency while meeting the user-defined quality constraints. The step also explores the potential of internal self-healing modules, as they can offer better error characteristics in case of vector operations. These approximations can also be coupled with retraining to partially compensate for the accuracy loss due to approximations.

Case Studies for Improving the Energy and Performance Efficiency of DNN Inference

Structured Pruning

This section highlights the effectiveness of the pruning step (i.e., Step 1 in Fig. 14) for improving the efficiency of DNN inference. Figure 15 presents the flow considered in this study for pruning filters/neurons from a pre-trained DNN. The main steps of the flow are:

1. Given a pre-trained DNN, first, the methodology computes the saliency of each filter/neuron of the network using a suitable saliency measure, e.g., L1-norm.
2. Then for each layer of the DNN, it creates a copy of the network and removes $x\%$ of the least significant filters/neurons from the layer while keeping all rest of the layers intact.
3. The methodology then computes the accuracy and compression ratio of each model and registers them in θ. Note that for fast execution of the methodology, only a subset of the validation dataset is used to estimate the accuracy.

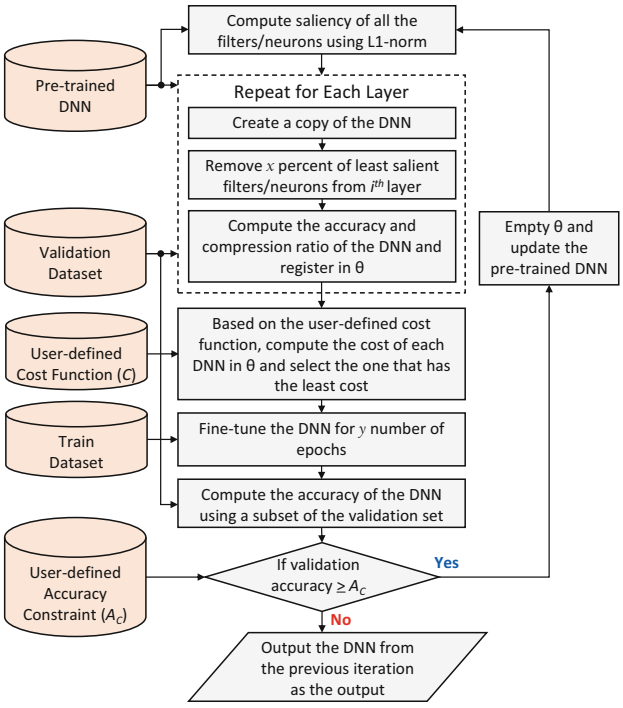


Fig. 15 The considered structured pruning methodology (Hanif and Shafique 2021)

4. A user-defined cost function C is then used to compute the cost of pruning in each individual layer.
5. The models in θ are then sorted based on their costs, and the one that has the least cost is selected, and all rest of the models are discarded.
6. The selected model is then fine-tuned for y number of epochs, and its accuracy is estimated using a subset of the validation dataset.
7. The accuracy is then compared with the user-defined accuracy constraint (A_c). If the accuracy is greater than the user-defined constraint, the pre-trained model is replaced with the pruned model, and the complete process is repeated until the accuracy falls below A_c. Once the accuracy is below A_c, the output of the previous iteration is passed as the final output of the methodology.

To show the effectiveness of pruning, the above flow is employed to prune filters/neurons from the LeNet5 and the VGG11 networks, both trained on the Cifar10 dataset. For these experiments, $C = 100 - (Accuracy + 4 * P_i / \sum_{j \in \{all\ layers\}} P_j)$ is used as the cost function, where $Accuracy$ is the estimated accuracy after pruning the ith layer and P_i is the number of parameters in the ith layer. For pruning, x is defined equal to 20, and for fine-tuning during the process, y is defined equal to 2. The results are presented in Figs. 16a and 17a. It can be seen from the figures that the methodology helps maintain the accuracy close to its baseline till a significant

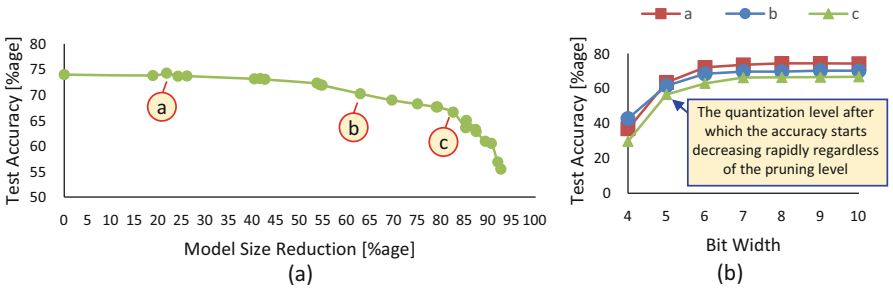


Fig. 16 Results of structured pruning when applied to the LeNet5 network trained on the Cifar10 dataset (Hanif and Shafique 2021). (**a**) Impact of structured pruning on accuracy. (**b**) Impact of quantization on the accuracy of the models having different compression ratios. The models are marked in (**a**)

