Application Aware, Life-Cycle Oriented
Model-Hardware Co-Design Framework
for Sustainable, Energy Efficient ML Systems

Hardware accelerator architecture report

Deliverable D2.1

WP2 - Hardware Architectures for Low Power ML

This project has received funding from the European Union's Horizon Europe research and innovation programme (HORIZON-CL4-2021-HUMAN-01) under grant agreement No 101070408

Project

Title:	SustainML: Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems
Acronym:	SustainML
Coordinator:	eProsima
Grant agreement ID:	101070408
Call:	HORIZON-CL4-2021-HUMAN-01
Program:	Horizon Europe
Start:	01 October 2022
Duration:	36 months
Website:	https://sustainml.eu
E-mail:	sustainml@eprosima.com
Consortium:	**eProsima (EPROS)**, Spain
	DFKI, Germany
	Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau (RPTU), Germany
	University of Copenhagen (KU), Denmark
	National Institute for Research in Digital Science and Technology (INRIA), France
	IBM Research GmbH, Switzerland
	UPMEM, France

Deliverable

Number:	**D2.1**
Title:	**Hardware accelerator architecture report**
Month:	12
Work Package:	WP2 - Hardware Architectures for Low Power ML
Work Package leader:	RPTU
Deliverable leader:	RPTU
Deliverable type:	R, DEM
Dissemination level:	Public (PU)
Date of submission:	2023-09-28
Version:	v1.0
Status:	Finished

Version history

Version	Date	Responsible	Author/Reviewer	Comments
v1.0	28-09-2023	RPTU	RPTU	

Executive summary

SustainML project aims to develop a design framework and an associated toolkit, so-called SustainML, that will foster energy efficiency throughout the whole life-cycle of Machine Learning (ML) applications: from the design and exploration phase that includes exploratory iterations of training, testing and optimizing different system versions through the final training of the production systems (which often involves huge amounts of data, computation and epochs) and (where appropriate) continuous online re-training during deployment for the inference process. The framework will optimize the ML solutions based on the application tasks, across levels from hardware to model architecture. It will also collect both previously scattered efficiency-oriented research, as well as novel Green-AI methods. Artificial Intelligence (AI) developers from all experience levels can make use of the framework through its emphasis on human-centric interactive transparent design and functional knowledge cores, instead of the common blackbox and fully automated optimization approaches.

This report corresponds to *Deliverable D2.1 - Hardware accelerator architecture report* of the SustainML project. This deliverable covers the detailed analysis of suitable cross-layer optimizations used to design low-power and energy-efficient hardware accelerator architectures.

Contents

Executive Summary **3**

Contents **4**

Acronyms **5**

1 Introduction **6**

2 Optimizations **7**
2.1 Hardware-Aware Deep Neural Networks 7
2.2 Implementation Platforms 9
2.3 Co-design and Automation 12

3 Methodology **12**

4 Hardware library **13**

5 Custom data type **15**

6 Framework **16**

7 Summary **18**

References **20**

© 2023 SustainML | HORIZON-CL4-2021-HUMAN-01 | 101070408

Acronyms

1D-CNN	One-dimensional Convolutional Neural Network.
AI	Artificial Intelligence.
ASIC	Application-Specific Integrated Circuit.
BRAM	Block RAM.
CLB	Configurable Logic Block.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
DARTS	Differentiable ARchiTecture Search.
DMA	Direct Memory Access.
DNN	Deep Neural Network.
DNNU	Deep Neural Network Unit.
DRAM	Dynamic Random Access Memory.
DSP	Digital Signal Processing.
FF	Flip-Flops.
FIFO	First In First Out memory.
FP32	floating-point 32-bit.
FP8	floating-point 8-bit.
FPGA	Field-Programmable Gate Array.
GPU	Graphics Processing Unit.
HALF	Holistic Auto machine Learning for FPGAs.
HDL	Hardware Description Language.
HLS	High-level Synthesis.
IP	Intellectual Property.
LUT	LookUp Table.
ML	Machine Learning.
NAS	Neural Architecture Search.
NN	Neural Network.
P&R	place and route.
PL	Programmable Logic.
PS	Processing System.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
TPU	Tensor Processing Unit.

1 Introduction

In the following, we are focusing on the efficient hardware implementation of Deep Neural Networks (DNNs) as one of the most challenging classes of machine learning algorithms. Efficient implementation of DNNs in hardware requires rigorous exploration of the design space on different layers of abstraction, including *algorithmic*, *architectural*, and *platform* layers as depicted in Fig. 1. First, the *application* formulates a problem by a dataset and requirements in terms of *constraints* and *objectives*, e.g., minimum accuracy and maximum latency. At the highest level of 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.

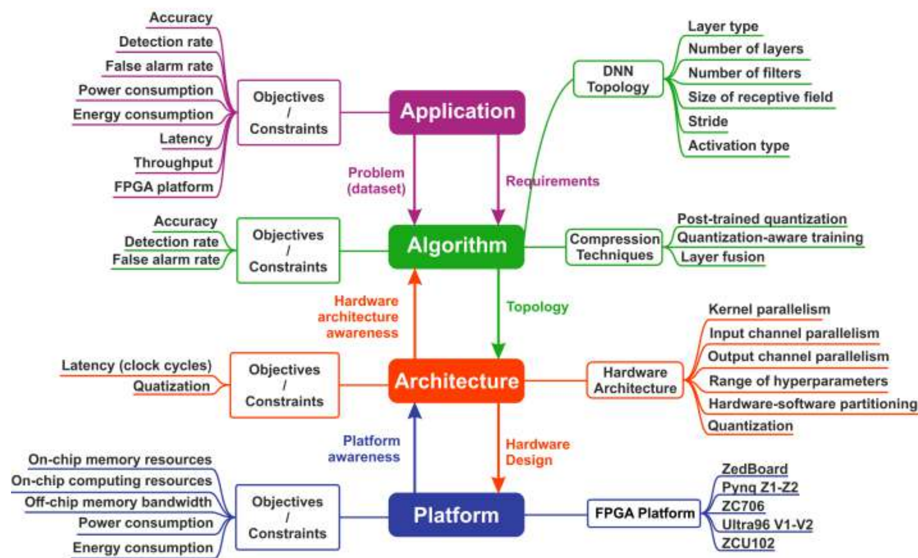

Figure 1: Design hierarchy and multi-layer design space.

