2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00069

HALF: Holistic Auto Machine Learning for FPGAs

Jonas Ney*§, Dominik Loroch†§, Vladimir Rybalkin*§, Nico Weber†, Jens Krüger† and Norbert Wehn*
* University of Kaiserslautern, Kaiserslautern, Germany
† Fraunhofer ITWM, Kaiserslautern, Germany
{ney, rybalkin, wehn}@eit.uni-kl.de, {dominik.loroch, nico.weber, jens.krueger}@itwm.fraunhofer.de

***Abstract*—Deep Neural Networks (DNNs) are capable of solving complex problems in domains related to embedded systems, such as image and natural language processing. To efficiently implement DNNs on a specific FPGA platform for a given cost criterion, e.g., energy efficiency, an enormous amount of design parameters must be considered from the topology down to the final hardware implementation. Interdependencies between the different design layers must be taken into account and explored efficiently, making it hardly possible to find optimized solutions manually. An automatic, holistic design approach can improve the quality of DNN implementations on FPGA significantly. To this end, we present a cross-layer design space exploration methodology. It comprises optimizations starting from a hardware-aware topology search for DNNs down to the final optimized implementation for a given FPGA platform. The methodology is implemented in our Holistic Auto machine Learning for FPGAs (HALF) framework, which combines an evolutionary search algorithm, various optimization steps, and a library of parametrizable hardware DNN modules. HALF automates both the exploration process and the implementation of optimized solutions on a target FPGA platform for various applications. We demonstrate the performance of HALF on a medical use case for arrhythmia detection for three different design goals, i.e., low-energy, low-power, and high-throughput. Our FPGA implementation outperforms a TensorRT optimized model on an Nvidia Jetson platform in both throughput and energy consumption.**

***Index Terms*—Neural Architecture Search, NAS, FPGA, Hardware Library**

I. Introduction

Efficient implementation of Deep Neural Networks (DNNs) in hardware requires rigorous exploration of the design space on different layers of abstraction, including *algorithmic*, *architectural* and *platform* layers, as it is depicted in Fig. 1. First, the *application* is defined by a dataset and requirements in terms of *constraints* and *objectives*, e.g., accuracy, latency, etc. At the highest level in the design hierarchy is the *algorithm*, which is the most abstract description of the data and control flow in the form of a DNN *topology*. The *architecture* layer maps the topology to a *hardware design*, which is implemented on the platform. At the lowest level is the *platform*, which describes the hardware and its physical properties. All design layers introduce a large number of design choices as well as interdependencies (see Fig. 1).

In our methodology, the architectural layer is represented by a highly parametrizable architecture template that is used to instantiate various DNN topologies in hardware. We formulate hardware models for latency, power, and energy derived from the architecture template expressed in terms of topology hyperparameters to include hardware awareness in the algorithmic layer. The hardware models together with the architecture template describe a bridge between the algorithmic and platform layer. However, this one bridge is not enough, since the dependence of the application to the architecture layer cannot be formalized. To this end, we use Neural Architecture Search (NAS) in the algorithmic layer and augment it with the hardware-awareness models of the architecture layer. The NAS performs a full cross-layer optimization, which is guided by optimization objectives targeting both application requirements and hardware performance. In summary, the hardware-aware NAS in strong coupling with the architecture template and its modeling spans a bridge from the application down to the platform layer, which allows a fully automatic design flow for optimized implementations.

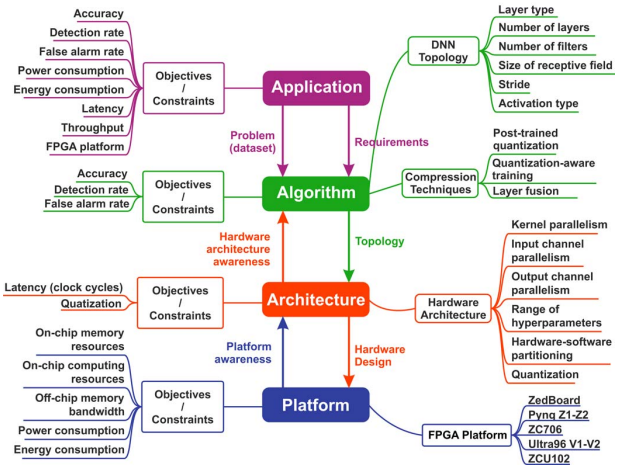

Fig. 1: Design hierarchy and multi-layer design space.