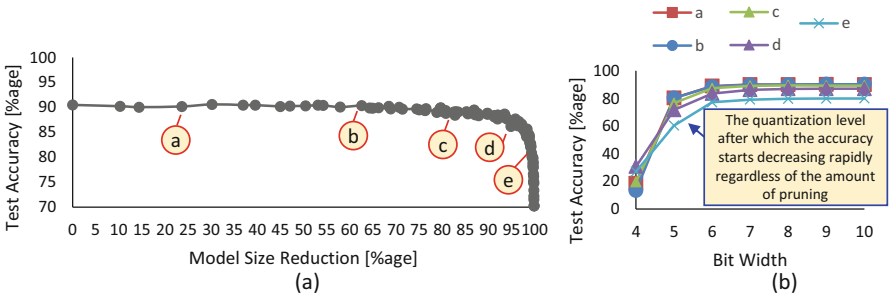


Fig. 17 Results of structured pruning when applied to the VGG11 network trained on the Cifar10 dataset (Hanif and Shafique 2021). (**a**) Impact of structured pruning on accuracy. (**b**) Impact of quantization on the accuracy of the models having different compression ratios. The models are marked in (**a**)

amount of compression, and, after a point, any further compression results in a rapid decrease in the accuracy. Note that intermediate fine-tuning, i.e., $y > 0$, is the key factor for achieving a high compression ratio.

Quantization

To further compress the DNN and to simplify the arithmetic modules in hardware accelerators, network quantization (i.e., Step 2 in Fig. 14) is applied after pruning. For this study, post-training quantization approach with uniform bit-width across the network is considered, for both weights and activations. To quantize the weights of a layer, the following equations are employed:

$$\hat{W_i^{<l>}} = round(W_i^{<l>} \times W_{scale}^{<l>}) \tag{8}$$

$$W_{scale}^{<l>} = 2^{floor(log_2(\frac{2^{n-1}-1}{max(abs(W^{<l>}))}))}$$

where $W^{<l>}$ is the set of all the weights, $W_i^{<l>}$ is the ith element in $W^{<l>}$, $\hat{W^{<l>}}$ represents the set of quantized weights, $W_{scale}^{<l>}$ is the scale factor, and n is the bit-width.

To quantize the activations, first, the activations are profiled using a set of input samples, and then the scale factor is defined using the following equation:

$$A_{scale}^{<l>} = 2^{floor\left(log_2\left(\frac{2^{n-1}-1}{max(abs(A^{<l>}))}\right)\right)}$$

Here $A^{<l>}$ is the set of all the logged activations from the input of the lth layer, and $A_{scale}^{<l>}$ is the scale factor. During the run-time, the activations are scaled with the help of following equation:

$$\hat{A_i^{<l>}} = round(A_i^{<l>} \times A_{scale}^{<l>}) \tag{9}$$

where $\hat{A^{<l>}}$ represents the quantized activations. Note that $W_{scale}^{<l>}$ and $A_{scale}^{<l>}$ are intentionally defined to be in the power of two to simplify the intermediate conversion operations.

Figure 16b shows the accuracies of five DNNs when exposed to different levels of quantization. All the DNNs are variants of the same LeNet5 model trained on the Cifar10 dataset but have different pruning ratios. The baseline models are marked in Fig. 16a with the help of labels. From the figure, it can be observed that the networks with high compression ratios are more sensitive to quantization. Moreover, the accuracy of the networks drops sharply after a specific quantization level. The same trend is observed for the VGG11 network trained on the Cifar10 dataset (see Fig. 17). From this analysis, it can be concluded that higher pruning levels are usually more beneficial than post-training quantization for achieving high overall compression while maintaining close to the baseline accuracy.

Hardware-Level Approximations: Impact of Self-Healing and Nonself-Healing Designs on DNN Accuracy

This section analyzes the impact of using approximate arithmetic modules for internal dot product operations of DNNs on their accuracy. This corresponds to Step 4 in Fig. 14. For this analysis, modules designed using conventional as well as self-healing methods are employed. The key distinction between these designs can be observed from Fig. 18. Figure 18a illustrates a system where the computational modules are replaced with their approximate variants without considering the overall computational flow. In such designs, the selection can be based on thorough design space exploration, but the system is not designed in a manner that the approximation error of one module is compensated by the error of the other modules. The self-healing designs exploit the fact that most of the real-world

systems involve accumulation of multiple computations. The accumulation stage is viewed as the healing stage, while the computational modules are approximated such that they generate complementary errors (Gillani et al. 2018, 2019). This way the error generated by one module is compensated by the error in other modules, and the overall application-level accuracy is not affected. The key advantage of self-healing is that it allows to apply more aggressive approximations in the system compared to the conventional methodology. Figure 18b and c show two different methods for introducing self-healing-based approximations in a system.

