ASNet: Introducing Approximate Hardware to High-Level Synthesis of Neural Networks

Saman Froehlich Lucas Klemmer Daniel Große Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH and Group of Computer Architecture, University of Bremen, Germany
froehlich@cs.uni-bremen.de lucas.klemmer@dfki.de grosse@cs.uni-bremen.de drechsle@cs.uni-bremen.de

***Abstract*—Approximate Computing is a design paradigm which makes use of error tolerance inherent to many applications in order to trade off accuracy for performance. One classic example for such an application is machine learning with Neural Networks (NNs). Recently, LeFlow, a High-Level Synthesis (HLS) flow for mapping Tensorflow NNs into hardware has been proposed. The main steps of LeFlow are to compile the Tensorflow models into the LLVM Intermediate Representation (IR), perform several transformations and feed the result into a HLS tool.**

In this work we take HLS-based NN synthesis one step further by integrating hardware approximation. To achieve this goal, we upgrade LeFlow such that (a) the user can specify hardware approximations, and (b) the user can analyze the impact of hardware approximation already at the SW level. Based on the exploration results which satisfy the NN quality expectations, we import the chosen approx. HW components into an extended version of the HLS tool to finally synthesize the NN to Verilog. The experimental evaluation demonstrates the advantages of our proposed ASNet for several NNs. Significant area reductions as well as improvements in operation frequency are achieved.

I. Introduction

Approximate Computing (AC) is an emerging field of research which deals with exploiting the inherent error tolerance of applications. AC tries to improve the performance of these applications in terms of computation time, power consumption and/or HW complexity by introducing errors which are usually either timing induced or caused by functional approximation. The applications of AC range from machine learning and digital image processing to robotics, just to name a few.

Neural Networks (NNs) have been successfully utilized to solve problems of applications in the context of machine learning and pattern recognition. NNs usually consist of multiple different layers with many neurons and filters leading to very complex structures with many parameters. These parameters are adjusted (trained) with a set of input-output pairs and finally evaluated wrt. a test data set.

In order to deal with the complexity of NNs, different SW tools have been published to model and work with these structures efficiently. One of the most commonly used frameworks is *Tensorflow* (TF) [1]. TF is an open source machine learning library which utilizes data flow graphs. For the *High-Level Synthesis* (HLS) of NNs, recently, LeFlow [2] has been proposed. LeFlow is a HLS design flow which allows to generate synthesizable Verilog for TF models of NNs via the HLS tool LegUp [3].

Since NN models can be very complex and computing intensive, novel approaches are needed to boost the performance of NN implementations. Different approaches have been proposed. The most common approaches are quantization and pruning methods (e.g. [4], [5], [6]) which aim to reduce the size and complexity of NNs. In general, all these approaches either reduce the size of the needed memory (weight quantization) or modify the architecture of the NN (pruning). However, none of these approaches allow to use and evaluate custom approx. HW to further increase the performance of the final HW implementation. A lot of different AC architectures exist which allow to reduce power consumption, HW complexity and/or delay of functional units (e.g. [7], [8], [9]). However, the large range of possible AC configurations for NNs leads to the need of efficient approaches for design space exploration and prototyping.

In this paper, we present such an approach for *Approximate High-Level Synthesis of Neural Networks* (ASNet[1]). A core component of ASNet is the novel *Approximation Test Bed* (ATB) which allows to perform approx. design space exploration on the SW level, i.e. the user can evaluate whether the quality expectations (e.g. classification error of the NN) are still met when integrating approx. HW components.

To the best of our knowledge, ASNet is the first HLS approach which allows to automatically evaluate approx. HW wrt. NN accuracy. It enables fast prototyping and testing of different configurations.

In the experiments we show the applicability of ASNet by applying ASNet to different NNs, data sets and HW configurations. We show that using ASNet, HW configurations can be analyzed wrt. their impact on classification accuracy before synthesis is performed and feasible configurations can be identified. Finally, ASNet can be used to perform the synthesis of the feasible HW configurations which are analyzed wrt. their performance in terms of area and operation frequency.

II. Related Work

To the best of our knowledge no other *Approximate High-Level Synthesis* (AHLS) tool tailored for NNs and therefore includes mechanisms for quality evaluation of the generated designs has been proposed. Nevertheless, we present some of the general AHLS tools in this section.

