FlexRFML: Dynamic Neural Networks on FPGAs for Next-Generation Radio Spectrum Perception

Francesco Pessia, A.Q.M. Sazzad Sayyed and Francesco Restuccia
Northeastern University, United States

Abstract

Enabling spectrum perception with deep neural networks (DNNs) directly connected to the radio front-end is of fundamental importance to realize next-generation spectrum-aware wireless systems. As such, low-latency DNN inference in reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs) is a necessary precursor to enable spectrum perception in real-world wireless systems. The key issue with existing work is that it considers DNNs that have fixed weights and architecture. On the other hand, it has been shown that *dynamically* changing the structure and weights of the DNN at runtime can lead to improved efficiency and adaptability. This work fills the current research gap by proposing FlexRFML, the first framework to integrate dynamic DNNs in the RF-front spectrum perception loop. FlexRFML includes High-Level Synthesis (HLS)-based design as well as customized circuits to achieve dynamic hardware reconfiguration and accelerate the DNN. We have prototyped FlexRFML on a Xilinx system-on-chip (SoC) ZCU102 by considering both modulation recognition and radio fingerprinting classification problems where the DNNs are dynamically adapted based on a preliminary classification of the input. Experimental results show that FlexRFML can decrease the inference latency by up to 35.6% with respect to static DNN inference with negligible additional hardware overhead. We pledge to release the FlexRFML hardware and software code.

CCS Concepts

• **Computer systems organization** → **Real-time systems**; • **Hardware** → **Hardware-software codesign**.

Keywords

Edge Computing, Adaptive Inference, Neural Networks, FPGA.

ACM Reference Format:
Francesco Pessia, A.Q.M. Sazzad Sayyed and Francesco Restuccia. 2025. FlexRFML: Dynamic Neural Networks on FPGAs for Next-Generation Radio Spectrum Perception. In *International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '25), October 27–30, 2025, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3704413.3764472

MobiHoc '25, October 27–30, 2025, Houston, TX, USA

ACM ISBN 979-8-4007-1353-8/25/10
https://doi.org/10.1145/3704413.3764472

1 Introduction and Motivation

The radio spectrum is a finite natural resource that lies at the foundation of wireless communications. The fast-paced rise of the Internet of Things (IoT), projected to reach 50B by 2030 [1], will exacerbate the spectrum crunch in the years to come [9]. Techniques such as spectrum sharing and dynamic spectrum access through deep neural networks (DNNs) have demonstrated their effectiveness in enhancing spectrum utilization [26], radio fingerprinting [23] and automatic modulation recognition [20], just to name a few [12]. To obtain actionable knowledge, the DNNs must perform real-time inference on I/Q samples with latency in the order of microseconds or below [25], which implies that any computation needs to be implemented directly in the radio frequency (RF) processing chain. Sub-milliseconds inference on local low-power devices enables ML-driven optimizations of the network protocol stack based on temporarily relevant spectrum information, enhancing its utilization in the next generation of radio sensing devices. *As such, graphic processing units (GPUs) and similar software-based accelerators cannot be used for radio spectrum perception.*

To tackle this issue, the concept of "*learning-in-the-RF-loop*" has been recently proposed to achieve real-time spectrum perception by implementing DNNs in the reconfigurable radio hardware of wireless devices, thus significantly decreasing latency with respect to software-based inference [24]. The key issue is that existing work considers the case of *static* DNN inference. However, prior work has shown that *dynamic neural networks (DyNNs)* can lead to significant performance improvement. Critically, DyNNs specialize the DNN computation at inference time according to a preliminary characterization of the input, thus improving efficiency, adaptability, security and interpretability [11]. The top portion of Figure 1 shows an example of a static DNN trained to performed modulation recognition. To withstand different Signal-to-Noise-Ratio (SNR) levels, and a DyNN in spectrum perception applications, decreasing the DNN complexity in case of favorable channel conditions – i.e. high SNR – may lead to a decrease in inference latency.