All design layers are defined by a large number of parameters, which have a mutual effect on other layers. Top-down and bottom-up interdependencies express these effects. The former ones are provided by *topology* and *hardware design*. The latter ones are facilitated by *hardware awareness*, namely *platform awareness* and *hardware architecture awareness*, which model how design choices influence the physical implementation on the *platform* layer. According to our methodology proposed in [1], the architectural layer is represented by highly parametrizable architecture templates, which are used to instantiate various DNN topologies in hardware. The correspondence between the DNN topologies and architectural templates allows linking topology choices on the algorithm layer to effects on the platform layer. This way, we can model hardware characteristics, e.g., latency and power consumption, by formulas derived from the architecture templates expressed in terms of topology hyperparameters to include hardware awareness to the algorithmic layer. Neural Architecture Search (NAS) is a technique for automating the design of neural networks that can be on par with or outperform state-of-the-art hand-designed models. In the proposed methodology, we use NAS on the algorithmic layer and augment it with the hardware-awareness models. The NAS performs a cross-layer optimization since it has all information starting from the application layer down to the platform layer.

In summary, the large design space is explored by the NAS guided by optimization objectives targeting both application and hardware requirements, which, however, raises a question of which cross-layer

optimizations applied on one layer of design abstraction can be efficiently facilitated on another layer enabling more efficient hardware implementation resulting in lower power consumption and higher energy efficiency.

2 Optimizations

In the following section, we explore various cross-layer optimizations, which enable more efficient hardware implementation resulting in lower power consumption and higher energy efficiency.

2.1 Hardware-Aware Deep Neural Networks

DNNs have to be designed to achieve the required accuracy and fulfill hardware criteria, especially on edge/embedded devices because they have small memory, constrained computing capabilities, memory bandwidth, power, and energy budgets.

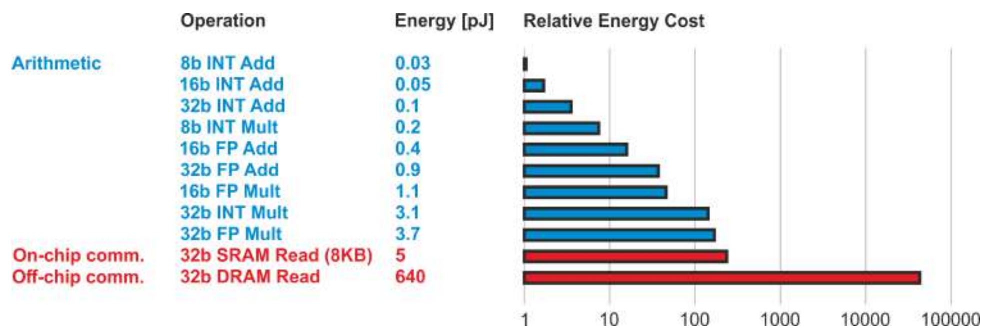

Figure 2: Energy costs for various operations (45nm 0.9 V). Based on [2].

Figure 2 compares operations considering their energy costs. It shows that arithmetic operations are "cheap" while memory accesses are "expensive" and have a cost that is a function of the size of the memory being accessed. The relative energy cost should be used as the main guidance for designing DNNs. The hardware-aware DNNs have to be small to fit into on-chip memory ideally, or to mitigate any communication with the external memory, they have to use fewer operations and leverage low-precision data types.

In summary, DNNs have to be not only accurate but also hardware-aware, meaning that they have to be designed considering a trade-off between accuracy and implementation cost.

There is a lot of ongoing research on developing networks with lower computation costs and storage consumption without impairing classification accuracy. Comparing different architectures in Fig. 3, it can be seen that some DNNs have a smaller model and require to perform fewer operations while achieving higher classification accuracy. For example, an architecture, like MobileNetV2 (14MB) [3] performs about as well as VGG-16 (552MB) model [4], despite being nearly 40 times smaller.

The efficient models became possible due to multiple macro- and micro-architectural improvements of the models. The types of layers and their arrangement are referred to as macro-architecture. The efficient micro-architectural approaches are: *a*) very deep models are replaced with fewer layers, but with more channels, *b*) activation feature maps are kept smaller, *c*) models are enhanced with skip and residual connections that have been proven to improve accuracy, *d*) standard convolutions are replaced with depth-wise separable ones. The micro-architecture also defines methods applied to individual layers, like replacing big convolutional kernels with smaller ones and fusing different layers.

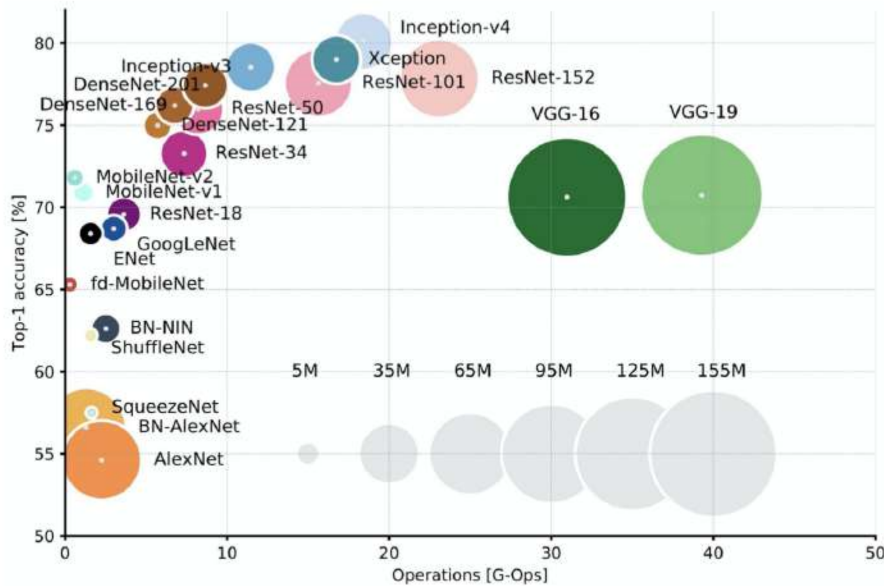

Figure 3: Top-1 accuracy on the ImageNet challenge versus number of parameters and amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters. Source [5].

Further techniques have been proposed to alleviate the computing and storage challenges. Among the most common ones are distillation [6], pruning [7, 8, 9], and quantization or a combination thereof [10]. All of them are based on the typically inherent redundancy contained within the Neural Networks (NNs), meaning that the number of parameters and precision of operations can be significantly reduced without affecting accuracy.

Knowledge distillation uses a larger "teacher" model to train a smaller "student" model to produce similar predictions while using fewer parameters. Polino et al. [11] were able to achieve a 46× reduction in size for ResNet models trained on CIFAR10 with only 10% loss of accuracy, and a 2× reduction in the size of ImageNet with only a 2% loss of accuracy using quantization and distillation. The combination of pruning, quantization and Hoffman coding allowed Han et al. [10] to reduce the storage required by AlexNet by 35×, and VGG-16 by 49×, without loss of accuracy in both cases.