The methodology is implemented in the Holistic Auto machine Learning for FPGAs (HALF) framework, which comprises a hardware-aware evolutionary NAS and an Field-Programmable Gate Array (FPGA) implementation framework plus a hardware library. We demonstrate the efficiency of our approach in a case study on energy-efficient FPGA implementations for atrial fibrillation detection in a real-world application scenario using a dataset provided by the Charité in Berlin. The novel contributions of the paper are:

- A design space exploration methodology, which enables cross-layer optimization of DNN for efficient hardware implementation.
- A framework that automatically produces low-energy, low-power or high-throughput FPGA solutions.
- A library of parameterizable low-power, ultra-low-latency

§first three authors contributed equally

978-1-6654-3759-2/21/$31.00 ©2021 IEEE
DOI 10.1109/FPL53798.2021.00069

hardware architectures of DNN layers.
- We demonstrate FPGA implementations for arrhythmia detection targeting different application scenarios which outperform an embedded GPU with respect to throughput, power and energy efficiency.

II. Related work

A. Neural Architecture Search

For the task of anomaly detection in Electrocardiogram (ECG) data, [1; 2; 3; 4] demonstrate that 1D Convolutional Neural Networks (1D-CNNs) can be used with raw ECG samples without any hand-crafted feature extraction. Instead of manually designing DNN topologies for a specific detection task, NAS describes methods which automatically explore the search space spanned by DNNs architectures. Among the most successful techniques are gradient-based and evolutionary methods. Recently, there is less activity around reinforcement learning (RL) based approaches, as both gradient-based and evolutionary methods can produce similar results in often significantly less time [5; 6]. Nevertheless, RL methods are explored in combination with hardware-awareness for hardware accelerators [7] and FPGAs [8]. The most prominent gradient-based method is differentiable architecture search (DARTS) [9], which searches for subgraphs in a larger, differentiable supergraph. [5] introduced hardware-awareness to DARTS in the form of latency minimization for mobile phones. The authors of [10] used a very similar DARTS setup as [5], but for FPGAs, and the layer latencies were modelled as functions of the topology hyperparameters instead of lookup tables. Although DARTS is very fast, the structure of the manually designed supergraph can impose a strong bias. In contrast, evolutionary algorithms do not require a supergraph. Genetic algorithms use an encoding to describe the structure of the DNNs, which enables biologie inspired concepts like crossover [11] and aging [6] to be implemented. Network morphisms are a class of operators on the graph structure, which change the DNN such that retraining from scratch is not necessary [12; 13]. [13] uses network morphisms in their LEMONADE algorithm, which uses a bayesian method to generate DNNs in a multi-objective search, although not hardware-aware. Both [13] and [8] distinguish computationally ”cheap” and ”expensive” objectives, and skip unnecessary, ”expensive” computations of bad candidates. Later, [14] augmented LEMONADE with hardware-awareness and error resilience.

B. Automatic Hardware Generation and Hardware Architectures for DNN

Among the most used frameworks for automatic hardware generation are FINN [15], Xilinx ML Suite [16] that is a toolchain for development on xDNN [17] general processing engine, and Deep Neural Network Development Kit (DNNDK) [18] that is an SDK for Deep Learning Processor Unit (DPU) [19] programmable engine. However, none of them is a fully automatic approach, but rather a collection of tools that help to convert a DNN into a custom FPGA accelerator, like FINN or compile them to a sequence of instructions executed on a programmable engine. All frameworks provide tools for DNN compression and optimization and runtime support.

Both Xilinx ML Suite and DNNDK target DNN execution on programmable engines that are designed to support a wide range of DNN topologies. They trade off flexibility for generality. Contrary, FINN uses an High-level Synthesis (HLS) hardware library [20] of hardware layers and components that are used to generate streaming architectures customized for each network. Other tools for automatic hardware generation are FlexCNN [21], integrating an FPGA implementation framework into Tensorflow and DNNBuilder [22], which uses software-hardware co-design to perform an end-to-end optimization of deep learning applications.

Our approach is similar to FINN. Specifically, it maps DNN on a set of highly optimized hardware components. Conceptually, the HALF framework is different from all previous approaches as it includes NAS for hardware-optimized topologies. Also, we are targeting a fully automatic solution.

