Joint Software-Hardware Design for Green AI

Author/s

***Abstract*—Green artificial intelligence (AI) models or so-called energy efficient machine learning models have gained a lot of attention in recent years. Sparse quantized deep neural networks (DNN) are considered green AI in opposition to their dense non-quantized counter parts. Green models can provide impressive performance in diverse digital signal processing tasks with a relatively low carbon footprint. There exists an array of techniques in the software domain to improve energy efficiency of such models. However, software simulation alone can often be misleading for implementing such models where resource constraints are acute such in IoT or edge devices. Therefore, there is a need for a framework that combines software and hardware implementation level optimization on power and area requirements of such models.**

In this paper, we propose a joint neural architecture search (NAS) for DNNs from python simulation to hardware-FPGA implementation. Proposed approach sequentially explores for best design in each paradigm and reaches the best implementation in terms of power and area requirements. We evaluate our method on a real-time signal-processing model and find that we can achieve 1.7x improvements in power and 40x improvements in area than the baseline implementation of the same model.

I. Introduction

Deep neural networks (DNN) have gained widespread popularity in various domains, including data analytics and signal processing. However, despite their high performance, DNN models are known to be energy-intensive [1]. For instance, the energy required to train a large DNN model for natural language processing can result in a significant carbon footprint, with an estimated 284 metric tons of CO_2 emissions, equivalent to the lifetime emissions of five cars [2]. This has led to the emergence of a new research direction called "green AI" [3]–[6], which aims to balance the tradeoff between power efficiency and inference accuracy. Green AI models have shown promise in accelerating DNN models, particularly on field-programmable gate-array (FPGA) platforms [7]–[9].

The high energy consumption of DNN models is largely attributed to their architecture. Most DNN layers comprise affine transforms, involving vector matrix multiplication and bias addition. Multiplication is a computationally expensive operation in FPGAs and is usually implemented using highly-customized digital signal processing (DSP) blocks. Moreover, the size of the training parameters, including weights and biases, is another critical factor that affects the energy consumption of DNN models. Larger parameter sets generally require more energy. To address this challenge, many green AI models have been proposed, including knowledge distillation, pruning, and quantization techniques, that aim to reduce the size of DNN models while maintaining high inference accuracy. Many green AI models are designed with either knowledge distillation, pruning, or quantization to downsize the DNN models while maintaining high inference accuracy.

A joint optimization of both hardware and software is necessary to achieve the best energy efficiency for designing the best sparse DNN. While sparse DNNs have lower computational complexities from a software perspective, the hardware resources required to implement them efficiently can be quite high. Therefore, an optimized design must take into account both the software and hardware requirements of the sparse DNN, with the aim of minimizing the energy consumption of the overall process. By jointly optimizing the hardware and software components, it is possible to achieve a balance between computational efficiency and energy consumption, resulting in a system that is both fast and energy-efficient. We propose a framework to achieve energy goal by optimizing a DNN model from Python implementation to FPGA deployment. Our key contributions are as follows:

- We present an end-to-end design methodology for generating machine learning models that are optimized for energy efficient hardware implementation.
- We propose an optimal word size for fixed precision data that leads to a compact circuitry.
- Our methodology enables the efficient implementation of sparse DNN, which can significantly reduce the computational complexity and energy consumption of the model.

Following sections are organized as follows. Section II highlights the related works, section III describes the proposed method, section IV demonstrates the efficacy of the proposed method, and section V concludes the paper with discussion.

II. Related Works

Many optimization techniques are used when building a machine-learning model in the software domain. Such optimization targets better accuracy, lighter implementation and computation overhead, smaller weight, etc, which can reduce the power needed to train and run the models. The optimization activity can be divided into two parts.

