Using Libraries of Approximate Circuits in Design of Hardware Accelerators of Deep Neural Networks

Vojtech Mrazek, Lukas Sekanina, Zdenek Vasicek
Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno University of Technology, Brno, Czech Republic
{mrazek,sekanina,vasicek}@fit.vutbr.cz

***Abstract*—Approximate circuits have been developed to provide good tradeoffs between power consumption and quality of service in error resilient applications such as hardware accelerators of deep neural networks (DNN). In order to accelerate the approximate circuit design process and to support a fair benchmarking of circuit approximation methods, libraries of approximate circuits have been introduced. For example, EvoApprox8b contains hundreds of 8-bit approximate adders and multipliers. By means of genetic programming we generated an extended version of the library in which thousands of 8- to 128-bit approximate arithmetic circuits are included. These circuits form Pareto fronts with respect to several error metrics, power consumption and other circuit parameters. In our case study we show how a large set of approximate multipliers can be used to perform a resilience analysis of a hardware accelerator of ResNet DNN and to select the most suitable approximate multiplier for a given application. Results are reported for various instances of the ResNet DNN trained on CIFAR-10 benchmark problem.**

***Index Terms*—Approximate circuit, genetic programming, convolutional neural network, hardware accelerator**

I. Introduction

Many computationally intensive applications (such as image recognition, video processing and data mining) feature an intrinsic *error-resilience* property [1]. Since they often process noisy or redundant data and their users are willing to accept certain errors in many cases, the principles of *approximate computing* can be employed in the design of their energy-efficient implementations [2]. At the circuit level, approximations (i.e. circuit simplifications) are intentionally introduced to find a good trade-off between power consumption, performance and error. A distinguished class of applications among all these error resilient applications are hardware accelerators of deep neural networks (DNNs) [3]. In the case of DNNs, approximate implementations have been proposed at the level of DNN architecture, data representation, arithmetic operations, memory access and memory cells [3]–[5].

The approximations can be introduced to the circuit in various steps of the standard circuit design flow. In this work, we primarily focus on the technology independent logic synthesis step. The approximations introduced in this step, the so-called *functional approximations*, modify the Boolean function of the circuit. It has one important advantage —– the approximate circuit can be implemented in arbitrary ASIC as well as FPGA technology because it is assumed that the technology dependent implementation is performed by means of common open source or commercial tools after the approximation is conducted.

The methods introduced for the functional approximations can be divided into two categories: (1) manual, and (2) automated. The manual (ad-hoc) methods have been developed for specific circuit components such as adders and multipliers [6], [7]. On the other hand, the automated methods use some general-purpose circuit simplification, resynthesis and approximation techniques and enable us to approximate arbitrary circuits. These methods start with an original (exact) circuit and, typically iteratively, modify its structure.

However, the functional approximation of complex circuits is a time-consuming process. As many of these circuits contain common arithmetic components (circuits) such as adders and multipliers, they can be approximated by replacing selected components by their approximate implementations available in a suitable library. A comprehensive library of approximate arithmetic circuits was introduced in 2017, see EvoApprox8b in [8]. These circuits were designed by means an automated approximation algorithm based on genetic programming, which will be described in Section II. EvoApprox8b contains hundreds of 8-bit approximate adders and multipliers.

The goal of this paper is to extend this library to contain more approximate implementations of arithmetic circuits, with the focus on hardware accelerators of DNNs. By means of genetic programming we generated an extended version of the library in which thousands of 8- to 128-bit approximate arithmetic circuits are included. These circuits form Pareto fronts with respect to several error metrics, power consumption and other circuit parameters. In our case study we show how a large set of approximate multipliers can be used to perform a resilience analysis of a hardware accelerator of ResNet DNN and to select the most suitable approximate multiplier for a given application. Results are reported for various instances of the ResNet DNN trained on CIFAR-10 benchmark problem [9].

II. Automated construction of approximate arithmetic circuits

The method used to obtain the library follows the methodology introduced in [10]. It is a general-purpose approximation method for combinational circuits based on Cartesian Genetic Programming (CGP). CGP represents candidate circuits as directed acyclic graphs and iteratively modifies these circuits

978-1-7281-4922-620$31.00

to reach the design objectives while ensuring that various constraints (e.g. the error is below a given threshold) are not violated.