In [10] a general *Approximate High-Level Synthesis* (AHLS) flow has been introduced. This flow focuses on using approx. HW components. An analytic error-model is proposed, and for scheduling an iterative algorithm is presented. The advantage of the scheduling algorithm is that it allows to merge approx. operations with exact operations avoiding unnecessary overhead. In ASNet, the approximation is incorporated only after

This work was supported by the German Research Foundation (DFG) within the project PLiM (DR 287/35-1) and project number 276397488 – SFB 1232 in subproject P01 'Predictive function'.

[1]ASNet is available at https://github.com/LucasKl/asnet

all optimizations have been done. This way, the question of handling relationships between approx. and exact operations is avoided. Just like [10] other work has presented analytic error-models (e.g. [11]). However for analytic error-models, an error-model for each used approx. component is needed. Further, when computing how an error propagates many computations are needed to have an estimation for the actual error and scalability becomes an issue. Since the quality of NNs is evaluated wrt. test data sets, there is no need for an analytical model for these kinds of applications.

The authors of [12] provide an AHLS tool which is able to generate RTL from a given C-file using approx. HW. The tool allows to approx. operations by rounding or eliminating them completely. [12] provides the option to use voltage scaling for approximation by employing an energy model. In contrast, ASNet can directly work with TF NNs which are defined in Python and ASNet is tailored for the evaluation of the effects of approx. HW on the given network. Further, instead of having a fixed set of approximation operations (rounding, elimination of operations etc.), ASNet allows the use of custom approx. HW which can be tailored for the specific application and thus is much more effective.

[13] presents ABACUS, an automatic synthesis method for generating approx. circuits. Given a behavioral or a RTL description, multiple approx. variants are generated using an Abstract Syntax Tree. These approx. variants are synthesized and compared to the exact design wrt. accuracy and design metrics. While ABACUS allows for approx. HLS, the accuracy of the designs can only be evaluated after the synthesis has been performed. ASNet allows for accuracy evaluation at SW level. Thus, designs for which error bounds do not hold can be eliminated before the synthesis step. Further, for ASNet, the NNs are specified in Python and ASNet allows for the use of custom approx. HW designs.

ALWANN [14] is an efficient tool for design space exploration of approximate HW designs for NNs. The approx. HW designs are integrated into a custom TF operation and an evolutionary algorithm is used to find the best configuration. However, compared to ALWANN we present an AHLS flow, which ultimately leads to synthesizable Verilog output. Thus, the user does not need to reconstruct the NN in Verilog himself.

III. Preliminaries

A. LegUp

LegUp [3] is an open source HLS tool which can be used to synthesize High-Level descriptions of HW systems to Verilog. It features two modes: pure HW and a HW/SW hybrid flow. We utilize the pure HW flow in this paper, but an extension to a HW/SW hybrid flow is possible. At its core LegUp features the LLVM framework and LLVM-IR[2] which is used by LeFlow to pass the description of the NNs to LegUp. Instructions in LLVM-IR correspond directly to HW operations. LegUp performs the classic HLS steps on the IR which are allocation, scheduling, binding and generation of HDL [16]. Further,

[2]The LLVM project [15] is a collection of modular and reusable compiler and toolchain technologies. The core of LLVM is the *Intermediate Representation* (IR), a low-level programming language similar to assembly.

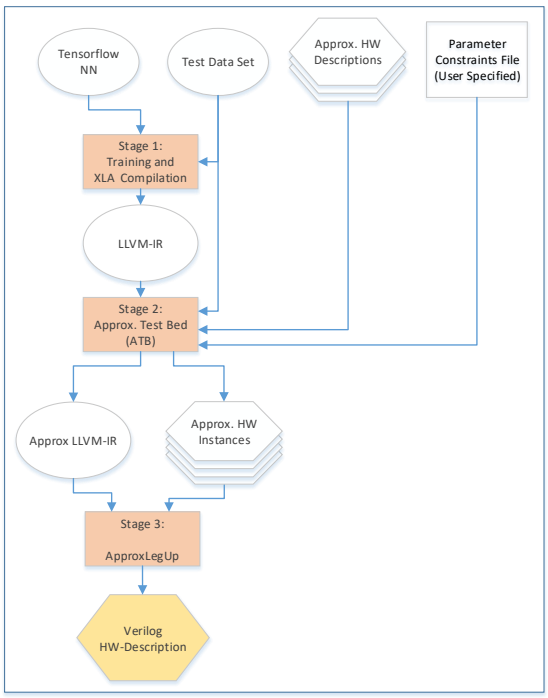


Fig. 1. ASNet Overview

LegUp allows for several optimizations such as loop unrolling and dead code removal.