A. Model Optimization in Python

One popular approach to optimizing a DNN model is to reduce the precision of the weights and activations, reducing the amount of data that needs to be transferred and processed. This can be achieved through weight quantization [10], [11] and activation quantization techniques, such as Fixed-Point Quantization and Dynamic Fixed-Point Quantization. The Hardware-Aware Automated Quantization (HAQ) [12] framework leverages reinforcement learning to determine the quantization policy for different neural network and hardware architectures, effectively reducing latency and energy consumption with negligible loss of accuracy. [13] outlines SqueezeNet, in which a set of modifications made to the

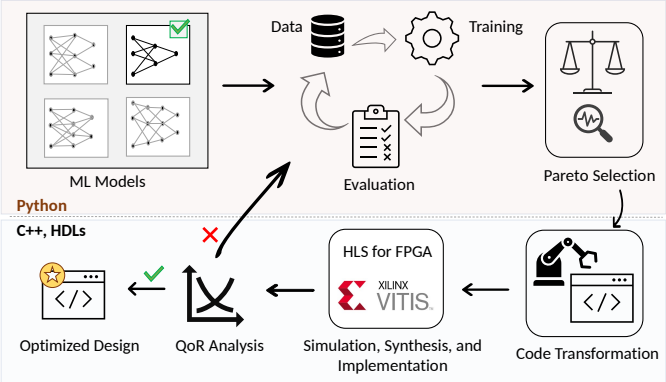


Fig. 1: Joint optimization of ML models.

network architecture to achieve energy goal, includes aggressive channel reduction, separable 3x3 convolutions, and an element-wise addition skip connection, and optimization of the architecture by simulation, but no FPGA targeted optimization is discussed.

Another approach is to compress the size of the DNN model through techniques such as pruning [14], knowledge distillation [15], and parameter sharing [16]. Pruning removes unimportant connections or filters in the network. At the same time, knowledge distillation trains a smaller network to mimic the behavior of a more extensive network.

B. Model Optimization in FPGA

Many works have explored hardware-level optimizations, such as optimizing the hardware architecture and designing dedicated hardware accelerators for DNNs. These techniques exploit the properties of DNNs and can significantly reduce the power consumption of DNNs [17].

One common approach is to use fixed-point arithmetic instead of floating-point arithmetic to represent the weights and activations in DNNs, as fixed-point arithmetic is more hardware-friendly and requires less power to perform computations [18]. Additionally, binary weights [19] restrict the weights of a neural network to only two possible values, typically -1 or 1. This allows for replacing many multiply-accumulate operations with simple additions, significantly reducing power consumption and hardware complexity.

Therefore, optimization techniques can be applied to reduce the power consumption of DNNs implemented on FPGAs, ranging from software-level techniques such as quantization and model compression to hardware-level techniques such as pipelining and parallelization and DVFS have been explored from different perspectives. We optimize both in software and hardware domains, explicitly targeting power efficiency.

III. Proposed Method

A general overview of the proposed approach is given below. The framework consists of python simulation, C++ simulation, FPGA synthesis, and implementation for finding the quality of results.

A. Python Simulation

This step consists of model selection, hyperparameter selection, and model architecture selection. Once a suitable machine learning model is selected, the input output-specific adjustment is performed. Some applications might need big kernels, while others might be doing well if more hidden layers could be added. Various quantization and pruning techniques are applied in this step to maximize the performance. Some critical factors could be the number of epochs, non-zero parameters, loss, etc. Some optimization happens in this phase includes:

Weight Quantization: This technique involves reducing the precision of the weights in the neural network, which reduces the amount of memory and computation required during inference.

Pruning: This technique removes weights from the neural network that have little effect on the output, reducing the number of computations required during inference.

Weights are also stored in this step. Trained models or weights are the outcomes of this step.

B. Pareto Selection

In the context of deep neural networks (DNNs), the complexity of a model is closely related to the number of non-zero parameters (nnz), which in turn is determined by the number of hidden layers, the size of the kernel, and the size of the channels. In order to optimize DNNs for efficient hardware implementation, quantization of weights has been widely used as a technique for reducing computational complexity and memory requirements. Specifically, we consider three quantization schemes for the weights: 1. No quantization, 2. Power of Two (PoT) quantization, and 3. Additive PoT (APoT) quantization. To achieve the primary optimization objective of power efficiency, we employ a Pareto optimization approach that selects the most power-efficient solutions based on their trade-offs between accuracy and nnz. This involves identifying the Pareto front, which represents the optimal trade-off curve between accuracy and nnz, and selecting the Pareto-optimal solutions that provide the best performance for a given power budget.