The dot product operation is the most common operation involved in DNN execution. It comprises multiplications followed by the accumulation of the products. As multiplication is one of the most costly operations, in this work, approximations are deployed in the multipliers in hardware accelerators. Moreover, conventional as well as self-healing approximate multipliers are considered to study the effectiveness of functional approximations in arithmetic circuits. Figure 19a shows the baseline 8×8 multiplier design used in this work, constructed using 2×2 multipliers. The design of the accurate 2×2 multiplier is shown in Fig. 20a. For approximations, the designs shown in Fig. 20b–20d are employed, where the designs in Fig. 20b and d approximate 3×3 to 7 and 5, respectively (i.e., negative error), and the design in Fig. 20c approximates 3×3 to 11. The 8×8 multiplier configurations used in the analysis are illustrated in Fig. 19b–19j, and their error characteristics are presented in Table 5. Note, for this analysis, it is assumed that the same multiplier design is used for all the multipliers in the hardware accelerator, i.e., homogeneous design. The approximate multiplier configurations that are composed of modules that generate only negative errors represent the conventional multipliers

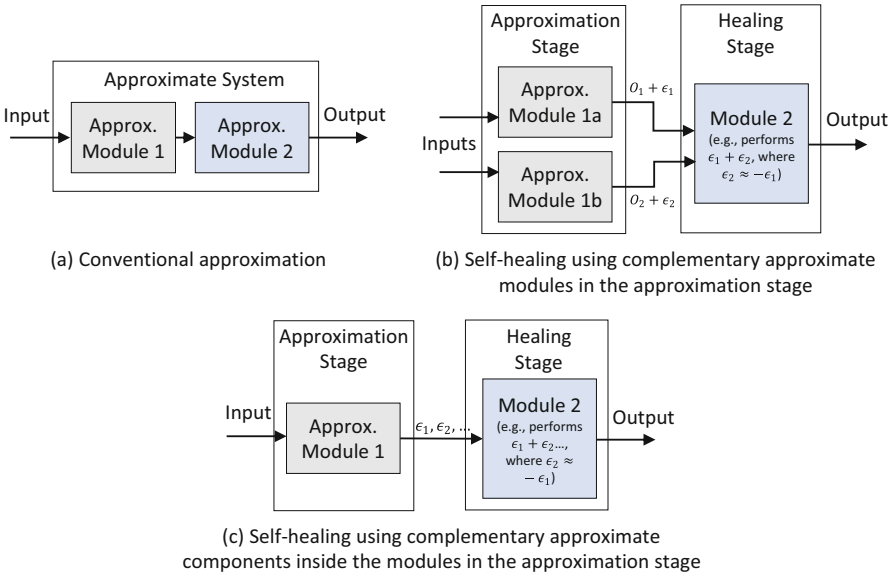


Fig. 18 A comparison of conventional and self-healing approaches (Hanif and Shafique 2021)