There are publications on augmenting NAS with hardware awareness for FPGA. [10] proposed NAS for their accelerator capable to process CNNs layer-by-layer similar to xDNN and DPU. With respect to hardware awareness, their approach optimizes DNNs for low latency only. [8] proposed a hardware and software co-exploration framework that uses NAS for optimizing DNNs for implementation on multiple FPGAs, however without much consideration of optimizations on a level of separate FPGAs. Their approach is primarily focused on optimizing for high throughput.

Our framework is different as we use hardware-aware NAS for dataflow-style fully on-chip architectures customized for each network and optimized for various objectives, namely low latency, low power, and low energy.

III. HALF Framework

The HALF framework is comprised of two main components, which are the hardware-aware NAS and the FPGA implementation framework (see Fig. 2). The inputs are a dataset and requirements specified in terms of application-level and hardware-level constraints and optimization objectives. The output is a hardware configuration for the selected FPGA platform that fulfills the requirements.

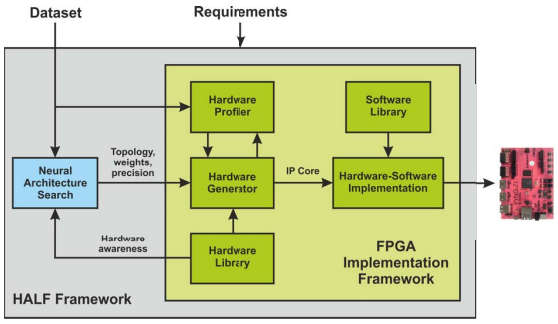

Fig. 2: Components of HALF framework

HALF generates the output automatically, and therefore it significantly accelerates the deployment of DNNs on FPGAs. The NAS takes approximately two days, depending on the complexity of the underlying search space and the dataset, while manual search would take weeks, even without considering hardware awareness. Including hardware awareness into the NAS shortcuts the otherwise time-consuming manual design and evaluation cycles of different FPGA implementations to identify candidates with the best trade-offs. The

automatic hardware generation and implementation only take a few hours, in contrast to a manual hardware design process that can take days or even weeks, especially if hardware components have to be designed from scratch.

A. Neural Architecture Search

The NAS is the first step in the framework and finds optimal topologies for the implementation framework. It is based on an evolutionary algorithm, which are very flexible, as they do not impose strong restrictions on the search space or the objective functions, especially the latter do not need to be differentiable. We use the genetic algorithm proposed by [23], which boosts the search via the concept of dormant genes.

For the selection strategy, we use a similar, bayesian-based method as [13], which explores the Pareto Frontier of DNN candidates efficiently in a two-step procedure, preselecting candidates based on computationally inexpensive objectives first. Additionally to this two-step procedure, [13] uses network morphisms to increase the throughput of fully evaluated DNNs. We do not use network morphisms, which limits the range of mutation operations and also would be a bad fit for our genetic encoding. Instead, we handle the large training workload by implementing a dynamic workload scheduler, which leverages parallel processing on high performance computing (HPC) systems.

Hardware-awareness is incorporated twofold, i.e. via the search space and the optimization objectives. The search space is constrained to layers which are included in the hardware library, thus the models from the NAS are guaranteed to be mappable to the device. This encompasses aspects like layer types and valid hyperparameter combinations, but also the quantization of the inputs, weights and feature maps. The second dimension of hardware-awareness is introduced by the optimization objectives, described in section IV. Before passing the found topology with its trained weights to the hardware implementation framework, preprocessing and tuning techniques such as batchnorm-folding are applied to further compress the model.

B. FPGA Implementation Framework

The FPGA implementation framework comprises a hardware generator, a custom hardware library, a profiler, a software library, and a hardware-software implementation step. The hardware generator produces a hardware architecture of the neural network using components from the hardware library described in Section V. Using Xilinx Vivado HLS, the framework generates an Intellectual Property (IP) core from the DNN topology, which is based on the layers of the hardware library. The model weights are also integrated into the IP core at this point because the hardware architecture uses only on-chip memory for model storage. Additionally, it instantiates interfaces for communication with external memory and FIFO buffers for connecting the elements. The hardware generator also calculates parallelization factors for each layer, which are based on the required throughput and are mainly constraint by the target platform, i.e., number of resources, available memory bandwidth, FPGA model. While the quantization of weights and activations is provided by the NAS, the quantization for the internal accumulators is found by profiling. The profiler identifies the optimal range and precision for all accumulators in the hardware and sets the bit widths accordingly. In the last hardware-software implementation step, Xilinx Vivado Design Suite is used to generate a bitstream for FPGA. The software is compiled for running on the processor cores of the board that is used to transfer input and output data to the FPGA and to control the IP core.