Step 1: Generate Pareto solutions for each quantization scheme for different kernel sizes by plotting nmse (dB) against nnz. Select the set of solutions with the best nnz for nmse optimization.

Step 2: Select the most optimized solutions for synthesis by comparing the three sets of Pareto solutions generated by the three quantization schemes and refining them to generate a final Pareto front.

It is possible that some solutions only varies on one axis or, in some case, even overlap. In that case, the most promising solution selection can be made using the following rules.

1) Let S be a set of solutions and let nnz(s) denote the number of non-zero elements in solution s. Among all solutions in S, select s* that satisfies the condition:

$$s^* = \arg\min\{nnz(s) : s \in S\}.$$

Therefore, if solutions vary on nnz but nmse is constant, pick one with the lowest nnz.

2) Let S be a set of solutions and let nmse(s) denote the normalized mean squared error of solution s. Among all solutions in S, select s* that satisfies the condition:

$$s^* = \arg\min\{nmse(s) : s \in S\}$$

If solutions vary on nmse, pick one that has the lowest nmse.

3) Let S be a set of solutions that overlap with different quantization types and let P(s) denote the power consumption of solution s. Among all solutions in S, select s* that satisfies the condition:

$$s^* = \arg\min\{P(s) : s \in S\}$$

If multiple solutions overlap with different quantization types, the most power-efficient quantization will be selected.

4) Let S be a set of solutions that coincide with the same quantization type and let C(s) denote the number of channels used in solution s. Among all solutions in S, select s* that satisfies the condition:

$$s^* = \arg\min\{C(s) : s \in S\}$$

If more than one solution overlaps with the same quantization, then the solution with the lowest number of channels will be selected.

5) Let S be a set of solutions that overlap with the same quantization type and the number of channels and let H(s) denote the number of hidden layers in solution s. Among all solutions in S, select s* that satisfies the condition:

$$s^* = \arg\min\{H(s) : s \in S\}$$

If multiple solutions overlap with the same quantization and the same number of channels, then the solution with the lowest number of hidden will be selected.

The aforementioned set of five rules will be henceforth referred to as "exclusion rules", with subsequent sections referring to them by their assigned rule number as appropriate.

C. HLS Conversion, Simulation and Synthesis

The process of transforming machine learning models from Python implementation to HDL code using the Vitis [20] software suite is explained in this section. This includes converting CNN layers, activation functions, and multi-input multi-output convolutions (FIR) to C++ code while retaining the optimizations made in Python.

To ensure the correctness of the conversion process, csimulation needs to be performed on the converted C++ code. Furthermore, performance and efficiency optimization of the implemented FIR function can be done by creating three different versions with regular multiplication, PoT (Power-of-Two), and APoT (Additive Power-of-Two) techniques.

High-level synthesis (HLS) techniques can facilitate rapid prototyping of hardware designs, including the corresponding CNN layers, activation functions, and convolution operations. Conversion of Python to HLS enables exploration of more compact and efficient designs, such as the use of specific word length for optimal area usage. Synthesis specific results obtained in this phase could help find the optimal design for implementation.

D. HDL Implementation

In the Vitis HLS tool, the implementation step plays a crucial role in transforming the high-level C++ design into an optimized hardware design that meets the specific constraints and requirements of the target FPGA device. This process accurately estimates the resources required for the design, including the number of logic cells, DSP blocks, memory blocks, and other FPGA resources. This step is critical for ensuring that the design meets power constraints. The implementation step generates the HDL code, which can be used to program the FPGA to implement the hardware design. Depending on the user's preference, the HDL code can be VHDL or Verilog. Close to reality power estimation can be obtained in this step.