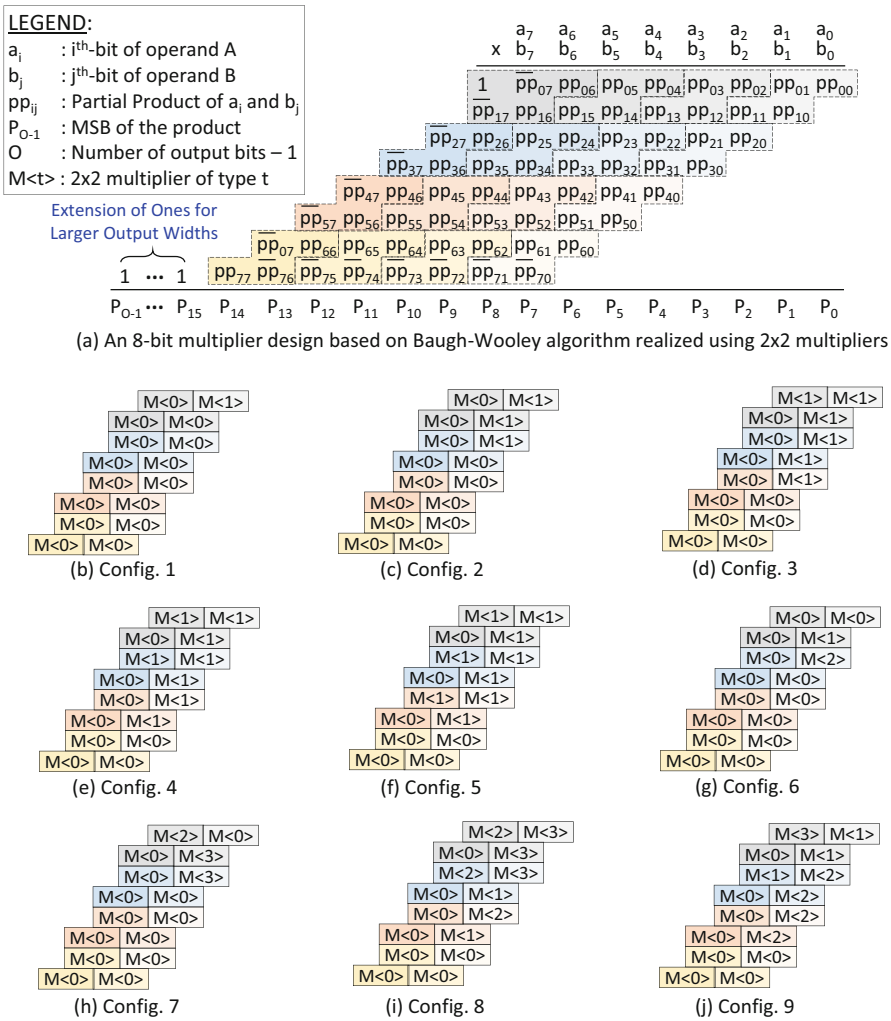
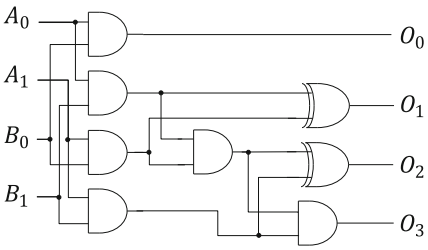


Fig. 19 Types of 8 × 8 approximate multipliers considered for simulations (Hanif and Shafique 2021). (**a**) An 8-bit multiplier design based on Baugh-Wooley algorithm realized using 2 × 2 multipliers. (**b**) Config. 1. (**c**) Config. 2. (**d**) Config. 3. (**e**) Config. 4. (**f**) Config. 5. (**g**) Config. 6. (**h**) Config. 7. (**i**) Config. 8. (**j**) Config. 9

(i.e., configurations in Fig. 19b–19f), and the configurations that generate both positive and negative errors represent the self-healing designs (i.e., configurations in Fig. 19g–19j). The hardware characteristics of all the configurations are presented in Table 6. The results are generated for 65 nm technology using Cadence Genus Synthesis tool with TSMC 65 nm library.

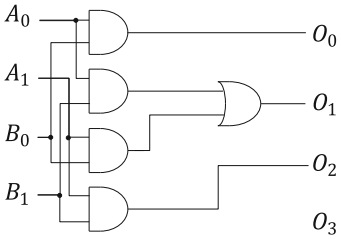
To evaluate the impact of approximations on the accuracy of DNNs, functional models of these approximate multipliers are integrated in a PyTorch-based simulation framework. Figure 21 shows the results obtained when different approximate



(a) Accurate 2x2 multiplier: M<0>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001

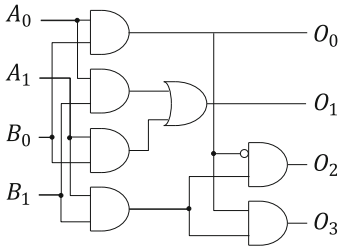
(e) Truth table of M<0>



(b) Approximate 2x2 multiplier having 3x3→7: M<1>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0111

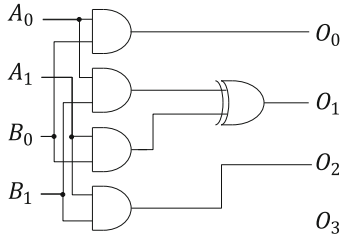
(f) Truth table of M<1>



(c) Approximate 2x2 multiplier having 3x3→11: M<2>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(g) Truth table of M<2>



(d) Approximate 2x2 multiplier having 3x3→5: M<3>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0101