The goal of the pruning is to optimize the model by eliminating less significant weights and neurons without affecting accuracy as depicted in Fig. 4. The most common weight pruning leverages the redundancy in the number of weights in the network. We can eliminate some connections that result in a sparse network. If you could rank the weights in the network according to how much they contribute, you could then remove the low-ranking weights from the network, resulting in a smaller and faster network without impairing classification accuracy.

The idea of quantization is to represent each weight in a network with only a few bits instead of a 32-bit floating-point number that is the most typical for general-purpose platforms, like Central Processing Units (CPUs) and Graphics Processing Units (GPUs). Weight quantization leverages the redundancy in the number of bit representations of weights. Quantization can drastically reduce the model size, which is crucial to fit the model into small on-chip memory. If we further quantize the activations, efficient hardware implementations become possible. As a result, cumbersome float operations can be replaced by cheaper procedures, which make inference faster and more energy-efficient. The quantization

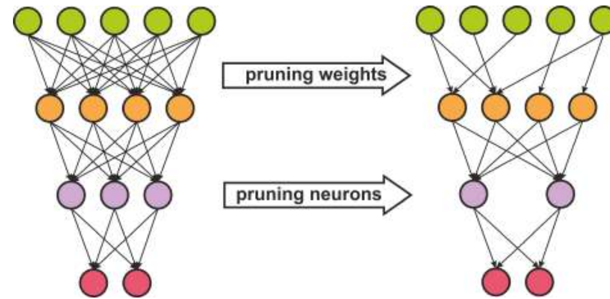

Figure 4: Effects of pruning in a fully connected Neural Network.

can be applied during training or after training. The latter one is called the post-training quantization approach. It does not involve training the quantized model, nor it requires the availability of the full dataset. However, post-training quantization is less efficient than quantization-aware training that allows for achieving higher classification accuracy using fewer bits [12].

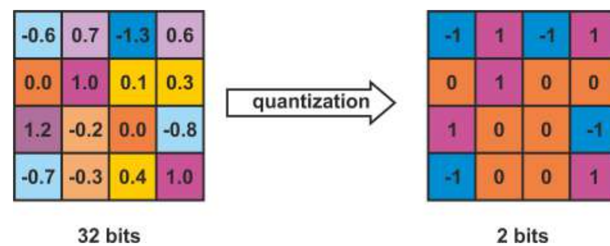

Figure 5: Quantization of the weights.

The resulting model after knowledge distillation does not require special treatment during inference, unlike after pruning and quantization. Pruning results in fewer multiplications and reduced storage consumption because of fewer parameters; however, at the cost of irregular parallelism, more sophisticated hardware, and more complicated sparse model storage representation. Quantization saves memory, and at the same time, reduces computation cost because of low-precision multiplication without the drawbacks of pruning. However, not all computing platforms support arbitrary precision and sparse matrix operations equally well.

In summary, pruning and quantization are the primary model compression techniques, but they require special treatment that raises the question of selecting an implementation platform that can fully benefit from them.

2.2 Implementation Platforms

Most of the computing today, including computing at the edge, happens on general-purpose programmable processors or CPUs. They became so widespread due to the ease of programming that hinges on executing a sequence of simple instructions. However, the energy overhead associated with fetching and decoding an instruction can be up to four orders of magnitude higher than the energy required to perform a simple logical or arithmetic operation [13] as illustrated in Fig. 6. To provide the required performance and energy efficiency, we must consider alternative architectures with lower overhead, such as domain-specific accelerators. They are hardware computing engines that are specialized for a particular domain. In the following, we will focus on the advantages and disadvantages of the computing platforms with respect to effectiveness for DNNs.

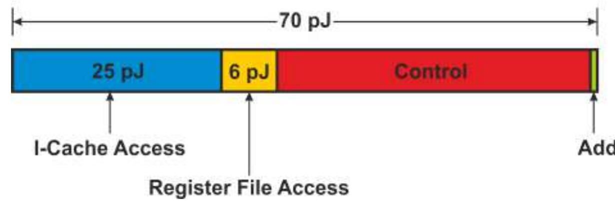

Figure 6: Instruction energy breakdown (45nm 0.9 V). Based on [2].

Comparing different platforms we can refer to Fig. 7. From left to right in the figure, we have different ways to implement an algorithm: CPU, GPU, Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuits (ASICs). We can see that these options present a trade-off between flexibility and efficiency (performance/W).

Figure 7: Flexibility vs. Efficiency for different computing platforms. Source Microsoft.

CPU as the most general-purpose computing platform, is incredibly flexible and capable of executing a great variety of singular tasks at a very high speed. GPU is less flexible than a CPU and requires a bit more specialized programming. However, modern GPUs contain thousands of cores capable of a very high degree of parallelism for regular and independent (do not rely on each other) tasks. Pruning results in the sparse matrices exhibiting irregular parallelism that is less efficient than operations on dense matrices when implemented on CPUs and especially GPUs. Further, general-purpose computing platforms support only a limited set of precisions: double, single, and recently half-precision and integer operations down to 8 bits. Meaning, intermediate precisions cannot fully benefit from CPUs and GPUs.

The domain-specific accelerators can be built on FPGA and ASIC. FPGAs are less flexible than GPUs and CPUs, and require even more specialized programming skills, but are highly efficient and can be reprogrammed with new code as requirements change. ASICs are the most efficient, they are also inexpensive to manufacture at scale, but the development process is cost-prohibitive and lengthy. The major disadvantage is that the functionality implemented in silicon cannot be changed after manufacturing. Pruning and quantization are advantageous for all platforms including CPUs and GPUs [14] but only ASICs and FPGAs can fully benefit from them. The main reasons why FPGAs and ASICs are highly efficient in comparison to general-purpose computing platforms are:

Optimized memory. Customized memory hierarchy allows us to achieve very high on-chip memory bandwidth and to utilize off-chip memory bandwidth more efficiently. For example, when weights and biases are stored in many small local memories, they achieve very high memory bandwidth with low latency and energy consumption. Off-chip memory access patterns can be optimized to utilize the available bandwidth more efficiently.

Data specialization. Support for custom precision data types increases the effective size and bandwidth of local memory, allows the external memory bandwidth to be utilized more efficiently, and enables efficient arithmetic and logic operations as most operations do not need 32-bit precision which is common

on general-purpose computing platforms.

Massive Parallelism. Multiple levels of parallelism can be implemented on domain-specific accelerators without an overhead. For example, parallelism can be applied on a level of neurons and at the same time on a dot-product level without a synchronization overhead. Furthermore, FPGA and ASIC support bit-level parallelism meaning fine-grained parallelism with down to 1 bit granularity. The degree of parallelism is only practically limited by the available resources.

Reduced overhead. Specialized hardware mitigates the overhead of instruction interpretation. Custom pipelined datapath allows implementation of complex dataflow and control flow efficiently. For example, recurrent dataflow of Recurrent Neural Networks (RNNs) and pruning can be efficiently implemented without an overhead associated with instruction fetch and decode, branch prediction, and speculation used by modern out-of-order processors.