E. QoR Analysis

QoR (Quality of Results) is a metric that evaluates the overall quality of a design in terms of factors such as performance, power consumption, area utilization, and timing. The purpose of QoR is to enable designers to compare and select the most optimal design among a set of design alternatives. To perform a comparative analysis of the quality of results (QoR) in FPGA design space exploration, various methods can be used, including:

1) Timing Analysis: It involves analyzing the timing constraints of the design and determining if they are met. The timing information includes delay, clock frequency, and slack. The timing report generated after the synthesis process provides insight into how well the design meets the timing requirements.
2) Resource Utilization Analysis: It involves analyzing the FPGA resources used by design, such as the number of logic cells, memory blocks, DSPs, and I/O pins. Resource utilization analysis helps ensure the design fits within the target FPGA device's capacity.
3) Power Analysis: Power analysis involves measuring the design's power consumption. Power analysis can be performed at different stages of the design flow, including simulation, synthesis, and implementation. It helps to optimize the design for low power consumption.

By performing the above analysis artifacts, we can evaluate the QoR of the design and make informed decisions on modifications and optimizations to reach the ultimate golden design. This work considers resource utilization and power consumption for design optimization.

IV. Experiment and Analysis

While the proposed method could be applied to any DNN, we demonstrate joint optimization for a real-world low latency high throughput CNN network. All of our experiments are

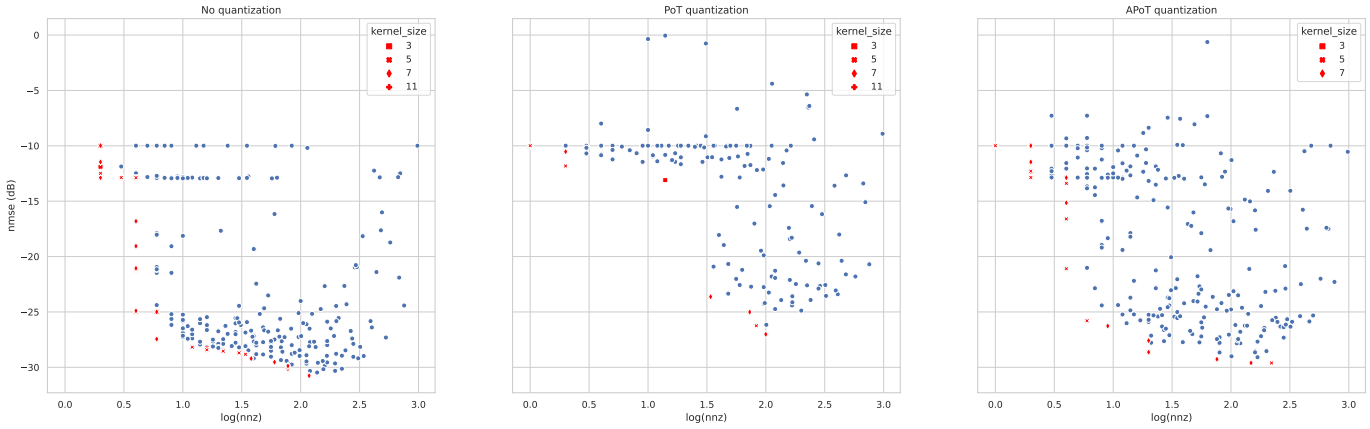


Fig. 2: Pareto solutions for each quantization type.

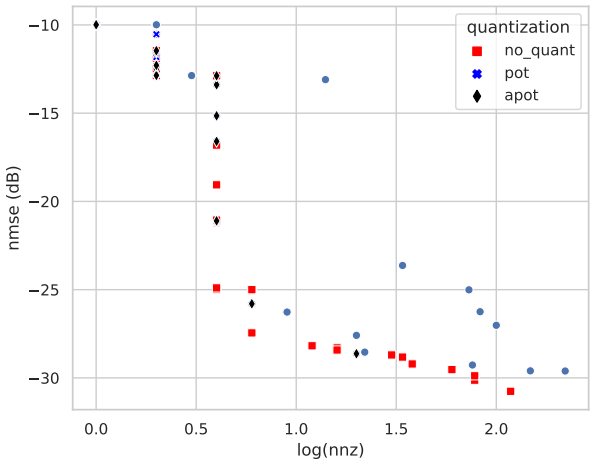


Fig. 3: Pareto front from the Pareto solutions of each quantization type.

conducted on a machine with Intel(R) Core(TM) i7-8700K CPU 3.70GHz, 64GB of main memory configuration, and Ubuntu 20.04.5 LTS as the operating system. The Target FPGA synthesis board is Xilinx ZCU104. The kernels are synthesized using Vitis HLS 2022.1.