Eventhough LegUp is open source, according to [3] LegUp produces HW implementations that are of comparable quality to commercial HLS tools.

B. LeFlow

LeFlow is a design flow, that is based on LegUp. Instead of feeding a C-Program into LegUp, LeFlow uses TF and Googles XLA compiler [17] to compile a NN description (given in Python) to an optimized LLVM-IR.

LeFlow has two stages: The first stage is implemented in Python. In this stage the description of the NN is imported and trained. After compiling it to LLVM-IR using Googles XLA compiler, the LLVM-IR is restructured such that it can be fed to LegUp. In the second stage, the restructured LLVM-IR is fed to LegUp which generates synthesizable Verilog.

IV. ASNet: Approximate High-Level Synthesis of Neural Networks

In this section we introduce ASNet. At first, we present a general overview of ASNet. Then, all stages of ASNet are described in detail in the respective subsections.

A. ASNet Overview

In general, there are two variants of approx. HW descriptions. The first is a fixed approx. HW description. The second is a parametrized description, where the degree of approximation can be adapted using one or more parameters (for example an approx. multiplier which ignores mantissa bits; the number of mantissa bits can be controlled by a parameter). Examples for fixed HW designs can be found in [18]. Examples for parametrized HW designs are described in [8], [9]. ASNet can handle both fixed and parametrized HW designs via the Approximate Test Bed (see stage 2).

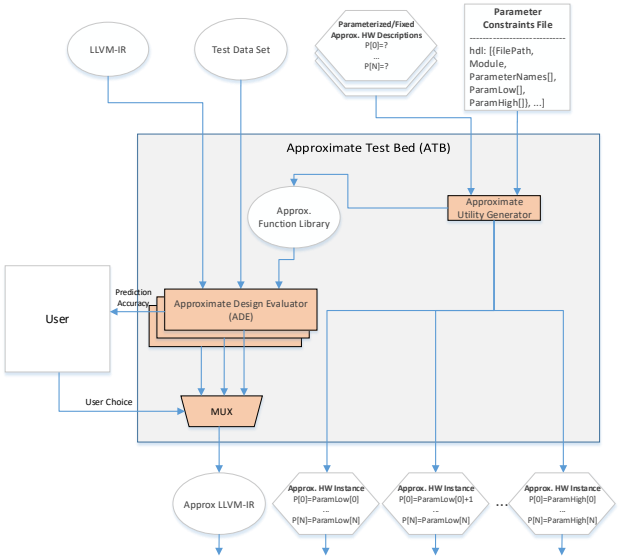


Fig. 2. Approximate Test Bed (ATB)

Algorithm 1 Approximate Test Bed (ATB)

```
1: function EvalDesignSpace(TestData, IR, HWDescr, Params)
2:   [ApproxLib, HWInst] = ApproxUtilGen(Params, HWDescr)
3:   ADEs = GenerateADEs(ApproxLib, IR, TestDataSet)
4:   Accuracies = emptySet()
5:   for all ADE in ADEs do
6:     Accuracies.append(ADE.evaluateConfiguration())
7:   end for
8:   SelConfig=PresentAccuracies(Accuracies)
9:   return SelConfig, HWInst
10: end function
```

Figure 1 gives an overview of ASNet. The user provides four inputs to ASNet (see top of Figure 1): The first is the definition of the TF NN in Python. The second is the test data set which is used to evaluate the prediction accuracy of the NN. The next parameter is the set of potentially suitable approx. HW descriptions in Verilog. Finally, the user specifies the constraints on the parameters of each approx. HW description in a *Parameter Constraints File* (JSON-format).

In the following subsections, we first describe the stages of ASNet. In Section IV-B, we give a short description of the first, unchanged stage wrt. LeFlow. Successively, we introduce the Approximate Test Bed (Stage 2) in Section IV-C. We conclude this section by describing ApproxLegUp (Stage 3) in Section IV-D.