IV. Hardware-aware Objective Functions

We choose energy, power and latency as the hardware-aware objectives and model them as functions of the topology and FPGA parameters. The latency is defined as

$$t_{\text{total}} = \sum_{j=1}^{N} \left(n_{\text{in},j} - 1\right) \sigma_{j-1} + l_j, \tag{1}$$

where N is the number of layers, $n_{in,j}$ is the number of values to initially fill the input buffer (e.g. the kernel size in case of convolution layers), σ_{j-1} is the output rate of the previous layer in clock cycles and l_j is the latency of the layer to produce its output. Notice that $\sigma_i = \max(l_i, \sigma_{i-1})$ evaluates recursively and describes the pipelined nature of the hardware architecture. The latencies l_i depend on the layer type and hyperparameters such as strides and kernel size, but also loop unrolling factors α_i of the FPGA implementation. In section VI, the results are reported using the throughput instead of latency, which includes the contribution of data parallelism and is the batch size divided by the latency t_{total}.

We model the effective power, so that the energy can be simply described as the product of the runtime and the effective power. The total power consumption is

$$P_{\text{total}} = P_{\text{mem}} + P_{\text{board}} + P_{\text{stat}} + P_{\text{dyn}}. \tag{2}$$

P_{mem} is the power from memory transactions and mostly does not depend on the topology, because all the weights and activations are kept on the chip in our architecture. The size of the input sample, however, does influence P_{mem}, but since an input must always be read and cannot be optimized, its contribution to the power model is excluded. A model for P_{mem} can be added easily to the framework, though. P_{board} is from other peripheral components of the hardware platform, but since it cannot be influenced by topology parameters, it is not modeled. P_{stat} and P_{dyn} are the static and dynamic power consumption of the architecture, which we model in the total power with

$$P_{\text{total}} = \sum_{i=1}^{N} \alpha_i P^*_{\text{idle},i} + \alpha_i \frac{t_{a,i}}{t_{\text{total}}} P^*_{\text{calc},i}. \tag{3}$$

We assume that the power scales linearly with the loop unrolling factors α_i. $P^*_{\text{idle},i}$ and $P^*_{\text{calc},i}$ are the power consumption when the layer is idling and calculating, respectively, for an unrolling factor of one, which can be estimated from the hardware profiler of the FPGA implementation framework. $t_{a,i}$ is the time a layer is actively computing and it is the product of the total number of outputs the layer produces multiplied with the latency to produce one such output. The power can be minimized by using no unrolling (min α) and stretching out the total runtime, with compliance to latency constraints.

The total energy is the product of the effective total power with the total runtime

$$E_{\text{total}} = t_{\text{total}} P_{\text{total}} = \sum_{i=1}^{N} \alpha_i t_{\text{total}} P^*_{\text{idle},i} + \alpha_i t_{a,i} P^*_{\text{calc},i}. \quad (4)$$

Looking at Eq. (4), it appears that minimizing the α_i is the best strategy to minimize the energy. However, since high α_i reduce both t_{total} and $t_{a,i}$ superlinearly, it is high unrolling factors that reduce the total energy consumption. Also, the energy consumed by the entire platform is the product of P_{board} and t_{total}. Since P_{board} can be much larger than the other contributions to P_{total}, minimizing t_{total} is the most effective way to reduce the measurable energy consumption.