A. Errors of approximate circuits

Selection of the error metrics is the key step of the whole design. The quality of approximate combinational circuits is typically expressed using one or several error metrics, where the most commonly used ones are: the error rate (ER), the mean absolute error (MAE), the mean square error (MSE), the mean relative error (MRE), the worst case error (WCE), the worst case relative error (WCRE), see eq. 1 – 6 in which the output of the approximate circuit and original (exact) circuit is O_{approx} and O_{orig}, n_i is the number of primary inputs, the operand's width is $n_i/2$ bits and $\forall i$ enumerates all possible input vectors.

$$\text{ER} = \frac{\sum_{\forall i: O^{(i)}_{\text{approx}} \neq O^{(i)}_{\text{orig}}} 1}{2^{n_\text{i}}} \tag{1}$$

$$\text{MAE} = \frac{\sum_{\forall i} \left| O^{(i)}_{\text{approx}} - O^{(i)}_{\text{orig}} \right|}{2^{n_\text{i}}} \tag{2}$$

$$\text{MSE} = \frac{\sum_{\forall i} \left| O^{(i)}_{\text{approx}} - O^{(i)}_{\text{orig}} \right|^2}{2^{n_\text{i}}} \tag{3}$$

$$\text{MRE} = \frac{\sum_{\forall i} \frac{\left| O^{(i)}_{\text{approx}} - O^{(i)}_{\text{orig}} \right|}{\max(1, O^{(i)}_{\text{orig}})}}{2^{n_\text{i}}} \tag{4}$$

$$\text{WCE} = \max_{\forall i} \left| O^{(i)}_{\text{approx}} - O^{(i)}_{\text{orig}} \right| \tag{5}$$

$$\text{WCRE} = \max_{\forall i} \frac{\left| O^{(i)}_{\text{approx}} - O^{(i)}_{\text{orig}} \right|}{\max(1, O^{(i)}_{\text{orig}})} \tag{6}$$

B. Cartesian genetic programming

CGP particularly differs from other genetic programming branches in (1) the solution representation and (2) the search mechanism [11].

1) Representation: A candidate circuit is represented as an integer netlist (the so-called chromosome) describing a constant number of nodes (N). These nodes are organized in a two-dimensional grid of n_c columns and n_r rows ($N = n_c \cdot n_r$). The number of primary inputs and outputs of the circuit is denoted n_i and n_o. Each node implements one of the functions specified in the set of functions Γ and has up to n_a inputs and a single output. For gate-level circuits, Γ usually contains a set of binary logic functions ($n_a = 2$). Fig. 1 gives an example.

2) Search algorithm: Every candidate circuit represents one design point in the design space. In CGP, new designs are created by introducing small random modifications to the chromosome. This operation is called the mutation and it typically modifies h integers of the chromosome. Note that all modifications must lead to valid circuits, i.e. only valid function codes and connections can be created.

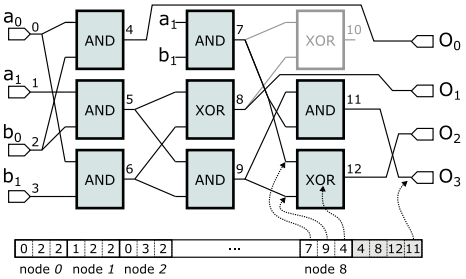

Fig. 1. A two-bit multiplier represented in CGP with parameters: $n_i = n_o = 4, n_c = n_r = 3, n_a = 2, \Gamma = \{0^{identity}, 1^{not}, 2^{and}, 3^{or}, 4^{xor}, 5^{nand}, 6^{nor}, 7^{xnor}, 8^{cont0}, 9^{const1}\}$.

The search method is based on the $(1 + \lambda)$ evolutionary strategy which is usually used for a single-objective circuit approximation [11]. The search algorithm can start with either a randomly generated initial population or existing designs. The population size is $1 + \lambda$. After evaluating the initial population (i.e. measuring the circuit functionality and cost) the following steps are repeated until the termination condition is not satisfied: (i) the best-scored circuit (called the parent) is selected; (ii) λ offspring circuits are created from the parent by means of mutation; (iii) the population is evaluated.