B. Stage 1: Training and XLA Compilation

In the first stage ASNet follows LeFlow. The NN is created and trained in TF using a training data set. Its final quality is evaluated wrt. a test data set and compiled to LLVM-IR using Googles XLA compiler.

C. Stage 2: Approximate Test Bed (ATB)

In order to choose a suitable configuration of approx. HW components for a HW NN implementation, an efficient approach for the exploration of the approx. design space is needed. For this purpose ASNet includes an *Approximate Test Bed* (ATB). The ATB is depicted in Figure 2. The approx. design space is formed by the Cartesian product of the approx. HW instances. Suitable configurations (a configuration is a set of approx. HW instances) of the approx. design space are found by evaluating them using generated *Approximate Design Evaluators* (ADEs) (depicted on the left side of Figure 2 and detailed in Section IV-C2). The user can define the parameter space of each approx. HW description and thus the approx. design space inside the Parameter Constraints File. Each ADE evaluates the impact of a single approx. HW configuration on the prediction accuracy of the NN at the SW level. The Approximate Utility Generator generates the Verilog code of every approx. HW instance (see right side of Figure 2). A function library that wraps each approx. HW instance into a function is also generated by the Approximate Utility Generator to provide an interface for the ADEs. The ADEs use this interface to substitute regular operations of the IR code ('fmul', 'fdiv', ...) with their approx. counterpart.

The execution sequence of the ATB is presented in Algorithm 1. First, the Approximate Utility Generator is used to generate the approx. function library and to instantiate the approx. HW descriptions in Line 2, given the Parameter Constraints File and the approx. HW descriptions. Then, the ADEs are generated and evaluated in Lines 3-7. Finally, the evaluation results are presented to the user, who chooses the best approx. HW configuration. This configuration is returned together with the approx. HW instances in Line 9.

In the following we describe the Approximate Utility Generator and the ADE in detail.

1) Approximate Utility Generator: To measure the impact of HW approximation on the prediction accuracy of a NN at SW level, the accurate operations of the LLVM-IR need to be substituted by their approx. counter parts. In order to generate an approx. function library which implements these approx. counter parts, we have developed the Approximate Utility Generator. First, the Approximate Utility Generator instantiates parametrized approx. HW descriptions as approx. HW instances. The user specifies the feasible values of each parameter inside the Parameter Constraints File, i.e. ParamLow and ParamHigh. Each approx. HW description is instantiated for each feasible parameter value. Fixed approx. HW descriptions can be used directly.

The Approximate Utility Generator compiles a C++-simulator object for each approx. HW instance using Verilator [19]. A unique ID is assigned to each of these objects and they are grouped inside a wrapper function. This wrapper function is responsible for providing the inputs, running and evaluating the approx. HW instance and returning the result. Besides the inputs of the approx. HW instances, the ID of the HW instances is passed to the wrapper function. All wrapper functions are combined into an *approximate function library*.

2) Approximate Design Evaluator (ADE): The ADEs are responsible for evaluating the impact of approx. HW configurations on the prediction accuracy. Therefore, our ADEs provide an execution environment that takes care of input/output handling, parameter loading and result evaluation. In this execution environment, accurate operations in the unapproximated LLVM-IR are substituted by calls to their approx. counterparts from the approx. function library. An example for such a substitution is given in Listing 1. The call to the accurate

operation is depicted in Line 1. Line 2 shows the call to the approx. counterpart. The float operands %1 and %2 are passed together with the ID of the approx. HW instance (in this case 31, a 32-bit integer). After compilation, the call is redirected to the approx. function library.

Listing 1. LLVM-IR substitution

```
1   %3 = fmul float %1, %2
2   %3 = call float @_Z10approx_mulffj(float %1, float %2, i32 31)
```

After generating the ADE for a concrete approx. HW configuration, the ADE executes the approx. LLVM-IR representation of the NN on the test data set and evaluates the prediction accuracy.

D. Stage 3: ApproxLegUp

After determining which approx. HW components to use and how to configure the parameters, the selected configuration needs to be transmitted to the HLS tool for Verilog generation. In order to do this, we have modified the LLVM-IR in such a way, that a binary operator can have three parameters: besides the first and the second operand, a unique identifier for the selected approx. HW description is passed.