V. Hardware Library

We present an HLS hardware library of custom hardware architectures for standard 1D-CNNs, depth-wise separable 1D-CNNs and various other DNN layers and components. The hardware architectures are highly customizable, which allows the implementation of various neural topologies. The hardware library is written as a collection of C++ template functions with HLS annotations and modularity in mind to make it easily expandable by new layers. The hardware architecture is designed to be low power and ultra-low latency. Primarily, this is achieved by keeping all weights and intermediate results in on-chip memory since off-chip transfers consume more energy and introduce extra latency. External memory is only used to read input data and write results, reducing memory access to the absolute minimum. The architecture is fully pipelined, allowing all layers to operate concurrently and starting the computation as soon as the inputs are ready to reduce latency and energy consumption. The library is based on dataflow architectures, which can be easily customized for each network. The hardware modules are designed with streaming interfaces to facilitate fast design, debugging, interoperability, and ease of integration. Separate hardware modules dedicated to each layer are connected using on-chip data streams in a single top-level module called Deep Neural Network Unit (DNNU), as shown in Fig. 3. The top-level module is equipped with Direct Memory Access (DMA) components that allow access to external memory independent of any processor using AXI-Master interfaces.

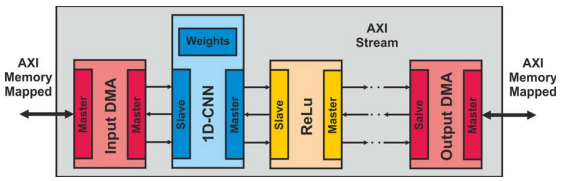

Fig. 3: Deep Neural Network Unit (DNNU).

In a pipelined architecture, there always exists a bottleneck stage, which determines the latency of the entire pipeline. The latency of the bottleneck stage can be decreased by spatial parallelism, which we refer to as unrolling (coming from loop unrolling). The hardware library is designed with parametrizable unrolling, which parallelizes the bottleneck stages efficiently. The parametrization allows coarse-grained parallelization on a level of filters for CNN layers and neurons for fully connected layers, and fine-grained parallelization on a level of dot-products, distinguishing kernel-level and input-channel parallelism.

VI. Results

The effectiveness of our methodology is evaluated on the task of a binary classification for ECG-based arrhythmia detection. The dataset, provided by the university hospital Charité in Berlin, contains 16000 samples, with 2 channels and a length of 60000 each. The dataset contains equal amounts of positive and negative samples. The task performance is measured in detection and false alarm rate, where we define hard limits of 90 % and 20 % for acceptance, respectively.

The NAS is performed on four Nvidia Titan X GPUs for 100 generations with 20 children per generation, which takes two days to finish. The search space constitutes of depth-wise separable convolutions with 60 different hyperparameter configurations and max pooling with 4 different strides. All DNNs end with a global average-pooling layer followed by a fully-connected layer. The depth of the topology is chosen by the NAS but restricted between 2 and 15 layers (final layers not included). The optimization objectives are power, energy and latency each with and without unrolling, and additionally number of parameters, detection and false alarm rate. All objectives are considered at the same time in the Pareto frontier.

A. Influence of NAS objectives on the topology

The network topology itself has a high impact on the final performance in terms of power and energy, which we show by comparing models optimized for the three different objectives of low-power with minimal parallelization (low P, min α_{NAS}), low-energy with minimal parallelization (low E, min α_{NAS}) and low-energy with maximal parallelization (low E, max α_{NAS}). The min α_{NAS} case is applied whenever the hardware resources are relatively low compared to the size of the DNN model, so full unrolling is not possible. The case (low P, max α_{NAS}) is excluded, since the objective of low power and maximum unrolling leads to unreasonable topologies, which can hardly be unrolled by design. The models optimized for high-throughput are the same as the ones for energy, thus not explicitly considered here. For the implementation strategy, we set the parallelization factor $\alpha_{\text{Impl.}}$ to either one or the maximum, while keeping other hardware-related parameters fixed.

TABLE I: Power and energy measurements using different objectives and resulting topologies.

NAS Objective	Impl. Strategy	Throughput [samples/s]	P_{total} [W]	E_{total} [µJ]
low E, max α_{NAS}	min $\alpha_{\text{Impl.}}$	$4.4{\cdot}10^3$	4.42	1010
low E, min α_{NAS}	min $\alpha_{\text{Impl.}}$	$\mathbf{5.3{\cdot}10^3}$	4.46	**841**
low P, min α_{NAS}	min $\alpha_{\text{Impl.}}$	$1.4{\cdot}10^3$	**4.40**	3120
low E, max α_{NAS}	max $\alpha_{\text{Impl.}}$	$\mathbf{4.8{\cdot}10^5}$	8.22	**1.7**
low E, min α_{NAS}	max $\alpha_{\text{Impl.}}$	$3.7{\cdot}10^5$	7.16	2.0
low P, min α_{NAS}	max $\alpha_{\text{Impl.}}$	$8.3{\cdot}10^4$	**6.10**	73.8