C. Circuit approximation using CGP

If a single-objective CGP is applied, the target error range (e.g. the worst-case error), determined by e_{min} and e_{max}, is specified by the user. The goal is to minimize the number of gates (or area or power consumption) while the error of the circuits is kept between the target values e_{min} and e_{max}. If various tradeoffs between the error and the number of gates are requested, CGP is executed several times with e_{max} as the control parameter.

The multi-objective CGP allows us to optimize the error and other key circuit parameters (area, delay and power consumption) together in one run. We are primarily interested in approximate circuits belonging to the *Pareto front* which contains the so-called *non-dominated solutions*. For example, consider two circuits C1 and C2. Circuit C1 *dominates* 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.

The design process typically starts with an accurate circuit. Candidate approximate circuits are generated from the original circuit using CGP. As millions of candidate circuits are often generated, the evaluation must be fast. For small circuits, an exhaustive circuit simulation utilizing all possible input vectors can be used. However, for large circuits this approach is not scalable. A possible solution is to employ advanced verification methods, based on e.g. Boolean satisfiability solving or binary decision diagram analysis [12], [13]. The methodology can easily handle arbitrary constraints.

III. Library of Approximate Arithmetic Circuits

Similarly to [8] we seeded CGP with conventional implementations of target arithmetic circuits. A typical single-objective CGP run uses the following parameters: $N = k$, where k is the number of gates of the original (exact) circuit with n_i primary inputs and n_o primary outputs, $\lambda = 1$, $h = 5$, Γ contains all 2-input gates and 1 million generations are produced. One CGP run is typically finished within the order of tens of minutes, depending on the circuit complexity. In the fitness function, the error is obtained by applying one of the error metrics (eq. 1 – 6) and the cost is estimated as the sum of weighted areas of the gates used in the circuit. At the end of evolution, the best-scored circuit is synthesized and its parameters are determined by a common design tool (we used Synopsys Design Compiler, 45 nm process, V_{dd} =1V).

The new version of the library contains thousands of various arithmetic circuits as shown in Table I. Since the enormous number of circuits makes the selection of the most suitable circuit for a given application difficult, we identified a subset of circuits and used them in our experiments. The selection follows the principles of Pareto optimality with respect to several objectives in which power consumption is compared against EP, MAE, WCE, MSE and MRE metrics. For each of the five subsets of components, ten circuits evenly distributed along the power axis are taken.

TABLE I
The Number of Approximate Implementations of Arithmetic Circuits in the Proposed Library

Circuit	Bit-width	# approx. implementations
adder	8	6,979
	9	332
	12	4,661
	16	1,437
	32	916
	64	176
	128	196
multiplier	8	29,911
	12	3,495
	16	35,406
	32	349

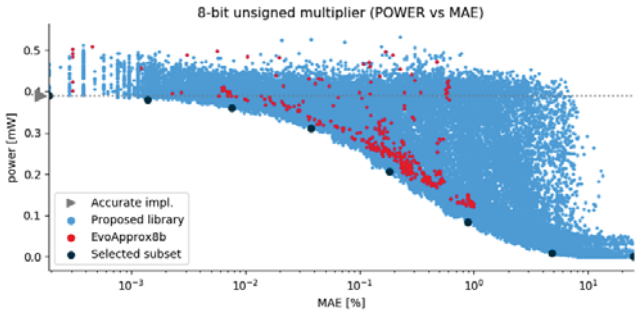

Fig. 2. Parameters of 8-bit approximate multipliers (black points) selected from all the approximate multipliers (blue points) and compared to the former version of EvoApprox8b library (red points).

Parameters of evolved approximate 8-bit multipliers (power vs. MAE) are shown in Fig. 2. The evolved circuits can be seen as the new state-of-the-art solutions as they provide better tradeoffs than the circuits of the original version of the EvoApprox8b library; the blue points (that occupy the Pareto front "power vs. MAE") are clearly better solutions than the red points representing the original circuits of EvoApprox8b. Note that EvoApprox8b was compared with the state of art approximate circuits in a greater detail [8], [10]. Selected approximate circuits and their various parameters can be downloaded from https://ehw.fit.vutbr.cz/evoapproxlib.