The identifier points to a Verilog file which implements the selected HW with the previously determined parameter configuration. We have upgraded LegUp to ApproxLegUp which is able to parse the modified LLVM-IR and substitute the functions calls in the output Verilog implementation of the NN with the approx. HW.

V. Experimental Results

We demonstrate the advantages of ASNet by applying it to three NNs using a variety of approx. floating-point (FP) multipliers as well as an configurable approx. FP divider. We have evaluated the NNs wrt. the well-known MNIST [20] and SVHN [21] data-sets. All networks have been trained without approximation.

This section is structured as follows: In Section V-A we give a general overview of the used NNs. Section V-B introduces the approx. HW descriptions. Subsequently, in Section V-C, we describe the setting of the experiments. Section V-D presents the results of the approx. design space exploration. In Section V-E, we present the evaluation for the final approx. architectures of each NN, which are based on the approx. design space exploration using the ATB of ASNet. Finally, in Section V-F, we demonstrate the applicability of ASNet to other approx. HW designs.

A. Overview of NNs

The first NN is the *classificationMNIST* network. It comes with LeFlow and implements a simple feed-forward NN with a single layer. This layer has 10 neurons and uses the softmax activation function.

Next, a convolutional NN for the SVHN data set is considered. It has two convolutional layers with 8 and 16 filters, respectively, each followed by a relu layer and a maxPooling layer. After the convolutional layers, a dense layer with 20 neurons and a linear activation function is used, followed by a 10 neuron output layer with softmax as activation function. The resulting NN is called *convolutionalSVHN*.

The third NN benchmark is called *convolutionalMNIST*. The architecture of this convolutional NN is similar to that applied to the SVHN data set. However, since the MNIST data set consists of 28x28x1 images (compared to 32x32x3 for the SVHN data set), we have adjusted the architecture of the convolutional NN accordingly. To show the generality of ASNet, we apply different approximation methods to convolutionalMNIST than to the other two NNs.

B. Approximate Hardware

For our experimental evaluation we have used two different approximation methodologies. The first has been proposed in [9] where the authors have introduced an approx. FP multiplier which ignores the last p bits of the mantissa. We have instantiated a 32-bit multiplier for FP multiplications with single precision and let p be in the range from 0 to 22. For evaluation purposes, we have also designed an approx. FP divider in the same way, i.e. ignoring bits of the inputs, in Verilog.

To show the generality of our approach, we have also designed a 32-bit approx. FP multiplier based on approx. integer multipliers from [22]. The authors of [22] propose to generate approx. multipliers using *Cartesian Genetic Programming* (CGP), and the generated approx. HW can be downloaded from [23]. Since this HW is generated using CGP, the area and delay may not necessarily decrease monotonically with the accuracy.

However, ASNet can be used to find the multiplier architectures for which the resulting HW implementation of the corresponding NN meets accuracy requirements and to do the final synthesis. The resulting approx. FP multiplier has a parameter m which specifies which approx. multiplier is to be used internally.

C. Experimental Setup

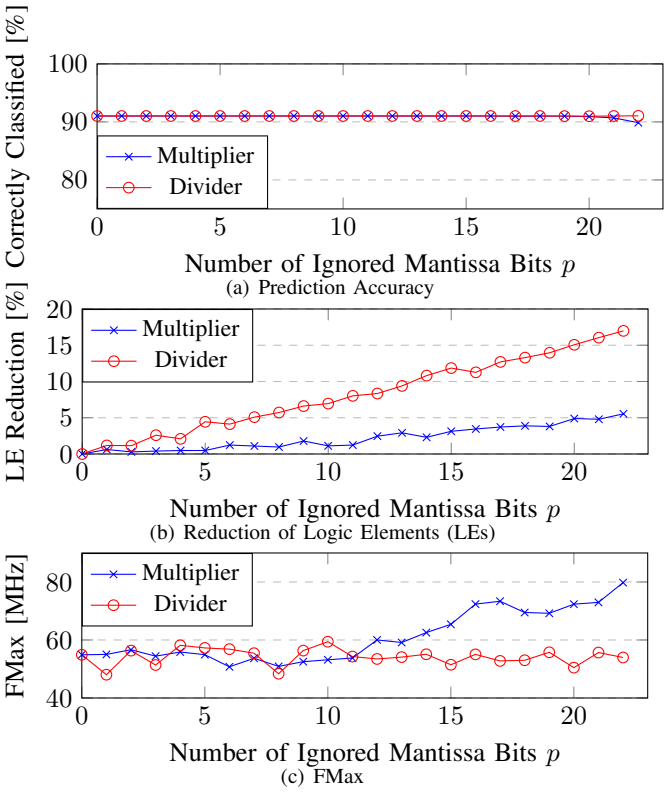
In order to evaluate the final designs, we use the Quartus II 64-Bit V. 15.0.0 Build 145 SJ Web Edition to synthesize the final Verilog file to a CycloneV FPGA on a DE1-SoC board.