Table I shows that the best results in terms of energy and power are obtained if the optimization objective matches the

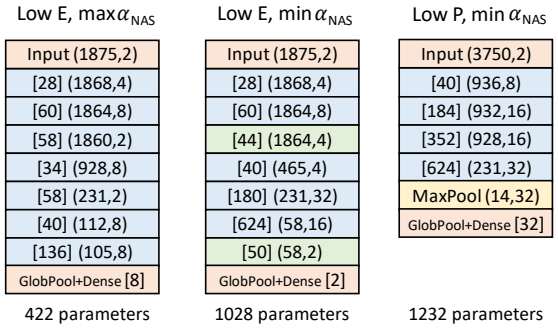

Fig. 4: Comparison of topologies for different optimization objectives. Blue: depthwise-separable convolution, green: kernel size 1. In the boxes: [#parameters], (output-width, channels)

implementation strategy. This demonstrates the effectiveness of the cross-layer optimization approach, where hardware-related parallelization factors influence the topology search, leading to better solutions. The three topologies used in Table I have meaningful differences in their structure, as shown in Fig. 4. For the first two models, both optimized for energy, the search converged to a very similar solution. Especially the first layers are identical and the position of the striding layers is the same. A parent-child relation in terms of evolution is ruled out since we selected both models from two different experiments. The key difference comes from the 3rd convolution layer, which becomes the bottleneck in the min $\alpha_{\text{Impl.}}$ case, which it is not for max $\alpha_{\text{Impl.}}$. The 2nd model uses a kernel size of one here, which lowers the latency of the entire pipeline significantly, resulting in higher energy-efficiency, although having *two times* more parameters. For the case of max $\alpha_{\text{Impl.}}$ the 1st model has lower energy consumption, because it needs less resources, thus more parallel instances can be implemented and the throughput is increased. The 3rd topology in Fig. 4 is less deep than the energy optimized models. Without unrolling, a shallower topology requires less hardware resources. The bottleneck layer is the 3rd convolution, which computes twice as long as the next slowest layer. This imposes a high idle time on the non-bottleneck layers, which results in lower power consumption, although having almost *three times* as many parameters as the 1st model.

In summary, it is not the size of the model alone, but its structure, which determines the hardware performance. The NAS is able to find significant structural features in the DNN models, based on the optimization objectives.

B. Efficiency of holistic methodology

To demonstrate the efficiency of our holistic approach, we present solutions for three different domains, namely low-power, low-energy, and high-throughput with optimizations applied on all design levels from the NAS down to the FPGA platform, see Table II. In each case, the target platform was selected according to the optimization goal, Pynq-Z1, Ultra96-V2, and ZCU102 for each domain, respectively. Additionally, we show results for the low-energy topology implemented on the Nvidia Jetson AGX Xavier embedded GPU optimized using TensorRT.

For the low-power domain, the NAS searched for a topology that exhibits the lowest power with unrolling factor constrained to one (low P, min α). In its turn, the hardware implementation framework instantiated only a single fully-folded instance of DNNU implemented with the lowest frequency that, however, still outperforms real-time requirements. Although the NAS includes separate objectives for low-energy and high-throughput, we observe that the best model for both cases is the same one, which is optimized considering the maximal unrolling factor (low E, low L, max α). The following hardware implementation step targeted different platforms but used identical strategies to achieve the highest frequency and maximally utilize the available resources by instantiating the maximal number of instances with the highest unrolling factor (α = 40). Table II demonstrates that each implementation achieves the best results in the targeted domain. The FPGA designs outperform the embedded GPU implementation regarding all shown metrics, although the GPU has a higher frequency, larger batch size, and a model optimized with TensorRT.

TABLE II: Comparison of FPGA implementations for various domains.