IV. Resilience Analysis in DNN Hardware Accelerators

In order to introduce suitable approximations to DNNs, a resilience analysis is conducted prior to any implementation steps. The resilience analysis of DNNs is usually performed by removing some neurons, weights, memory accesses or inserting some noise to neurons [4], [14]. Introducing approximate multipliers to convolutional layers is one of the most preferred approximation techniques [15]. The proposed library allows us to perform the resilience analysis (focused on these multipliers) in a more realistic way than previous methods. As many approximate multipliers are available, we can immediately analyze the impact of their utilization not only on the accuracy of classification, but also on the power consumption reduction.

In our case study, we investigate the impact of approximations introduced to the multipliers used in convolutional layers of ResNet [16] networks (Fig. 3) trained to classify CIFAR-10 using TensorFlow [17]. The smallest ResNet-8 network consists of three stages with $n = 1$ residual block in each stage. It contains 7 convolutional layers and performs 21 millions multiplications in the inference phase. The classification accuracy drops from 83.42% to 82.85% if a common floating point multiplication is replaced with the 8-bit (exact) multiplication. This 8-bit multiplier is considered as a golden solution and all proposed approximations will be compared against it. Note that no retraining is performed after introducing an approximate multiplier.

From the set of 29,911 approximate 8-bit multipliers available in the library, we identified a subset for our experiments in the following way. In order to obtain diverse designs, we selected 10 Pareto optimal multipliers with respect to power and MAE and repeated this selection for other four error metrics. After removing duplicate circuits we ended up with 35 approximate multipliers showing high-quality tradeoffs between power and the five error metrics.

All exact multipliers of a given layer of ResNet-8 were then replaced by one of the approximate multipliers. This process has been repeated for all the layers and all the approximate multipliers, but only one layer was modified and one type of approximate multipliers was used in each experiment. Tensor Flow extension TFApprox [18] enabled us to accelerate the simulation of ResNet networks containing approximate multipliers. Fig. 4 shows that the most interesting approximations

TABLE II

PARAMETERS OF SELECTED APPROXIMATE MULTIPLIERS EXPRESSED WITH RESPECT TO THE EXACT 8-BIT MULTIPLIER AND THE CLASSIFICATION ACCURACY (ON CIFAR-10) OF VARIOUS RESNET NETWORKS UTILIZING THESE CIRCUITS. MUL8U ARE EVOLVED MULTIPLIERS AND BAM MULTIPLIERS (h AND v ARE THE HORIZONTAL AND VERTICAL BREAK LEVELS) ARE CONSTRUCTED ACCORDING TO [7]