Algorithm-architecture co-design. For general-purpose platforms, the hardware architecture is fixed. In contrast, the algorithm and the hardware architecture can be jointly optimized to meet certain requirements when implemented on FPGA and ASIC.

Using these advantages, one can use FPGAs to provide ad hoc solutions to facilitate computationally intensive, time-critical tasks at low-power consumption in a reprogrammable manner, unlike ASICs. DNNs relying on trained network parameters can require reprogramming the weights as a result machine learning benefits from the reconfigurability of FPGAs. For example, Microsoft's machine learning architecture, called Project Brainwave [15], is instantiated using FPGAs [16] achieving competitive performance with Google's Tensor Processing Unit (TPU) [17]. TPU is Google's hardware approach to machine learning implemented on an ASIC and used for many of Google's most popular services, including Search, Street View, Translate, and more.

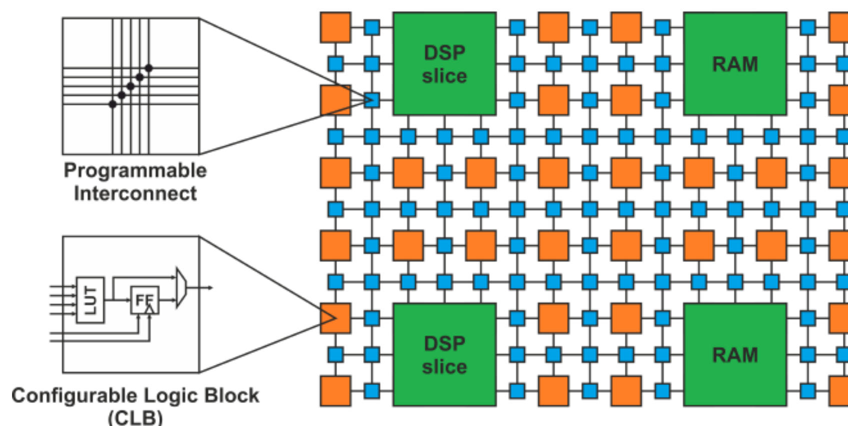

Figure 8: General FPGA structure.

FPGAs are semiconductor devices that are based on a matrix of Configurable Logic Blocks (CLBs) connected via programmable interconnects. Fig. 8 illustrates a general structure of FPGA. The CLBs include LookUp Tables (LUTs) that can be configured to implement arbitrary combinational functions, and Flip-Flopss (FFs) that are memory elements, which are used to implement sequential logic circuits. The other resources on FPGA are Block RAM (BRAM) which is distributed memory in close proximity to CLBs and Digital Signal Processing (DSP) slices to implement signal processing functions.

In summary, FPGA is a computing platform with a unique combination of programmability, development cost, and efficiency that can fully benefit from various compression techniques. How can one use these features to implement optimized DNNs efficiently?

2.3 Co-design and Automation

The design of custom hardware architectures for DNNs and their implementation on FPGA is a time-consuming process that requires much expertise in the field of deep learning, hardware design, and implementation. To efficiently implement DNNs on a specific FPGA platform and to meet certain requirements, e.g., power consumption and latency, we have to consider an enormous amount of design parameters starting from neural topology down to hardware architecture and physical implementation. Importantly, interdependencies between the different design layers have to be considered, making it impossible to find optimal solutions manually. In spite of the advantages of FPGA, very often CPUs and GPUs are preferred over FPGAs because of the faster and easier development process. Fast and efficient implementation of DNNs on FPGAs can be achieved by combining recent advances:

Co-design. DNN's topologies and hardware architectures have to be co-designed by joint optimization of performance and efficiency while maintaining accuracy. It can be achieved using an automatic search for neural topologies, like NAS [18, 19, 20, 21]. NAS is a technique for automating the design of neural networks that can be on par with or outperform state-of-the-art hand-designed models.

Parametrizable hardware. Deployment of libraries of parametrizable hardware components substantially accelerates the hardware design and implementation. Such libraries are already adopted in industry and research [22]. The parametrizable hardware components balance specialization and generality. A sweet spot lies in building a library of components that accelerate a domain of applications rather than a single application. In the context of DNNs, libraries, like [23, 22], accelerate various DNN layers rather than particular topology. Parametrization allows supporting a wide range of parameters of different layers.

Automated hardware design. Use of High-level Synthesis (HLS) is proved to significantly accelerate the development process and provide results competitive to Hardware Description Languages (HDLs). HLS is an automated design process that takes an abstract behavioral specification of a digital system usually expressed in a high-level programming language, like C/C++, and finds a register-transfer level structure that realizes the given behavior.

3 Methodology

As has been mentioned in the previous sections, the design space is composed of a very high number of parameters, which makes it impossible to find optimal solutions manually.

There are many techniques for automatic exploration of the vast design space of DNNs. Among the most successful approaches are Reinforcement Learning (RL), gradient-based, and evolutionary algorithms. The goal of RL is to train a so-called agent to take actions that maximize a reward. In the context of NAS, the RL-based methods usually treat the DNN hyperparameters as actions and the evaluation criteria (e.g., accuracy) as the reward. In [24], Baker et al. used RL agent to generate new neural topologies, while Zoph et al. in [25] used RL for training a RNN that was responsible for composing new topologies. One of the first attempts to augment NAS with hardware-aware objectives was made by Tan et al. in [18], who used RL-based multi-objective NAS algorithm for optimizing both accuracy and physically measured inference latency of models on mobile phones. The RL agent was able to find models with lower latency than hand-designed models. However, RL-based methods require training the agent in addition to performing the actual neural search, which makes these methods very time-consuming and less flexible with respect to changes in the search space. A gradient-based method, like Differentiable ARchiTecture Search (DARTS) that was proposed by Liu et al. in [26], allows faster search of the architecture using gradient descent, unlike RL based methods that use random sampling. Wu et al. in [19] using a similar setup to [18] replaced RL agent with DARTS. They could speed up the NAS by two orders of magnitude, meanwhile achieving even better characteristics than [18]. Evolutionary algorithms

do not require training an agent nor a supergraph like gradient-based methods. Genetic algorithms use biologically inspired concepts to describe DNN structure and perform a search for new architectures. Real et al. in[27] presented an evolutionary algorithm that outruns RL based method and outperformed state-of-the-art hand-designed models. In order to consider resource consumption of models in NAS, Elsken et al. in [28] proposed LEMONADE algorithm for multi-objective architecture search. Using LEMONADE in [29] Schorn et al. incorporated error resilience, energy consumption, latency, and required bandwidth of DNNs on hardware to the NAS.