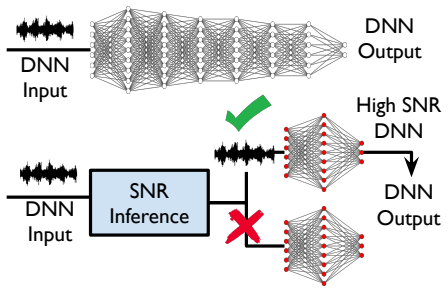

Figure 1: Static DNN (top) vs Dynamic DNN (bottom).

To support the rapid design and development of DyNNs, Field Programmable Gate Arrays (FPGAs) prototypes serve as essential precursors before large-scale production in the form of low-power application-specific integrated circuits (ASICs). Existing work on FPGA-based DyNNs acceleration – discussed in details in Section 4 – mainly focuses on early-exit DNNs [4, 8] as well as DNN with adaptive computation precision [16]. The former reduces latency by leveraging intermediate classifiers, called exiting points, while the latter dynamically switches between low and high-precision quantized version of the same accelerator. Dynamic pruning architectures where a run-time manager adaptively selects the most suitable pruning rates according to the workload have also been proposed [15]. However, these approaches resort to partial reconfiguration of the FPGA fabric, which incurs excessive latency – up to 42 ms [8] – and is not sustainable for ASICs. On the other hand, as shown in Figure 1, reconfiguring the DNN without modifying the FPGA fabric can enable fine-grained customized computation.

The key challenges in achieving a dynamic pruning architecture without relying on partial reconfiguration are: (i) designing a flexible data path from the input I/Q to the sub-DNNs and (ii) automating the selection and execution of the appropriate sub-DNN. The latter, in particular, requires an accelerator capable of efficiently executing sub-DNNs using dynamic convolutions. Unlike standard convolutions, dynamic convolutions involve processing only specific subsets of a layer's filters and input tensor channels, which are often stored in non-contiguous memory regions. As such, the designed architecture must include dedicated logic to accurately access discontinuous memory locations that could degrade the accelerator's efficiency due to cache misses and memory latency. Work like [17] goes around this limitation by introducing constraint on the pruned filters, which limits the achievable accuracy and acceleration gain. `FlexRFML` solves this challenge by going beyond the traditional data path and control path based architectures and introduces an address path for dynamic memory access.

To fill these research gaps, this paper makes the following core contributions:

• We propose `FlexRFML`, a framework that brings for the first time DyNN to FPGA for next-generation radio spectrum perception without resorting to partial reconfiguration. `FlexRFML` maps to hardware accelerators any user-defined DyNN through a customized library based on *High-Level Synthesis* (HLS). The proposed hardware-software `FlexRFML` architecture includes several hardware cores designed to accelerate the dynamic layers by also guaranteeing real-time reconfiguration. The cores configuration is changed though small binary files stored directly in the DNN internal memory. The correct configuration is autonomously selected at runtime according to the perceived input sample;

• We prototype `FlexRFML` on a Xilinx ZCU102 System-on-Chip (SoC) and evaluate its performance in terms of accuracy, latency, area and power consumption. We test `FlexRFML` on two use cases using existing radio perception datasets available to the community: (i) automatic modulation classification and (ii) transmitter authentication through radio fingerprinting. Experimental results indicate `FlexRFML` presents lower computation complexity (up to 69% FLOPs reduction) and lower inference latency (up to 35.6%) with respect to static DNNs.

2 The `FlexRFML` Architecture

Figure 2 provides a high-level overview of the `FlexRFML` design. We provide background notions in Section 2.1, followed by an overview of the novel hardware architectures in Section 2.2. Next, we provide a detailed discussion of the `FlexRFML` hardware and software components in Section 2.3 and 2.4, respectively. Finally, we provide the software-to-hardware orchestration module in Section 2.5.