A. Proposed CNN Model

We consider a 1D CNN-based Digital pre-distortion (DPD) model for joint optimization. DPD is used in digital communication systems to mitigate distortions introduced by nonlinearities in power amplifiers (PA). Such nonlinearities in power amplifiers can cause intermodulation distortion, spectral regrowth, and amplitude modulation. DPD typically requires a high-speed signal processing system like FPGA to implement complex mathematical algorithms.

In recent years, there has been increasing interest in using DNNs [21], [22] for DPD, as they have shown promising results in mitigating nonlinear distortion in PAs. DNN-based DPD works by training a neural network to learn the nonlinear behavior of the PA using a large set of input/output signal pairs. Once the neural network is trained, it can be used to pre-distort the input signal before it is amplified by the PA, effectively canceling out the nonlinear distortions introduced by the amplifier. Our test bed is a 1D CNN-based DPD system. The input signal is first segmented into small time intervals, or "windows," or kernel size, and each window is treated as a separate input signal. There are two input and two output channels in the DNN for NN.

1D CNN-based DPD has several advantages over other DPD methods: 1D CNNs can capture temporal dependencies in the input signal, which can be essential for capturing the nonlinear behavior of power amplifiers. 1D CNN-based DPD can be trained using relatively small training data, making it more practical for real-world applications. 1D CNN-based DPD can be implemented using relatively simple hardware, making it more cost-effective than other DPD methods. The proposed CNN comes with a set of adjustable network configuration parameters. The adjustable parameters we manipulate: are kernel size, quantization type, no. of hidden channels, no. of hidden layers, and percentage of weights to be pruned.

B. Optimization in Python

This experiment aims to apply various DNN optimization techniques to a CNN model coded in Python and collect data on the resulting quality of results (QoR) for different configurations. The optimizations considered include kernel size (four different values: 3, 5, 7, and 11.), quantization type (no quantization, PoT, APoT), number of hidden layers (1 or 2), number of hidden channels (2, 4, 6, 8, 10, 14, 16), and pruning percentage (0, 30, 65, 83, 91, 95, 98, 99).

We sweep through all the different configurations to collect the performance data, resulting in 840 samples. We accumulate two metrics for each instance: normalized mean squared error (nmse) and the number of non-zero weights (nnz). The nmse measures the model's accuracy, while the nnz measures the model's sparsity, indicating its computational efficiency.

We can analyze the impact of each optimization technique on the resulting QoR for different configurations. We can

TABLE I: Candidate Pareto Solutions. Kernel refers to kernel size. Pruning (prun.) is a percentage.

kernel	quant.	hid_ch	hid_layer	prun.	nmse	nnz
7	no_quant	4	1	30	-30.13	78
		8	1	65	-29.88	78
	apot	14	1	95	-28.63	20
5	no_quant	10	1	83	-28.82	34
	apot	8	1	95	-25.80	6
		4	1	95	-21.17	4
3	no_quant	8	1	83	-28.30	16
	apot	4	1	99	-12.84	2