The proposed methodology is based on the evolutionary algorithm presented in [30]. The main mechanisms of the evolutionary algorithm comprise *mutation*, *evaluation*, and *selection*. The selection strategy used in the current NAS is based on the Bayesian sampling method used in [28]. The candidate topologies are sampled from less explored regions of the Pareto frontier. These candidates are evaluated first on computationally "cheap" objectives that are hardware-level objectives. They are "cheap", as they do not require training a network and can be computed based on hyperparameters using a closed-form expression. The candidates that fulfill certain hardware criteria are selected for evaluation with computationally "expensive" objectives that are application-level objectives, e.g. accuracy. They are "expensive" as they require to train a network. At the end of selection and evaluation procedures, the Pareto frontier is updated with fully evaluated topologies. These topologies are mutated in the next generation using techniques from [28]. The hardware requirements are considered in the form of hardware awareness, which is expressed in two ways in the NAS, namely constrained search space and optimization objectives. First, the search space is constrained by the hardware library. The NAS only searches for DNNs that can be constructed from the hardware library, which encompasses the types of layers and combinations of hyperparameters, including quantization. Therefore, the NAS generates only fully hardware-compatible solutions. Second, the NAS searches for topologies that are optimal with respect to the hardware optimization objectives that are formulated based on the architecture templates as a function of hyperparameters of the neural topology and hardware parameters. The output of the NAS is a network topology, trained weights, and quantization of the weights and output activations.

4 Hardware library

To enable cross-layer optimizations, we present a flexible HLS hardware library of custom hardware architectures, which can facilitate various DNN topologies. Earlier, we presented custom hardware architectures for standard One-dimensional Convolutional Neural Networks (1D-CNNs), depth-wise separable 1D-CNNs, and various other DNN layers and components suitable for unidimensional signal processing. The hardware architectures are highly customizable, which allows the implementation of various neural topologies. The hardware library is written as a collection of C/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, therefore reducing memory access to the absolute minimum. 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. 9. 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. 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. 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.

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

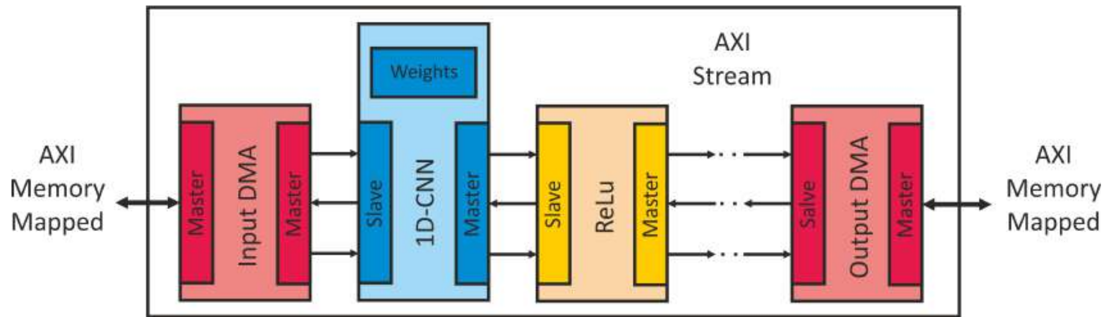

Figure 9: Deep Neural Network Unit (DNNU)

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 applying coarse-grained parallelization on a level of filters for Convolutional Neural Network (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.

Among other platforms, we target Zynq-7000 SoC and Zynq UltraScale+ MPSoC devices. These devices comprise the Programmable Logic (PL), which is the FPGA fabric, and the Processing System (PS), which is a processor. The DNNU can be integrated as a module in the PL of the device with direct connections to the PS for controlling the module as it is depicted in Fig. 10. Alternatively, the hardware architecture can be instantiated as a module on FPGA without any processor to coordinate data transfers to and from the design.

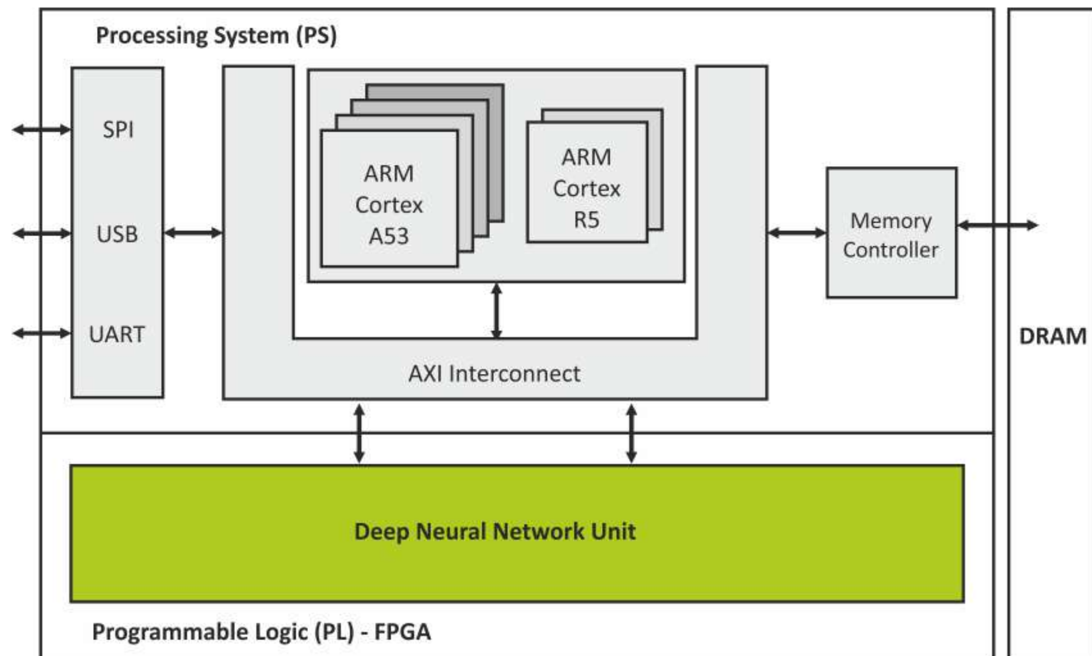

Figure 10: DNNU instantiation on Zynq

5 Custom data type

One of the recent emerging trends to increase energy efficiency is to adapt the low-precision data formats for both inference and training of DNNs. Typically, training is done using floating-point formats as they are able to provide wide dynamic range and high precision. The main challenge of using low-bit-width data formats such as floating-point 8-bit (FP8) data format is the limited dynamic range compared to floating-point 32-bit (FP32). One possible solution to compensate for the range limitation of FP8 is to shift the data format representable range to the desired location. This can be done by varying the bias value of the floating-point equation instead of using a common bias value for all FP8 values. Our target is to adaptively find the optimum bias value of FP8 for each given DNN model. Figure 11 shows an example of shifting FP8 dynamic range to the DNN parameter location.