Evaluating the accuracy of all options for an approx. HW component using 10,000 test images with ASNet at SW level has been executed in less than 10 minutes (less than 30 seconds per configuration) inside a virtual box for classificationMNIST using only a single thread. For convolutionalSVHN the evaluation with 3,000 test images has been carried out in about 3.5 hours (i.e. less than 10 minutes per configuration).

D. Approximate Design Space Exploration

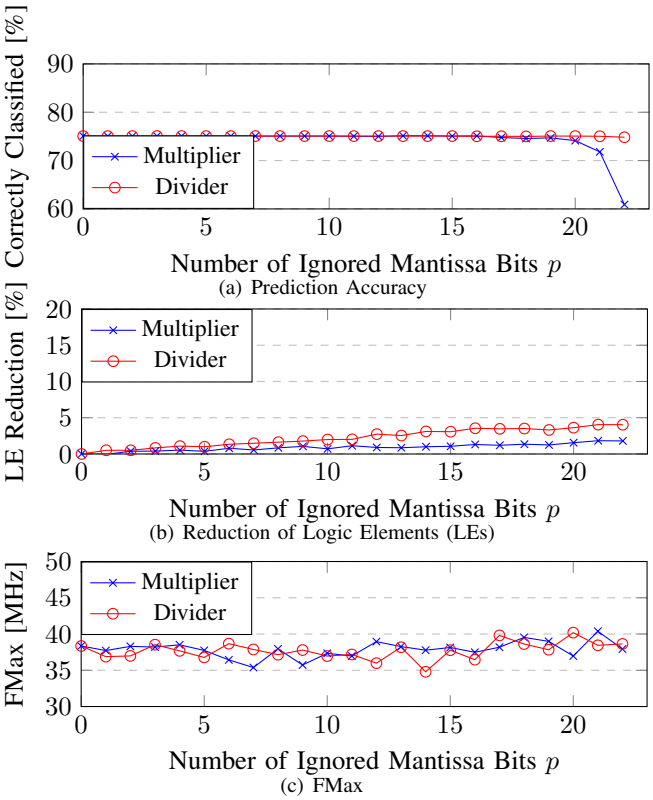
1) classificationMNIST: We have evaluated the approx. FP multiplier and divider in the respective 23 different configurations for the classificationMNIST network using ASNet. Results can be seen in Figure 3(a)-Figure 3(c). Figure 3(a) shows the prediction accuracy for each value of p, respectively. Figure 3(b) shows the reduction in used *Logic Elements* (LEs), while Figure 3(c) shows the resulting frequency FMax. The blue lines show the results when the approx. FP multiplier is used, while the red lines show the result for the divider.

As can be seen in Figure 3(a), the accuracy for the approx. multiplier remains almost constant for up to 20 ignored



(a) Prediction Accuracy
(b) Reduction of Logic Elements (LEs)
(c) FMax

Fig. 3. classificationMNIST



(a) Prediction Accuracy
(b) Reduction of Logic Elements (LEs)
(c) FMax

Fig. 4. convolutionalSVHN

mantissa bits and the accuracy only starts to drop slightly at $p = 21$ (89.87% at $p = 22$ vs 90.69% at $p = 21$). For the divider the influence on the accuracy is even smaller, since the division is only used in the softmax function. The softmax function is only used to normalize the output and since the differences between incoming values are large, the largest output value stays the same, even if the mantissa is shortened during the normalization step. Thus the influence of the approx. FP divider on the accuracy is negligible.