(h) Truth table of M<3>

Fig. 20 The 2 × 2 multiplier designs used for building 8 × 8 approximate multipliers (Hanif and Shafique 2021). (**a**) Accurate 2 × 2 multiplier: M<0>. (**b**) Approximate 2 × 2 multiplier having 3 × 3 → 7: M<1>. (**c**) Approximate 2 × 2 multiplier having 3 × 3 → 11: M<2>. (**d**) Approximate 2 × 2 multiplier having 3 × 3 → 5: M<3>. (**e**) Truth table of M<0>. (**f**) Truth table of M<1>. (**g**) Truth table of M<2>. (**h**) Truth table of M<3>

Table 5 Error characteristics of the multiplier configurations presented in Fig. 19 (Hanif and Shafique 2021)

	Multiplier configurations								
	Ax. 1	Ax. 2	Ax. 3	Ax. 4	Ax. 5	Ax. 6	Ax. 7	Ax. 8	Ax. 9
MSE	0.25	9.75	266.25	3102.30	24806.00	7.50	78.00	2128.00	2547.00
MED	0.13	1.13	7.13	23.13	55.13	0.94	3.38	19.94	21.90
Mean error	-0.13	-1.13	-7.13	-23.13	-55.13	0.00	0.00	-0.25	-0.13

Table 6 Hardware characteristics of the multiplier configurations presented in Fig. 19 (Hanif and Shafique 2021)

	Multiplier configurations									
	Accurate	Ax. 1	Ax. 2	Ax. 3	Ax. 4	Ax. 5	Ax. 6	Ax. 7	Ax. 8	Ax. 9
Area [cell area]	753	716	696	616	609	571	726	727	672	670
Power [μW]	46.04	44.98	44.92	40.81	40.98	38.96	45.49	45.05	43.48	42.94
Delay [ns]	1.92	1.86	1.73	1.73	1.73	1.73	1.95	1.87	1.73	1.77
PDP [fJ]	88.40	83.66	77.71	70.60	70.90	67.40	88.71	84.24	75.22	76.00

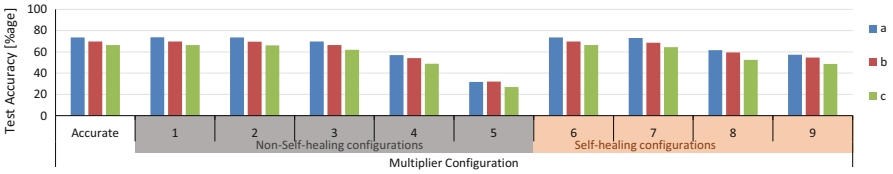


Fig. 21 Impact of using approximate multipliers on the accuracy of different pruned variants of the LeNet5 network (Hanif and Shafique 2021). The considered variants are marked in Fig. 16a

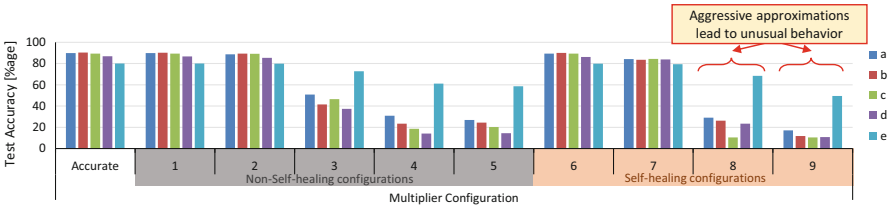


Fig. 22 Impact of using approximate multipliers on the accuracy of different pruned variants of the VGG11 network (Hanif and Shafique 2021). The considered variants are marked in Fig. 17a

multiplier configurations (shown in Fig. 19) are used for the LeNet5 network trained on the Cifar10 dataset. Note, for this analysis, multiple variants of the network are considered, each having experienced a different level of pruning. The network variants are highlighted in Fig. 16a. As can be seen in Fig. 21, with an increase in the compression ratio, the model becomes increasingly sensitive to approximations. Similar results are observed for the case of VGG11 network (see Fig. 22).

Conclusions

Approximations can offer high energy savings while meeting user-defined quality constraints. Besides the well-known techniques such as quantization (i.e., bit-width reduction) and code simplification (e.g., reducing the number of iterations of a loop), it is possible to approximate the functionality of circuits as well. The first part of the chapter primarily focused on functional approximations, where approaches for building approximate components such as adders and multipliers using both manual and automated methods were introduced.

The following section focused on the construction of complex hardware accelerators using existing libraries of approximate components (such as *EvoApproxLib*, *lpAcLib*, or *GeAR*). Two different types of accelerators were presented. For accelerators with irregular structures such as image processing accelerators, an automatic design space exploration and circuit approximation methodology *AutoAx* was presented. This methodology replaces operations in an original accelerator with approximate variants taken from a library of approximate components/circuits. To accelerate the approximation process, QoR and hardware parameters are estimated using computational models created using machine learning methods. It was shown that AutoAx methodology generates approximate accelerators that offer high-quality trade-offs between QoR and hardware parameters. The trade-offs are better than the SoA approaches based on selecting components with the same error or random selection.