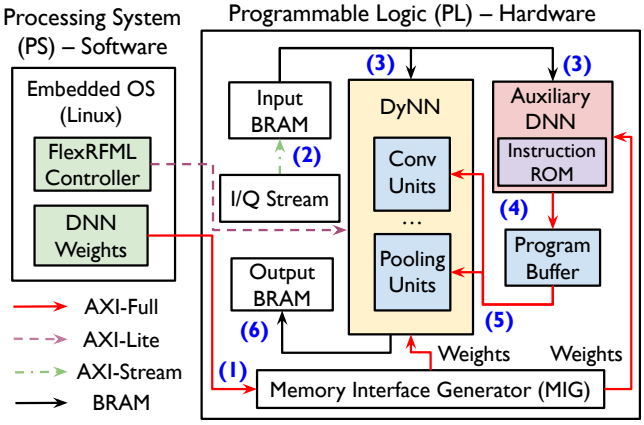

Figure 2: Overview of the `FlexRFML` SoC Architecture.

2.1 Background on High-Level Synthesis

The architectural components of `FlexRFML` reside in a System-on-Chip (SoC), which combines components such as Central Processing Unit (CPU), random access memory (RAM), input/output (I/O) ports, and secondary storage into a single substrate. Importantly, SoCs implement customizable hardware on the FPGA section of the chip, known as the programmable logic (PL). The PL can be configured and monitored by the processing system (PS), which includes CPU and RAM.

`FlexRFML` is heavily based on high-level synthesis (HLS) for its hardware components. HLS is an automated design process that maps high-level languages – e.g., C/C++ – to equivalent RTL descriptions [22]. HLS tools can be guided during synthesis though directives (or *pragmas*), to achieve the desired performances in terms of latency, area and power consumption through optimizations based on pipelining or parallelization (i.e, loop unrolling). HLS maps I/O signals to standard industrial interfaces. `FlexRFML` employs the *Advanced eXtensible Interface* (AXI) bus specification [32] to move data between cores in the PL and between the PS and the PL. Specifically, `FlexRFML` uses *AXI-Lite*, *AXI-Stream*, and *AXI-Full*. AXI-Lite is designed for register-level access to configure the circuits within the PL. AXI-Stream handles high throughput data transportation between circuits located in the PL. AXI-Full is used to support burst-based data transfer between the PL and PS (both directions) and between PL and external memory devices. Figure 2 depicts the AXI-Full, AXI-Lite, and AXI-Stream interconnections with continuous, dashed, and dot-dashed lines, respectively.

2.2 Novel Architectures

The three novel hardware components proposed in `FlexRFML` are: *(i) Dynamic DNN* (DyNN) capable of accelerating sub-DNNs through

dynamic convolution and pooling sub-units programmable through binary file selected by *(ii) Auxiliary DNN* that conducts a preliminary inference on the input and loads/stores instructions in *(iii) a Program Buffer* to reconfigure the DyNN in real-time.

2.3 Dynamic Neural Network Architecture

Figure 3 provides a detailed overview of the proposed FlexRFML DyNN and Auxiliary DNN internal structures. The hardware circuits comprise several FlexRFML sub-units that accelerate the DNN layers. These sub-units are connected to local buffers for loading/storing intermediate results. The local buffers are synthesized by the HLS compiler as BRAMs and are *surface-packed* organized. The architecture is not pipelined neither layer-wise nor per-segments [5]. This enhances hardware sharing through folding and time-multiplexing, allowing the deployment of large DNNs on relatively small FPGAs. Figure 3 shows a DyNN accelerating two convolution, two max pooling and one fully connected layers. While pipelining the architecture would have required 4 BRAMs to load/store the intermediate feature maps, folding demands only 2 BRAMs. This is achieved through a multiplexer-demultiplexer controller that synchronizes the local buffers data paths with the sub-units.

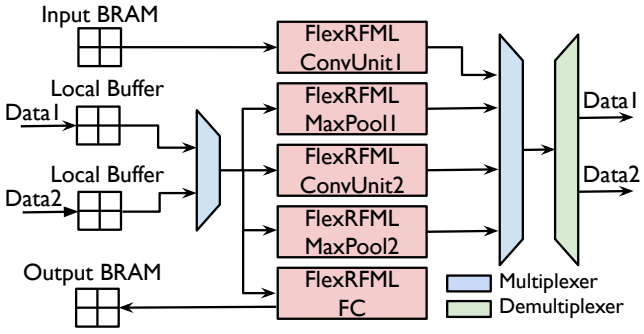

Figure 3: DyNN and Auxiliary DNN internal architectures.