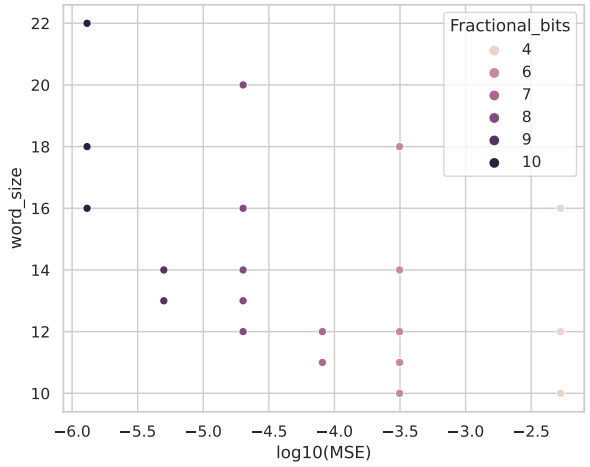


Fig. 4: Word size affects the accuracy in a fixed precision scheme.

then use this information to select Pareto solutions for each quantization type.

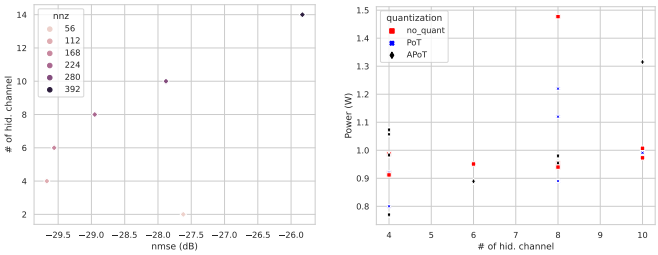
C. Pareto Solution Selection

After optimizing the Python code, the search for Pareto optimal solutions was carried out for three types of quantization. Following step 1, 216 Pareto optimal solutions were identified, as depicted in Fig. 2. However, this is a large search space if considering the synthesis and implementation time required in Vitis HLS. Thus, we reduce the number of solutions to 112 by following step 2, as illustrated in Fig. 3. We then apply exclusion rules further to narrow down the set of Pareto optimal solutions, resulting in the top 50 solutions, 8 of which are presented in Table I.

D. Optimization in HLS

The following parts of this section describe optimization that happens in the FPGA/HLS domain. Optimization in HLS occurs in a couple of aspects.

Optimal word_size: The use of fixed precision representation in FPGA provides several benefits, including a more compact area requirement, reduced power consumption, and increased speed of operations. Compared to arbitrary precision, fixed precision representation can reduce the number of



(a) nmse (b) Power

Fig. 5: Effect of Number of Channels on Model Performance, Sparsity, and Power Consumption

bits required to represent a number, significantly reducing the overall size of the circuit. This size reduction can result in lower power consumption and faster operations due to the reduced number of operations required. Additionally, fixed precision allows for more control over the precision levels and range of values, which can be optimized for specific application requirements.

The convolution operation is essential to the convolutional neural network (CNN). To optimize the implementation of CNN on FPGAs, fixed precision representation of floating point numbers is supported by synthesis tools. We compared the convolution outcomes for a range of inputs and arbitrary precision points to determine the optimal word size for fixed precision representation in CNN implementation. The results of our experiments, shown in Fig. 4, indicate that a word size of 16 or <16,6>, where the integer part is 6 and the floating part is 10, provides the best outcome in terms of mean squared error (MSE).

Pragma Insertion: Carefully using pragmas in High-Level Synthesis (HLS) can yield significant benefits. Pragmas can reduce power consumption by minimizing the number of operations and data movements in the design. Furthermore, pragmas can optimize memory access and utilization by controlling data placement and buffering in the generated hardware. As a result, HLS synthesis utilizes pragmas for loop unrolling, data pipelining, and other optimizations that reduce the usage of Digital Signal Processors (DSPs) blocks and other resources. We insert pragma in C++ converted CNN as an HLS optimization task.