The authors also focused on accelerators with a regular structure of processing elements. The methodology *ALWANN* that allows us to approximate hardware accelerators of convolutional neural networks and optimize their energy consumption for inference was introduced. Better energy savings with the same accuracy than the other algorithms that employ retraining were achieved. The retraining typically results in (i) approximation of significantly smaller networks due to scalability issues (Mrazek et al. 2016; Zhang et al. 2015) or (ii) limited set of considered approximate components (Sarwar et al. 2018).

Functional approximation is not the only approach to trade quality for energy efficiency. Developers may also use other techniques such as quantization and pruning. Toward this, a *cross-layer* optimization for neural networks was presented, which systematically combines software-level and hardware-level approximation techniques. The results showed that cross-layer optimization results in better quality-efficiency trade-off. However, note that cross-layer approximate computing is still an active area of research that is yet to uncover the ultimate potential of approximate computing. One of the key hurdles toward achieving that is the lack of sophisticated methodologies for evaluating the error masking and propagation characteristics of approximations, which will enable the projection of approximations across layers and therefore enable fast design space exploration. From approximations for DNNs' perspective, as most of the approximate components have irregular error distribution, there is a need for methodologies to adapt (retrain) DNNs for such approximations. Apart from that, there is a dire need to explore and determine the security of approximated DNNs against adversarial attacks.

Acknowledgments This work was partially supported by the Czech science foundation project 21-13001S.

References

Bailey B, Martin G, Piziali A, Burton M, Greenbaum J, Hashmi K, Haverinen A, Lavagno L, Meredith M, Murray B et al (2007) ESL design and verification: a prescription for electronic system level methodology. Elsevier Science. https://books.google.cz/books?id=raoeAQAAIAAJ

Češka M, Matyáš J, Mrazek V, Sekanina L, Vasicek Z, Vojnar T (2017) Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In: 2017 IEEE/ACM international conference on computer-aided design (ICCAD), pp 416–423

Chandrasekharan A, Soeken M, Große D, Drechsler R (2016) Approximation-aware rewriting of AIGs for error tolerant applications. In: Proceedings of the 35th international conference on computer-aided design, ICCAD'16. ACM, New York, pp 83:1–83:8

Chan WTJ, Kahng AB, Kang S, Kumar R, Sartori J (2013) Statistical analysis and modeling for error composition in approximate computation circuits. In: 2013 IEEE 31st international conference on computer design (ICCD), pp 47–53

Chang IJ, Mohapatra D, Roy K (2011) A priority-based 6t/8t hybrid sram architecture for aggressive voltage scaling in video applications. IEEE Trans Circuits Syst Video Technol 21(2):101–112

Chen TH, Alaghi A, Hayes JP (2014) Behavior of stochastic circuits under severe error conditions. it – Inf Technol 56(4):182–191

Chippa VK, Chakradhar ST, Roy K, Raghunathan A (2013) Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–9

Choi J, Venkataramani S (2019) Approximate computing techniques for deep neural networks. Springer, Cham, pp 307–329

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197

Du K, Varman P, Mohanram K (2012) High performance reliable variable latency carry select addition. In: Proceedings of the conference on design, automation and test in Europe, DATE'12. EDA Consortium, San Jose, pp 1257–1262

Esmaeilzadeh H, Sampson A, Ceze L, Burger D (2012) Architecture support for disciplined approximate programming. In: ACM SIGPLAN notices, vol 47. ACM, pp 301–312

Gillani GA, Hanif MA, Krone M, Gerez SH, Shafique M, Kokkeler AB (2018) Squash: approximate square-accumulate with self-healing. IEEE Access 6:49112–49128

Gillani G, Hanif MA, Verstoep B, Gerez SH, Shafique M, Kokkeler AB (2019) Macish: designing approximate MAC accelerators with internal-self-healing. IEEE Access 7:77142–77160

Gupta V, Mohapatra D, Park SP, Raghunathan A, Roy K (2011) IMPACT: imprecise adders for low-power approximate computing. In: Proceedings of 17th IEEE/ACM international symposium on low-power electronics and Design, pp 409–414

Hanif MA, Shafique M (2021) A cross-layer approach towards developing efficient embedded deep learning systems. In: Microprocessors and microsystems p 103609

Hanif MA, Hafiz R, Hasan O, Shafique M (2017) Quad: design and analysis of quality-area optimal low-latency approximate adders. In: DAC design automation conference 2017. ACM, New York, pp 42:1–42:6