While using approximation during multiplication and division has only little influence on the results, large gains in the maximum operation frequency and the utilization of LEs can be observed.

Figure 3(b) shows that the number of utilized *Logic Elements* (LEs) is reduced by up to 5%. For the FP divider an even greater LE reduction can be observed, i.e. up to 16.9% can be achieved without a notable loss in prediction accuracy.

Figure 3(c) shows the maximum operation frequency FMax. We can see that using approximation during the multiplication allows to increase FMax from 54.89 MHz to 79.79 MHz and thus increase the maximum operation frequency FMax by about 45%. The divider allows for almost no increase in operation frequency. This is because the divider is not on the critical path.

Overall, the results show that very fast approx. design space exploration (accuracy evaluation for all parameters of an approx. HW component) is possible which leads to considerable LE reduction and substantial improvement of FMax.

2) convolutionalSVHN: Unlike classificationMNIST, convolutionalSVHN features convolutional layers. The results of the evaluation are shown in Figure 4(a) - Figure 4(c). We use the same coloring as in the classificationMNIST graphs and perform the same approx. design space exploration (i.e. ignoring p mantissa bits using the approx. FP multiplier and the approx. FP divider). Eventhough convolutionalSVHN is more complex than classificationMNIST, the accuracy only starts to be reduced significantly if more than 20 mantissa bits are dropped during the multiplication. However compared to classificationMNIST, the drop is steeper. This is due to the increased number of multiplications in the NN. Again, the usage of an approx. FP divider does not reduce the accuracy of the NN significantly and allows for more LE reduction than the approx. FP multiplier, while the approx FP multiplier allows for a higher increase in operation frequency FMax.

TABLE I
FINAL DESIGNS FOR CLASSIFICATIONMNIST AND CONVOLUTIONALSVHN

Name	p_m, p_d	Correctly Classified	LE red.	FMax Gain
classificationMNIST	20, 22	88.72%	24.66%	35.62%
convolutionalSVHN	20, 22	73.70%	6.42%	2.88%

E. Final HW Designs

Based on the results of the approx. design space exploration obtained with ASNet and reported in the previous section, we have chosen approximation levels for the approx. FP multiplier and the approx. FP divider for each NN. The results are summarized in Table I. The first column shows the name of the NN, while the second column lists the used values for p for the multiplier and the divider (p_m and p_d), respectively. The evaluated accuracy of the network can be seen in the third column. The forth column gives the reduction of LEs and the last column provides the gain of the operation frequency

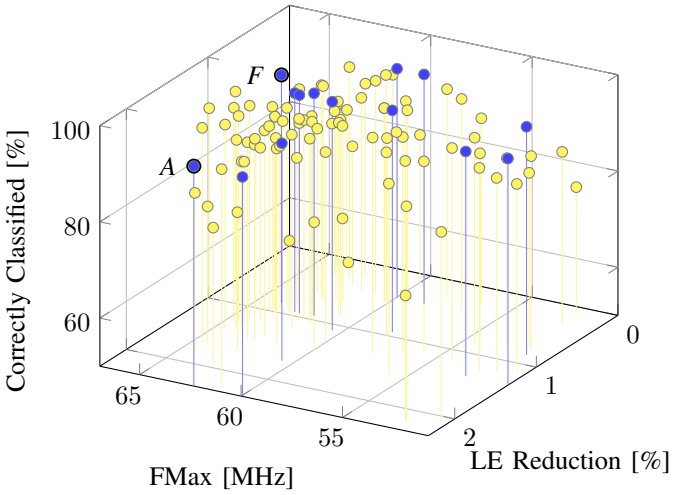


Fig. 5. convolutionalMNIST with approx multipliers

FMax. As can be seen the reduction in the number of LEs is up to 24.66% and the gain in FMax up to 35.62%.

F. Additional Approximation Techniques

To conduct further experiments based on alternative approximation techniques and to demonstrate that ASNet can handle them, we have modified the internal structure of the floating point multipliers by using more than 100 different approximate 32-bit integer multipliers presented in [23] for the multiplication of the mantissa. The resulting floating point multiplier has a parameter which defines which of the over 100 approximate multipliers is to be used for the multiplication. Using ASNet, we have evaluated how the approximate multipliers influence the accuracy of convolutionalMNIST. The results can be seen in Figure 5. The graph shows the resulting accuracy, LE-reduction (with the size of the largest HW design as baseline) and operation frequency for each HW design respectively.