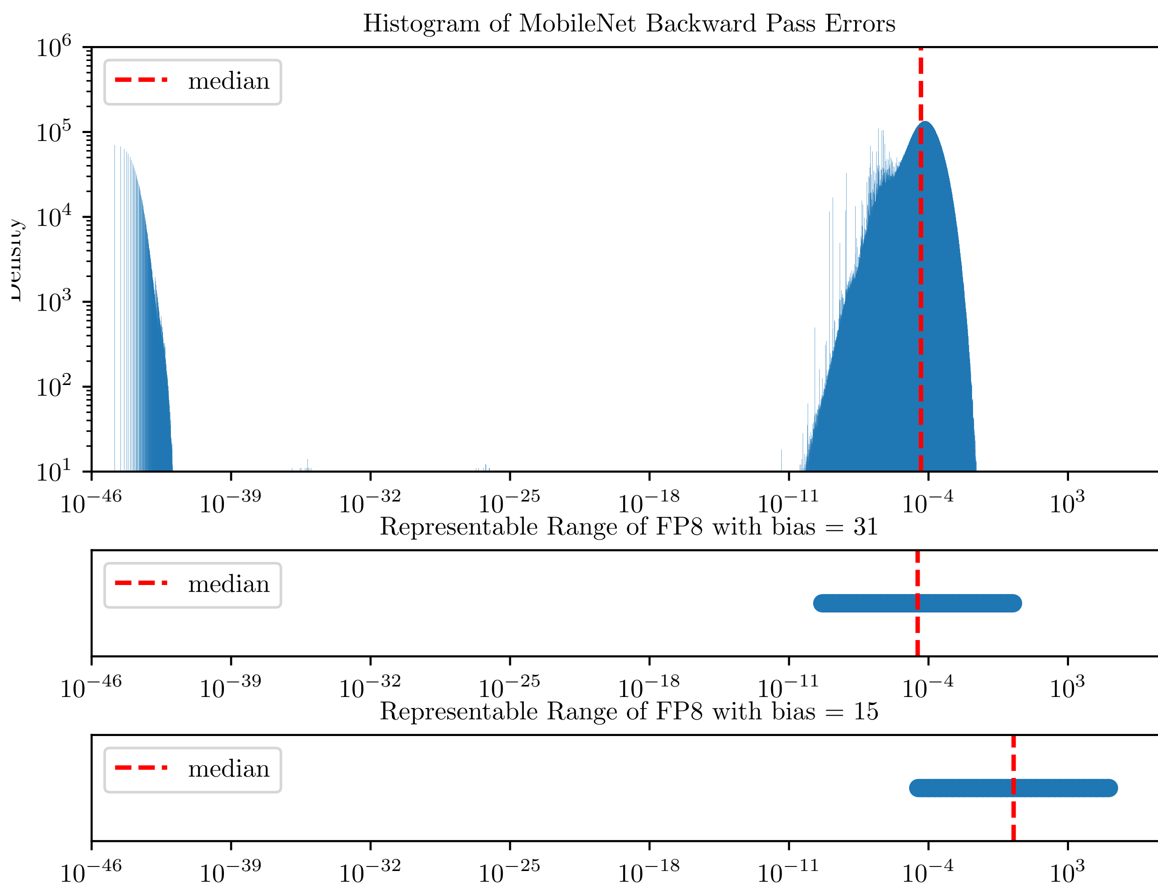

Figure 11: Fitting the FP8 data format range to the MobileNet-V2 backward pass errors histogram. The data points shown in this histogram exclude the sign.

There are two FP8 configurations, first with 5-bit exponent and 2-bit mantissa and another one with 4-bit exponent and 3-bit mantissa. It is proven in the state-of-the-art that using (1,4,3) in the backward pass is impossible. As we aim to use the same data format for forward and backward passes in this work, we use FP8 with a 5-bit exponent. Our methodology uses statistical analysis to find the optimum bias value for a given DNN model. Initial epochs of training are performed in FP32. The FP32 data that are read and written to the Dynamic Random Access Memory (DRAM) are sampled by a statistical analysis unit to identify a suitable bias value by analyzing the data distribution. Our statistical analysis unit finds the

Table 2: Reference table for mapping median value to Bias

Median Value	Bias	Median Value	Bias	Median Value	Bias
64	10	2	15	0.0625	20
32	11	1	16	.	.
16	12	0.5	17	.	.
8	13	0.25	18	.	.
4	14	0.125	19	0.0000305175	31

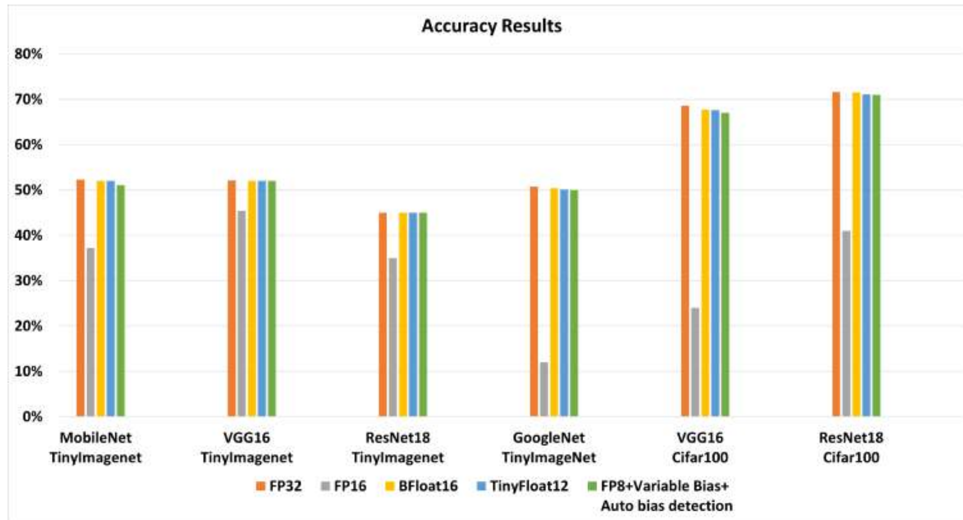

Figure 12: DNN training accuracy results comparison

median of DNN parameter. Then, we find the nearest median in Table 2. This table lists the middle value (which is equal to the median) of FP8 data format for different bias values. By this method, we find the bias value, which shifts the FP8 dynamic range to the location with the highest coverage for a given DNN model. After identification of a suitable bias value, DRAM write requests from the compute core are quantized to FP8. Similarly, the read data are dequantized to FP32 before forwarding it to the core. This enables a linear reduction of the total number of DRAM accesses since more data words can be packed in each transaction. Figure 12 shows the accuracy results of the proposed data format against other state-of-the-art data formats such as IEEE-754 FP32, FP16, and BFloat16. Our evaluations are conducted on DNN models such as VGG-16, ResNet18, MobileNet-V2, and GoogleNet. Furthermore, our evaluations consider datasets such as Cifar-100, and TinyImageNet. For all the networks, our methodology resulted in minimal DNN training accuracy loss. The resulting accuracy reduction of the FP8 format is on average 1% lower compared to the reference FP32 format. These results show we can reduce the energy consumption of DNN training by utilizing our custom data format with negligible accuracy loss.