Hashemi S, Tann H, Reda S (2018) BLASYS: approximate logic synthesis using boolean matrix factorization. In: Proceedings of the 55th annual design automation conference, DAC 2018, San Francisco, 24–29 June 2018. ACM, pp 55:1–55:6. https://doi.org/10.1145/3195970.3196001

He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. CoRR **abs/**1512.03385

Hrbacek R, Sekanina L (2014) Towards highly optimized cartesian genetic programming: from sequential via simd and thread to massive parallel implementation. In: Proceedings of the 2014 annual conference on genetic and evolutionary computation, GECCO'14. ACM, New York, pp 1015–1022

Jiang H, Liu C, Liu L, Lombardi F, Han J (2017) A review, classification, and comparative evaluation of approximate arithmetic circuits. J Emerg Technol Comput Syst 13(4):60:1–60:34

Judd P, Albericio J, Hetherington T, Aamodt T, Enright Jerger N, Urtasun R, Moshovos A (2018) Proteus: exploiting precision variability in deep neural networks. Parallel Comput 73:40–51

Kulkarni P, Gupta P, Ercegovac M (2011) Trading accuracy for power with an underdesigned multiplier architecture. In: 2011 24th internatioal conference on VLSI design, pp 346–351

Kyaw KY, Goh WL, Yeo KS (2010) Low-power high-speed multiplier for error-tolerant application. In: 2010 IEEE international conference of electron devices and solid-state circuits (EDSSC), pp 1–4

LeCun Y, Bengio Y, Hinton G (2015) Deep Learning. Nature 521(7553):436–444

Li C, Luo W, Sapatnekar SS, Hu J (2015) Joint precision optimization and high level synthesis for approximate computing. In: Proceedings of the 52nd annual design automation conference, DAC'15. ACM, New York, pp 104:1–104:6

Lotfi A, Rahimi A, Yazdanbakhsh A, Esmaeilzadeh H, Gupta RK (2016) Grater: an approximation workflow for exploiting data-level parallelism in FPGA acceleration. In: 2016 design, automation test in Europe conference exhibition (DATE), pp 1279–1284

Lu SL (2004) Speeding up processing with approximation circuits. Computer 37(3):67–73

Ma J, Hashemi S, Reda S (2019) Approximate logic synthesis using blasys. In: Proceedings of 1st workshop on open-source EDA technology (WOSET), p 3

Mahdiani HR, Ahmadi A, Fakhraie SM, Lucas C (2010) Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. IEEE Trans Circuits Syst I: Regul Pap 57(4):850–862

Mazahir S, Hasan O, Hafiz R, Shafique M (2017a) Probabilistic error analysis of approximate recursive multipliers. IEEE Trans Comput 66(11):1982–1990

Mazahir S, Hasan O, Hafiz R, Shafique M, Henkel J (2017b) Probabilistic error modeling for approximate adders. IEEE Trans Comput 66(3):515–530

Mishchenko A, Chatterjee S, Brayton R (2006) Dag-aware aig rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE design automation conference, pp 532–535

Mishra AK, Barik R, Paul S (2014) iact: a software-hardware framework for understanding the scope of approximate computing. In: Workshop on approximate computing across the system stack (WACAS)

Mohapatra D, Chippa VK, Raghunathan A, Roy K (2011) Design of voltage-scalable meta-functions for approximate computing. In: Design, automation & test in Europe conference & exhibition (DATE), 2011. IEEE, pp 1–6

Momeni A, Han J, Montuschi P, Lombardi F (2015) Design and analysis of approximate compressors for multiplication. IEEE Trans Comput 64(4):984–994

Mrazek V, Sarwar SS, Sekanina L, Vasicek Z, Roy K (2016) Design of power-efficient approximate multipliers for approximate artificial neural networks. In: Proceedings of the 35th international conference on computer-aided design, ICCAD'16. ACM, New York, pp 81:1–81:7

Mrazek V, Hrbacek R, Vasicek Z, Sekanina L (2017) Evoapprox8b: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Design, automation test in Europe conference exhibition (DATE), 2017, pp 258–261

Mrazek V, Vasicek Z, Hrbacek R (2018) The role of circuit representation in evolutionary design of energy-efficient approximate circuits. IET Comput Digit Tech 12(4):139–149

Mrazek V, Hanif MA, Vasicek Z, Sekanina L, Shafique M (2019a) AutoAx: an automatic design space exploration and circuit building methodology utilizing libraries of approximate components. In: Proceedings of the 56th annual design automation conference 2019. Association for Computing Machinery, New York

Mrazek V, Vasicek Z, Sekanina L, Hanif MA, Shafique M (2019b) ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining. In: 2019 IEEE/ACM international conference on computer-aided design (ICCAD), pp 1–8