Multiplier	Relative Power [%]	Arithmetic errors					Classification accuracy [%]							
		MAE [%]	*WCE [%]*	*MRE [%]*	*WCRE [%]*	*ER [%]*	*ResNet-8*	*ResNet-14*	*ResNet-20*	*ResNet-26*	*ResNet-32*	*ResNet-38*	*ResNet-44*	*ResNet-50*
8 bit (exact)	100.0	0.00	0.00	0.00	0.00	0.00	82.85	85.81	88.09	89.70	88.22	89.67	88.13	89.35
mul8u_1446	99.2	0.018	0.29	0.13	28.57	9.38	82.43	85.64	88.18	89.99	87.99	89.70	88.14	89.17
mul8u_2P7	98.7	0.0015	0.0046	0.052	100.00	64.06	82.96	85.71	88.13	89.66	88.19	89.73	88.13	89.06
mul8u_EXZ	97.2	0.0014	0.015	0.033	28.57	19.53	82.67	85.85	88.07	89.73	88.04	89.67	88.15	89.33
mul8u_KEM	94.6	0.0046	0.017	0.18	100.00	75.00	82.52	85.70	88.31	89.78	88.07	89.59	88.01	89.28
mul8u_GS2	91.0	0.057	1.14	0.51	64.00	29.93	82.25	85.53	88.38	89.64	88.12	89.53	88.01	88.88
mmul8u_QJD	88.0	0.017	0.082	0.51	200.00	74.80	82.61	85.99	88.17	89.96	88.32	89.39	88.19	89.12
mul8u_7C1	84.1	0.13	2.38	1.04	64.00	39.93	79.95	85.00	87.67	89.87	88.18	89.28	87.89	88.82
mul8u_2AC	79.5	0.037	0.12	1.25	3100.00	98.12	81.37	85.59	87.70	89.81	88.23	89.36	87.55	88.29
mul8u_ZFB	77.7	0.059	0.45	0.80	43.56	69.26	82.03	85.76	87.96	89.63	88.28	89.37	87.49	88.01
mul8u_NGR	70.6	0.065	0.25	1.90	150.00	96.37	81.02	85.48	88.00	89.76	88.24	89.28	87.71	88.39
mul8u_PKY	65.0	0.25	2.79	1.99	64.00	64.73	68.94	82.86	86.39	89.48	88.21	88.71	86.26	88.21
mul8u_DM1	49.9	0.20	0.89	4.73	700.00	98.16	62.13	82.71	84.94	83.03	86.44	86.86	82.54	84.04
mul8u_12N4	36.3	0.43	2.15	4.20	80.00	87.31	19.30	16.47	22.52	20.80	26.01	26.02	11.18	15.56
mmul8u_1AGV	24.3	0.67	2.94	12.14	300.00	99.05	9.12	11.82	11.53	12.96	11.58	13.14	9.74	10.39
mul8u_FTA	21.5	0.89	4.29	13.96	125.00	98.74	8.47	13.66	11.58	9.46	10.36	12.29	12.96	11.73
mul8u_YX7	15.6	4.84	49.22	15.66	66.67	88.71	12.68	10.84	10.03	9.94	10.03	10.00	10.84	11.20
mul8u_JV3	8.7	2.15	8.21	39.78	7100.00	99.16	11.10	11.05	10.99	10.89	11.24	11.48	10.31	10.42
mul8u_18DU	7.9	2.28	9.08	28.42	100.00	99.16	9.22	10.20	9.75	9.65	10.01	10.02	11.09	10.64
Truncated 7-bit	75.4	0.19	0.78	2.65	100.00	74.61	48.64	77.38	72.32	60.75	78.11	79.83	62.69	64.32
Truncated 6-bit	48.5	0.58	2.32	7.00	100.00	93.16	12.09	9.99	11.77	11.16	10.82	12.49	10.84	10.16
BAM $h=0, v=2$	98.6	0.0019	0.0076	0.077	100.00	50.00	82.75	85.66	88.23	89.77	88.19	89.66	88.18	89.18
BAM $h=0, v=4$	90.6	0.019	0.075	0.56	100.00	81.25	82.86	86.07	88.19	89.90	88.34	89.74	87.97	89.20
BAM $h=1, v=3$	82.2	0.10	0.40	1.47	100.00	74.80	60.61	81.02	78.41	69.86	81.83	82.93	69.60	70.72
BAM $h=0, v=6$	73.9	0.12	0.49	2.64	100.00	93.75	63.51	78.60	71.27	62.92	78.57	79.25	54.21	61.39
BAM $h=1, v=6$	66.2	0.20	0.78	3.43	100.00	94.34	16.34	20.80	27.26	22.81	35.20	38.79	18.05	24.78
BAM $h=0, v=7$	60.5	0.29	1.17	5.21	100.00	96.48	10.39	12.16	15.01	13.67	15.74	18.17	10.62	11.81
BAM $h=2, v=7$	50.2	0.49	1.95	7.00	100.00	96.97	8.90	13.63	11.05	10.62	10.04	13.15	10.23	9.39
BAM $h=2, v=8$	38.7	0.78	3.13	10.56	100.00	98.14	10.13	11.29	11.53	10.06	10.09	13.35	11.30	12.43

are obtained if the convolutional layer of the third stage is approximated (see S=3, R=1, C=1 in Fig. 4). As this layer contains 28.2% of all the multipliers, it should undergo the approximation with the highest priority. Introducing the approximate multipliers to the first layer (consisting of 2.09% multipliers) makes a negligible contribution.

Table II provides a detailed characterization of selected 8-bit approximate multipliers and classification accuracy if these multipliers are employed in all convolutional layers of various instances of ResNet and evaluated on CIFAR-10. Evolved approximate multipliers are compared with common high-quality approximate multipliers created with truncation and BAM algorithm [7]. One can observe many distinct tradeoffs between classification accuracy and power budget needed for multiplication operations in convolutional layers. This detailed analysis enables the user to identify the best tradeoff for a particular application. For example, if the goal is to reduce power consumption of multipliers to 50% then it makes no sense to use more complex ResNet than ResNet-32 which shows 86.86% accuracy.