Device	Freq. [MHz]	Batch Size	Throughput [samples/s]	P_{total} [W]	E_{total} [J]
Pynq-Z1	0.5	1	2.1	**1.9**	$9.1 \cdot 10^{-1}$
Ultra96-V2	333	4	$4.8 \cdot 10^{5}$	8.22	$\mathbf{1.7 \cdot 10^{-5}}$
ZCU102	322	16	$\mathbf{1.6 \cdot 10^{6}}$	33.9	$2.1 \cdot 10^{-5}$
Jetson AGX	1377	1024	$7.7 \cdot 10^{4}$	21.1	$2.7 \cdot 10^{-4}$

VII. Conclusion

We present a cross-layer optimization methodology, which allows searching for DNNs optimized for hardware. The methodology is based on a hardware-aware NAS coupled with a parametrizable hardware template. We implemented this approach in an automatic framework and demonstrate its performance by comparing power and energy for DNN models optimized for different objectives. The objectives affect the structure of the generated networks meaningfully, so that the model implementations outperform each other in their respective optimization target. Additionally, we exploit the full potential of our framework by automatically applying hardware-related optimizations that further tune the model, depending on the target platform. Considering every design aspect of the hardware implementation, we target different domain and show significant differences in the hardware metrics for a real-world application scenario of atrial fibrillation detection, outperforming the Nvidia Jetson AGX in throughput, power and energy consumption.

Acknowledgement

This work has been funded by the German Federal Ministry of Education and Research as a participant of the pilot innovation competition "Energy-efficient AI System".

REFERENCES

[1] S. Kiranyaz, T. Ince, and M. Gabbouj, "Real-time patient-specific ecg classification by 1-d convolutional neural networks," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 3, pp. 664–675, 2015.

[2] P. Rajpurkar, A. Y. Hannun, M. Haghpanahi, C. Bourn, and A. Y. Ng, "Cardiologist-level arrhythmia detection with convolutional neural networks," *arXiv preprint arXiv:1707.01836*, 2017.

[3] A. Y. Hannun, P. Rajpurkar, M. Haghpanahi, G. H. Tison, C. Bourn, M. P. Turakhia, and A. Y. Ng, "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network," *Nature medicine*, vol. 25, no. 1, p. 65, 2019.

[4] M. Kachuee, S. Fazeli, and M. Sarrafzadeh, "Ecg heartbeat classification: A deep transferable representation," in *2018 IEEE International Conference on Healthcare Informatics (ICHI)*. IEEE, 2018, pp. 443–444.

[5] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[6] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, 2019, pp. 4780–4789.

[7] Y. Zhou, X. Dong, B. Akin, M. Tan, D. Peng, T. Meng, A. Yazdanbakhsh, D. Huang, R. Narayanaswami, and J. Laudon, "Rethinking co-design of neural architectures and hardware accelerators," *arXiv preprint arXiv:2102.08619*, 2021.

[8] W. Jiang, L. Yang, E. H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4805–4815, 2020.

[9] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *International Conference on Learning Representations*, 2018.

[10] H. Fan, M. Ferianc, S. Liu, Z. Que, X. Niu, and W. Luk, "Optimizing fpga-based cnn accelerator using differentiable neural architecture search," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 465–468.

[11] Y. Sun, B. Xue, M. Zhang, G. G. Yen, and J. Lv, "Automatically designing cnn architectures using the genetic algorithm for image classification," *IEEE Transactions on Cybernetics*, 2020.

[12] M. Wistuba, "Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 243–258.

[13] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," in *International Conference on Learning Representations*, 2018.

[14] C. Schorn, T. Elsken, S. Vogel, A. Runge, A. Guntoro, and G. Ascheid, "Automated design of error-resilient and hardware-efficient deep neural networks," *Neural Computing and Applications*, pp. 1–19, 2020.

[15] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 65–74.

[16] "Xilinx ml suite overview," https://www.xilinx.com/publications/events/machine-learning-live/colorado/xDNN_ML_Suite.pdf.

[17] "Accelerating dnns with xilinx alveo accelerator cards," https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf.

[18] "Dnndk user guide," https://www.xilinx.com/support/documentation/user_guides/ug1327-dnndk-user-guide.pdf.

[19] "Zynq dpu v3.2. product guide," https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf.

[20] "Hls library for hardware acceleration of quantized neural network using finn," https://github.com/Xilinx/finn-hlslib.

[21] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 133–139. [Online]. Available: https://doi.org/10.1145/3373087.3375321

[22] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[23] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the genetic and evolutionary computation conference*, 2017, pp. 497–504.