Nair R (2014) Big data needs approximate computing: technical perspective. Commun ACM 58(1):104–104

Nepal K, Li Y, Bahar RI, Reda S (2014) Abacus: a technique for automated behavioral synthesis of approximate computing circuits. In: 2014 design, automation test in Europe conference exhibition (DATE), pp 1–6

Nepal K, Hashemi S, Tann H, Bahar RI, Reda S (2017) Automated high-level generation of low-power approximate computing circuits. In: IEEE transactions on emerging topics in computing, p 1

Pinos M, Mrazek V, Sekanina L (2021) Evolutionary neural architecture search supporting approximate multipliers. In: Hu T, Lourenço N, Medvet E (eds) Genetic programming. Springer International Publishing, Cham, pp 82–97

Ranjan A, Raha A, Venkataramani S, Roy K, Raghunathan A (2014) Aslan: synthesis of approximate sequential circuits. In: 2014 design, automation test in Europe conference exhibition (DATE), pp 1–6

Sarwar SS, Venkataramani S et al (2018) Energy-efficient neural computing with approximate multipliers. J Emerg Technol Comput Syst 14(2):16:1–16:23

Sengupta D, Snigdha FS et al (2017) Saber: selection of approximate bits for the design of error tolerant circuits. In: Design automation conference (DAC)

Shafique M, Ahmad W, Hafiz R, Henkel J (2015) A low latency generic accuracy configurable adder. In: Proceedings of annual design automation conference, DAC'15, pp 86:1–86:6

Shafique M, Hafiz R, Rehman S, El-Harouni W, Henkel J (2016) Cross-layer approximate computing: from logic to architectures. In: Proceedings of 53rd IEEE/ACM design automation conference

Sidiroglou-Douskos S, Misailovic S, Hoffmann H, Rinard M (2011) Managing performance vs. accuracy trade-offs with loop perforation. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM, pp 124–134

Soeken M, Große D, Chandrasekharan A, Drechsler R (2016) BDD minimization for approximate computing. In: Proceedings of the 21st Asia and South Pacific design automation conference (ASP-DAC), pp 474–479

Srinivasan G, Wijesinghe P, Sarwar SS, Jaiswal A, Roy K (2016) Significance driven hybrid 8t-6t sram for energy-efficient synaptic storage in artificial neural networks. In: 2016 design, automation test in Europe conference exhibition (DATE), pp 151–156

Tung F, Mori G (2018) CLIP-Q: deep network compression learning by in-parallel pruning-quantization. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 7873–7882

Vasicek Z, Sekanina L (2014) How to evolve complex combinational circuits from scratch? In: 2014 IEEE international conference on evolvable systems, pp 133–140

Vasicek Z, Sekanina L (2015) Evolutionary approach to approximate digital circuits design. IEEE Trans Evol Comput 19(3):432–444

Vasicek Z, Mrazek V, Sekanina L (2017) Towards low power approximate DCT architecture for HEVC standard. In: Design, automation test in Europe conference exhibition (DATE), 2017, pp 1576–1581

Vasicek Z, Mrazek V, Sekanina L (2019) Automated circuit approximation method driven by data distribution. In: 2019 design, automation test in Europe conference exhibition (DATE), pp 96–101

Vaverka F, Mrazek V, Vasicek Z, Sekanina L, Hanif MA, Shafique M (2020) Tfapprox: towards a fast emulation of dnn approximate hardware accelerators on GPU. In: 2020 design, automation and test in Europe conference (DATE), p 4

Venkataramani S, Sabne A, Kozhikkottu V, Roy K, Raghunathan A (2012) Salsa: systematic logic synthesis of approximate circuits. In: DAC design automation conference 2012, pp 796–801

Venkataramani S, Roy K, Raghunathan A (2013) Substitute-and-simplify: a unified design paradigm for approximate and quality configurable circuits. In: 2013 design, automation test in Europe conference exhibition (DATE), pp 1367–1372

Venkataramani S, Ranjan A, Roy K, Raghunathan A (2014) Axnn: energy-efficient neuromorphic systems using approximate computing. In: 2014 IEEE/ACM international symposium on low power electronics and design (ISLPED), pp 27–32

Xu Q, Kim NS, Mytkowicz T (2016) Approximate computing: a survey. IEEE Des Test 33(1): 8–22

Zhang Q, Wang T, Tian Y, Yuan F, Xu Q (2015) Approxann: an approximate computing framework for artificial neural network. In: Design, automation test in Europe conference exhibition (DATE), pp 701–706

Zhu N, Goh WL, Yeo KS (2009) An enhanced low-power high-speed adder for error-tolerant application. In: Proceedings of the 2009 12th international symposium on integrated circuits, pp 69–72