Since the approximate multipliers are generated using a CGP approach, there is no strict link between accuracy, area and operation frequency. However, ASNet can be used to filter the designs which meet the accuracy requirements. If for example an accuracy of more than 95% is required, more than 86% of the HW designs can be dropped already at the evaluation step (and thus before synthesis) and the number of designs which need to be synthesized is significantly reduced. In Figure 5 the designs which meet the accuracy requirement of 95% are marked with blue dots, while the rest is marked in yellow. Finally, the designs which meet the accuracy requirements can be synthesized with ASNet and an appropriate HW design can be determined. If the circuit is to be optimized wrt. to area, the design which results in the largest LE reduction can be selected (labeled *A* in Figure 5 with a total LE reduction of 2.324% and an operation frequency FMax of 62.33 MHz). However, if the circuit is to be optimized wrt. to frequency, the circuit with the highest operation frequency FMax can be chosen (labeled *F* in Figure 5 with a total LE reduction of 0.852% and an operation frequency FMax of 63.93 MHz).

VI. Conclusions

In this paper we have presented ASNet, an approach for *Approximate High-Level Synthesis of Neural Networks*. ASNet allows (a) integration of custom approx. HW descriptions plus the evaluation of the resulting quality of a given NN on its test data set, and (b) design space exploration for parametrized approx. HW descriptions via the *Approximate Test Bed* (ATB) already at the SW level. With the ATB the effects of the approx. HW on the accuracy can be evaluated *before* performing synthesis.

In the experiments we have evaluated a broad range of approx. HW for three different NNs. We have shown that significant reductions in the number of LEs (up to 24.66%) and major gains (up to 35.62%) in the operation frequency for an FPGA with negligible accuracy loss can be achieved.

Finally, we make ASNet open source to stimulate further research and development of approx. HLS methodologies for NNs.

References

[1] Martín Abadi et. al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[2] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," *ArXiv e-prints*, Jul. 2018.

[3] A. Canis et. al., "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[4] W. Pan, H. Dong, and Y. Guo, "Dropneuron: Simplifying the structure of deep neural networks," *CoRR*, vol. abs/1606.07326, 2016.

[5] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[6] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, Oct 2017, pp. 1921–1925.

[7] A. Qureshi and O. Hasan, "Formal probabilistic analysis of low latency approximate adders," *TCAD*, pp. 1–1, 2018.

[8] S. Hashemi, R. I. Bahar, and S. Reda, "Drum: A dynamic range unbiased multiplier for approximate applications," in *ICCAD*, 2015, pp. 418–425.

[9] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," in *TVLSI*, vol. 8, No. 3, Jun. 2000, pp. 273–285.

[10] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *DAC*, 2015, pp. 1–6.

[11] S. Lee, D. Lee, K. Han, E. Shriver, L. K. John, and A. Gerstlauer, "Statistical quality modeling of approximate hardware," in *Int'l Symp. on Quality Electronic Design*, 2016, pp. 163–168.

[12] S. Lee, L. K. John, and A. Gerstlauer, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *DATE*, 2017, pp. 187–192.

[13] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014, pp. 1–6.

[14] V. Mrazek, Z. Vasícek, L. Sekanina, M. A. Hanif, and M. Shafique, "ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining," *CoRR*, vol. abs/1907.07229, 2019. [Online]. Available: http://arxiv.org/abs/1907.07229

[15] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[16] Stefan Hadjis et. al., "Profiling-driven multi-cycling in fpga high-level synthesis," in *DATE*, 2015, pp. 31–36.

[17] Google - TensorFlow, "XLA," https://www.tensorflow.org/xla/, 2018.

[18] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *DATE*, 2017.

[19] W. Snyder, J. Coiner, D. Galbi, and P. Wasson, "Verilator," https://www.veripool.org/wiki/verilator, 2018.

[20] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 2018.

[21] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *IPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[22] M. Češka, J. Matyaš, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar, "Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished," in *ICCAD*, Nov 2017, pp. 416–423.

[23] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Approximate adders and multipliers library," http://www.fit.vutbr.cz/research/groups/ehw/approxlib/, 2016.