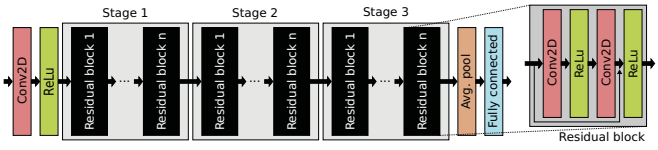

Fig. 3. Architecture of ResNet convolutional neural network

V. CONCLUSIONS

In this paper we presented a large library of approximate adders and multipliers which is primarily intended for creating approximate circuits needed in approximate implementations of complex applications such as energy-efficient hardware accelerators of DNNs. A subset of approximate 8-bit multipliers was then utilized in resilience analysis of ResNet. For various ResNet networks we obtained many tradeoffs between the classification accuracy and power consumption. This knowledge can be exploited during the hardware implementation of DNN accelerator.

Our future work will be devoted to applying the proposed error resilience analysis approach which is based on the existence of a large library of approximate components in other applications.

This work was supported by Czech Science Foundation project 19-10137S.

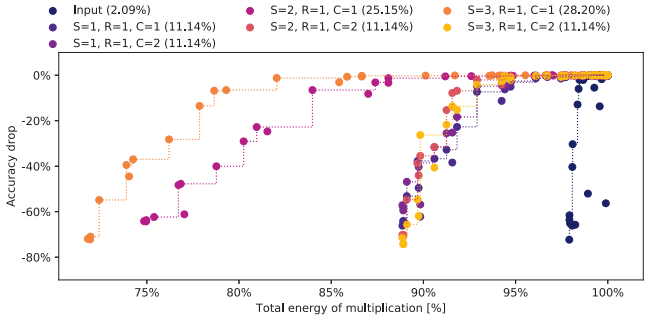

Fig. 4. The classification accuracy drop on CIFAR-10 and the power consumption drop measured when approximate multipliers are used in one layer of ResNet-8 (with reference classification accuracy 82.85%). Different layers are represented using different colors and characterized in terms of the number of stages (S), residual blocks (R), convolutional layers (C) and percentage of multipliers.

REFERENCES

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *The 50th Annual Design Automation Conference 2013, DAC'13*. ACM, 2013, pp. 1–9.

[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[3] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[4] P. Panda, A. Sengupta, S. S. Sarwar, G. Srinivasan, S. Venkataramani, A. Raghunathan, and K. Roy, "Invited – cross-layer approximations for neuromorphic computing: From devices to circuits and systems," in *53nd Design Automation Conference*. IEEE, 2016, pp. 1–6.

[5] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, "Understanding the impact of precision quantization on the accuracy and energy of neural networks," in *DATE*. EDAA, 2017, pp. 1478–1483.

[6] H. Jiang, C. Liu *et al.*, "A review, classification, and comparative evaluation of approximate arithmetic circuits," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 4, Aug. 2017.

[7] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, April 2010.

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

[9] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[10] L. Sekanina, Z. Vasicek, and V. Mrazek, *Automated Search-Based Functional Approximation for Digital Circuits*. Springer International Publishing, 2019, pp. 175–203.

[11] J. F. Miller, *Cartesian Genetic Programming*. Springer-Verlag, 2011.

[12] M. Ceska, J. Matyas *et al.*, "Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished," in *Proc. of 36th IEEE/ACM Int. Conf. On Computer Aided Design*. IEEE, 2017, pp. 416–423.

[13] Z. Vasicek, "Formal methods for exact analysis of approximate circuits," *IEEE Access*, p. 24, 2019.

[14] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, 2018.

[15] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 2, pp. 16:1–16:23, 2018.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[17] M. Abadi, A. Agarwal *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, "TFApprox: Towards a fast emulation of DNN approximate hardware accelerators on GPU," in *DATE*. EDAA, 2020, pp. 1–4. [Online]. Available: https://github.com/ehw-fit/tf-approximate