6 Framework

To facilitate fast implementation of topologies found by the NAS and mapped onto custom hardware architectures, we implemented the Holistic Auto machine Learning for FPGAs (HALF) framework that is comprised of two main components, which are the hardware-aware NAS and the FPGA implementation framework as depicted in Fig. 13. The HALF framework receives as an input a dataset, a sketch of the DNN design space, and the requirements, which can be specified in terms of application-level and

hardware-level constraints and optimization objectives. As an output, the framework automatically produces a hardware implementation for the selected FPGA platform that fulfills the requirements. The sketch of the design space encompasses the types of layers, the range of hyperparameters, etc.

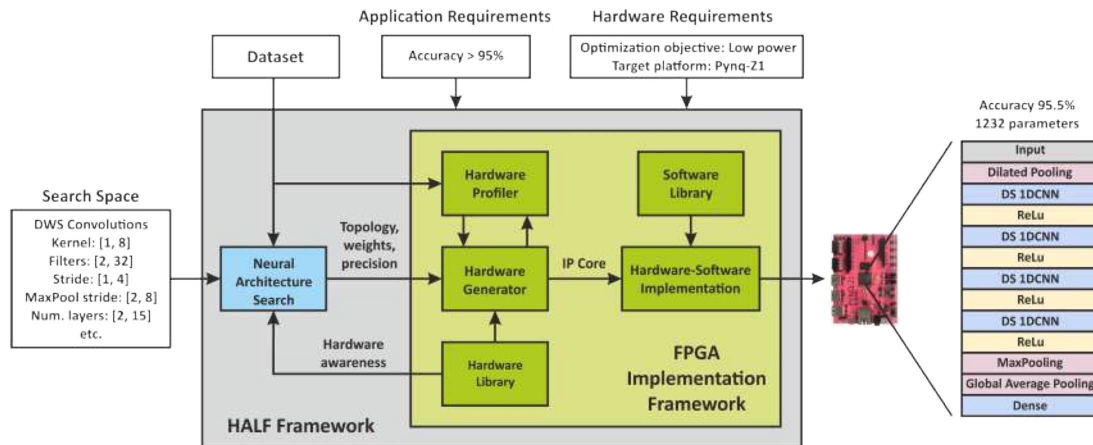

Figure 13: Components of the HALF framework.

The HALF significantly accelerates the deployment of DNNs on FPGAs. The automatic NAS takes days, depending on the complexity of the underlying search space and the size of the dataset, while the manual search can take weeks even without considering hardware awareness. Including hardware awareness into NAS shortcuts the otherwise time-consuming manual network design and evaluation of different FPGA implementations to identify candidates with the best trade-offs. The automatic hardware generation and implementation take hours in contrast to a manual process that can take days or even weeks, especially if hardware components have to be designed from scratch. The HALF framework accelerates the design cycle significantly, it reduces the deployment time from months to days.

The first step in the framework is NAS explained previously in Section 3. The second step is the FPGA implementation framework that comprises a hardware generator, a custom hardware library, a profiler for activations, 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 4. In particular, it uses Xilinx Vivado HLS to generate an Intellectual Property (IP) core from the DNN topology based on the components of the hardware library. At this point, the model weights are integrated into the IP core because the hardware architecture uses only on-chip memory for model storage. Additionally, it instantiates interfaces for communication with external memory and First In First Out memory (FIFO) buffers for connecting the elements. The hardware generator also calculates parallelization factors for each layer based on the layer's latency, the required throughput, the available resources, and the memory bandwidth on the target platform. While the quantization of weights and output activations is provided by the NAS, the quantization for the internal accumulators is found by the hardware profiler. The profiler identifies the optimal range and precision for all accumulators in the hardware and sets the bit widths accordingly. The IP core produced by the hardware generator is used in the hardware-software implementation step, which generates a bitstream for the FPGA configuration and a software executable. Xilinx Vivado Design Suite project is created at the hardware implementation step, the IP is instantiated and connected to the interfaces, logic synthesis and place and route (P&R) are performed, and the bitstream is generated to configure the FPGA. The software, compiled for running on the processor cores, is used to transfer input and output data to the FPGA and to control the IP core.

7 Summary

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. At the highest level of 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.

As the DNN topology is on the highest level of the design hierarchy, the changes applied on the topology have potentially the highest impact on the properties of the final implementation. DNNs have to be designed to achieve the required accuracy and fulfill hardware criteria, especially on edge devices because they have small memory, constrained computing capabilities, memory bandwidth, power, and energy budgets. The analysis shows that arithmetic operations are "cheap" while memory accesses are "expensive" and have a cost that is a function of the size of the memory being accessed. The relative energy cost should be used as the main guidance for designing DNNs. The hardware-aware DNNs have to be small to fit into on-chip memory ideally, or to mitigate any communication with the external memory, they have to use fewer operations and leverage low-precision data types.

There is a lot of ongoing research on developing networks with lower computation costs and storage consumption without impairing classification accuracy. The efficient models became possible due to multiple macro- and micro-architectural improvements of the models. The types of layers and their arrangement are referred to as macro-architecture. The efficient micro-architectural approaches are: *a*) very deep models are replaced with fewer layers, but with more channels, *b*) activation feature maps are kept smaller, *c*) models are enhanced with skip and residual connections that have been proven to improve accuracy, *d*) standard convolutions are replaced with depth-wise separable ones. The micro-architecture also defines methods applied to individual layers, like replacing big convolutional kernels with smaller ones and fusing different layers. Further techniques have been proposed to alleviate the computing and storage challenges. Among the most common ones are distillation [6], pruning [7, 8, 9], and quantization or a combination thereof [10]. The resulting model after knowledge distillation does not require special treatment during inference, unlike after pruning and quantization. In summary, pruning and quantization are the primary model compression techniques, but they require special treatment which raises the question of selecting an implementation platform that can fully benefit from them.