Synthesis and Implementation: Once the optimal word size and quantization type are known for a given neural network, network-specific parameters: kernel size, number of channels, number of layers, pruning percentage, and quantization type needs to be identified from the Pareto solutions. Then we can generate C++ code and synthesize it in Vitis HLS to obtain the estimated FPGA footprint. The footprint includes important metrics such as the number of DSPs, LUTs, FFs, and latency for each design. From the QoR report, we can analyze the design's power consumption, area usage, and sparsity. Fig. 5 (a) and (b) show that the number of hidden channels directly affects the number of non-zero values (nnz)

TABLE II: Synthesis and implementation profile. CNN model as kernel_hidden layers_hidden channels_pruning percentage

CNN Model	quant.	DSP	FF	LUT	Power (W)
5_1_4_95	apot	**0**	**36831**	**439933**	**0.77**
5_1_4_0	not_quant	40	82286	557820	1.2
7_1_4_30	no_quant	**38**	32901	321813	**1.07**
7_1_4_0	no_quant	50	17050	71555	1.1
3_1_4_99	apot	**0**	**77040**	**711369**	**0.95**
3_1_4_0	no_quant	25	152828	990968	1.395

and power consumption. Additionally, the number of DSPs, FFs, and LUTs is a function of the nnz. Based on these observations, we can select the optimal hidden channel size 4. Now we have 3 Pareto solutions with a hidden channel size of 4. These three solution goes for further implementation. The synthesis and implementation-specific profiles of these three solutions are shown in Table II.

V. Discussion & Conclusion

Discussion

From the literature, it is intuitive that PoT implementation of the kernel would be more energy efficient, but in reality, we find that apot yields better nmse and power consumption for a certain dataset and DNN. Our implementation of the convolution operation is zero aware. Therefore, the higher power requirement of pot kernel operation could be explained by the fact that pot implementation circuitry spends more energy to determine if the value of weight is non-zero. Table II shows the optimized designs against their non-optimized counterparts. In every row, the number of DSPs in the optimized CNN has been reduced because of the careful use of pragma. Optimization techniques worked better in apot quantizations but not in LUT and FF optimization in no-quantizations. The pots are absent from the top Pareto solutions as we prioritized nmse over nnz. The pots have a higher chance of being on the Pareto fronts if we prioritize nnz.

Conclusion

The proposed framework addresses the challenge of hardware-software optimization of DNNs by presenting an end-to-end design methodology for generating machine learning models that are optimized for hardware implementation. By jointly optimizing the hardware and software components, it is possible to achieve a balance between computational efficiency and energy consumption, resulting in a system that is both fast and energy-efficient. The contributions of the framework include saving critical circuit resources and efforts spent in discovering multiple designs for rapid hardware prototyping and enabling the efficient implementation of sparse DNNs.

References

[1] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

[2] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, July 2019. Association for Computational Linguistics.

[3] Jingjing Xu, Wangchunshu Zhou, Zhiyi Fu, Hao Zhou, and Lei Li. A survey on green deep learning. *arXiv preprint arXiv:2111.05193*, 2021.

[4] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[5] Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *arXiv preprint arXiv:2106.08962*, 2021.

[6] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[7] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[8] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *arXiv preprint arXiv:1803.05900*, 2018.

[9] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of FPGA-based neural network accelerator. *arXiv preprint arXiv:1712.08934*, 2017.

[10] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[11] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. Deepshift: Towards multiplication-less neural networks. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2359–2368, 2021.

[12] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8604–8612, 2019.

[13] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. Squeezenext: Hardware-aware neural network design. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[14] Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[16] Pedro Savarese and Michael Maire. Learning implicitly recurrent cnns through parameter sharing. *arXiv preprint arXiv:1902.09701*, 2019.

[17] Ran Wu, Xinmin Guo, Jian Du, and Junbao Li. Accelerating neural network inference on fpga-based platforms—a survey. *Electronics*, 10(9), 2021.

[18] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 451–461, Red Hook, NY, USA, 2018. Curran Associates Inc.

[19] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[20] Xilinx. Vitis high-level synthesis user guide (ug1399). https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas, February 2022.

[21] Meenakshi Rawat and Fadhel M. Ghannouchi. A mutual distortion and impairment compensator for wideband direct-conversion transmitters using neural networks. *IEEE Transactions on Broadcasting*, 58(2):168–177, 2012.

[22] Masaaki Tanio, Naoto Ishii, and Norifumi Kamiya. Efficient digital predistortion using sparse neural network. *IEEE Access*, 8:117841–117852, 2020.