Comparing different hardware platforms: Central Processing Unit (CPU), Graphics Processing Unit (GPU), Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuits (ASICs), one can see that these options present a trade-off between flexibility and efficiency. While CPUs, GPUs are highly flexible, they cannot fully benefit from the major optimization techniques, like pruning and quantization. Only ASICs and FPGAs can fully benefit from them. The main reasons why FPGAs and ASICs are highly efficient in comparison to general-purpose computing platforms are: *a*) optimized memory, *b*) data specialization, *c*) massive parallelism, *d*) reduced overhead, *e*) algorithm-architecture co-design. Using these advantages, one can use FPGAs to provide ad hoc solutions to facilitate computationally intensive, time-critical tasks at low-power consumption in a reprogrammable manner, unlike ASICs. In summary, FPGA is a computing platform with a unique combination of programmability, development cost, and efficiency that can fully benefit from various compression techniques. How can one use these features to implement optimized DNNs efficiently?

The design of DNN topologies, custom hardware architectures for DNNs, and their implementation on FPGA is a time-consuming process. To efficiently implement DNNs on a specific FPGA platform and to meet certain requirements, e.g., power consumption and latency, we have to consider an enormous amount of design parameters starting from neural topology down to hardware architecture and physical implementation. Importantly, interdependencies between the different design layers have to be considered, making it impossible to find optimal solutions manually. Fast and efficient implementation of DNNs on

© 2023 SustainML | HORIZON-CL4-2021-HUMAN-01 | 101070408

FPGAs can be achieved by combining recent advances: *a*) co-design, meaning DNN's topologies and hardware architectures have to be co-designed by joint optimization of performance and efficiency while maintaining accuracy, which can be achieved using Neural Architecture Search (NAS), *b*) parametrizable hardware templates to build libraries of hardware components that support a wide range of parameters of various layers, *c*) facilitated hardware design using, e.g. High-level Synthesis (HLS) to accelerate the development process.

There are many techniques for automatic exploration of the vast design space of DNNs. Among the most successful approaches are *a*) Reinforcement Learning (RL), *b*) gradient-based, *c*) and evolutionary algorithms. Evolutionary algorithms do not require training an agent, like RL-based methods, nor a supergraph, like gradient-based methods. It was shown that the evolutionary algorithm can outrun RL-based method and outperforms state-of-the-art hand-designed models. The proposed methodology is based on the evolutionary algorithm presented in [30]. The main techniques responsible for the high efficiency of the NAS implementation are: *a*) multi-objective Pareto optimization, which enables multi-criteria optimization, *b*) Bayesian sampling method to improve the candidate selection process, *c*) evaluation of the candidates using "cheap" and "expensive" objectives to accelerate the evaluation process. Therefore, the NAS finds fully hardware-compatible solutions that are optimal with respect to the hardware optimization objectives and fulfill the application requirements.

To enable cross-layer optimizations, we present a flexible HLS hardware library of highly customizable hardware architectures, which can facilitate various DNN topologies. The hardware library is written as a collection of C/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 *a*) keeping all weights and intermediate results in on-chip memory since off-chip transfers consume more energy and introduce extra latency, *b*) external memory is only used to read input data and write results, therefore reducing memory access to the absolute minimum, *c*) separate hardware modules dedicated to each layer are connected using streaming interfaces to facilitate fast design, debugging, interoperability, and ease of integration, *d*) 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.

One of the recent emerging trends to increase energy efficiency is to adapt the low-precision data formats for both inference and training of DNNs. Typically training is done using floating-point formats as they are able to provide wide dynamic range and high precision. The main challenge of using low-bit-width data formats is the limited dynamic range compared to floating-point 32-bit (FP32). One possible solution to compensate for the range limitation of low-bit-width data formats is to shift the data format's representable range to the desired location. This can be done by varying the bias value of the floating-point equation instead of using a common bias value for all values. Our methodology uses statistical analysis to find the optimum bias value for a given DNN model. By this method, we find the bias value, which shifts the low-bit-width data format dynamic range to the location with the highest coverage for a given DNN model. We demonstrate the efficiency of the approach facilitating floating-point 8-bit (FP8) format for several datasets and state-of-the-art DNN topologies. This enables a linear reduction of the total number of Dynamic Random Access Memory (DRAM) accesses, which increases energy efficiency while keeping the accuracy on par with FP32 format.

To facilitate fast implementation of topologies found by the NAS and mapped onto custom hardware architectures, we implemented the Holistic Auto machine Learning for FPGAs (HALF) framework that is comprised of two main components, which are the hardware-aware NAS and the FPGA implementation framework. The framework automatically produces a hardware implementation for the selected FPGA platform that fulfills the requirements. The HALF framework accelerates the design cycle significantly, it reduces the deployment time from months to days.

References

[1] Jonas Ney et al. "HALF: Holistic Auto Machine Learning for FPGAs". In: *Accepted for publication, the 31st International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2021.

[2] Mark Horowitz. "1.1 computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE. 2014, pp. 10–14.

[3] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[4] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. "An analysis of deep neural network models for practical applications". In: *arXiv preprint arXiv:1605.07678* (2016).

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

[7] Baoyuan Liu et al. "Sparse convolutional neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 806–814.

[8] Song Han et al. "Learning both weights and connections for efficient neural network". In: *Advances in neural information processing systems*. 2015, pp. 1135–1143.

[9] Wei Wen et al. "Learning structured sparsity in deep neural networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2074–2082.

[10] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[11] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *arXiv preprint arXiv:1802.05668* (2018).

[12] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[13] William J Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators". In: *Communications of the ACM* 63.7 (2020), pp. 48–57.

[14] Jiecao Yu et al. "Scalpel: Customizing dnn pruning to the underlying hardware parallelism". In: *ACM SIGARCH Computer Architecture News* 45.2 (2017), pp. 548–560.

[15] *Project Brainwave - Microsoft Research*. `https://www.microsoft.com/en-us/research/project/project-brainwave/`. Accessed: 2021-05-26.

[16] Jeremy Fowers et al. "A configurable cloud-scale DNN processor for real-time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 1–14.

[17] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[18] Mingxing Tan et al. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[19] Bichen Wu et al. "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. 10734–10742.

[20] Hongxiang Fan et al. "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.

[21] Weiwen Jiang et al. "Hardware/software co-exploration of neural architectures". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4805–4815.

[22] Farah Fahim et al. "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices". In: *arXiv preprint arXiv:2103.05579* (2021).

[23] Yaman Umuroglu et al. "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.

[24] Bowen Baker et al. "Designing neural network architectures using reinforcement learning". In: *arXiv preprint arXiv:1611.02167* (2016).

[25] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016).

[26] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *International Conference on Learning Representations*. 2018.

[27] Esteban Real et al. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 2019, pp. 4780–4789.

[28] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution". In: *International Conference on Learning Representations*. 2018.

[29] Christoph Schorn et al. "Automated design of error-resilient and hardware-efficient deep neural networks". In: *Neural Computing and Applications* (2020), pp. 1–19.

[30] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. "A genetic programming approach to designing convolutional neural network architectures". In: *Proceedings of the genetic and evolutionary computation conference*. 2017, pp. 497–504.