Figure 4 provides a detailed overview of the FlexRFML sub-units interfaces. To ease readability, we have depicted in red the AXI-full and in black the BRAM interfaces. The FlexRFML ConvUnit handles dynamically adapted convolutions through software-based reconfiguration. This is achieved by fetching instruction from the Program Buffer in single-burst access. The program is executed by dedicated logic called *address path* and consists of a sequence of unsigned integers representing the *filter indexes* and *channel indexes*. The *filter indexes* indicate to the architecture the convolution filters to fetch from the DDR memory and apply. The *channel indexes* represent the active channels in the input feature map. Channels are active or disabled according to the *filter indexes* of the preceding convolution layer. The core intuition behind the proposed software reconfiguration is to dynamically compute memory offsets from the corresponding indexes to efficiently identify non-contiguous memory locations.

As a first step, the FlexRFML ConvUnit resets the content of the local buffer and loads the program to execute. A burst access request is issued by the architecture to fetch one filter from the DDR. The offset for the burst access is calculated resorting on one *filter index*. *Channel indexes* are leveraged to select and store in the *First-In*

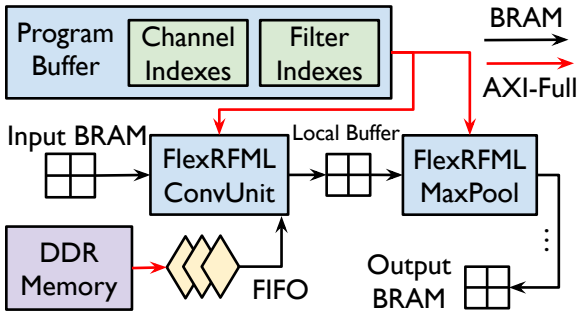

Figure 4: Overview of FlexRFML sub-units interfaces.

First-Out (FIFO) only the active channels of the previously fetched filter. Filter's channels can also be referred to as kernels. Next, the FlexRFML ConvUnit loads one active input channel and its corresponding filter's kernel in the *Convolution core*. The *Convolution core* architecture is inspired by the Sliding Window Unit [7]. This design decision handles: *(i)* input fed in a streaming fashion, *(ii)* fine-grained dynamic pruning rates *(iii)* stationary kernel's weights to avoid expensive memory transactions. Input channels and their corresponding filter kernels are fetched by the *Convolution core* until the FIFO is empty. At this stage, a new burst access request is issued to fetch another filter if there are *filter indexes* available. Bias are loaded from the DDR and accumulated in the buffer. To enhance computation latency, the core has the option to activate the ReLu function concurrently with bias aggregation. Pooling operations (i.e, Max, Avg) are conducted by the FlexRFML pooling unit. This sub-unit interfaces only with the local and Program buffers as shown in Figure 4. Furthermore, the program fetched by the FlexRFML pooling sub-unit comprises only *channel indexes*. Indeed, pooling operations are conducted only across active input channels. Next, multi-dimensional feature maps are flattened into vectors. The FlexRFML flattening unit conducts zero padding on the flattened disabled channels. This approach enables to directly feed the flattened vector to fully connected layers sub-units, thus executing heavy matrix multiplication operations in a static fashion and avoiding extra burden of indexing, weight-copying, or zero-masking that can lead to performances degradation [17].

2.4 Software Components

The PS consist a CPU running *PetaLinux*. The latter is a Linux version based on the Yocto Project [27], a widely-used platform for building custom Linux kernels. The Yocto project provides an interactive interface to integrate customized hardware drivers. In this work, two drivers have been developed to control the DyNN and the Auxiliary DNN. The drivers are tasked with several preliminary actions. Specifically, the drivers resort on the AXI-Lite interface to program the offsets of the DDR partitions in the DyNN and Auxiliary DNN internal registers. Furthermore, the driver configures the AXI-Full interfaces to enable data transfer from the instruction ROM to the DyNN. The configuration process consist of programming the DyNN and Auxiliary DNN registers with the physical address assigned during synthesis to the Program buffer. At this stage, the driver stores the DNN parameters in the DDR memory and places the I/Q stream to the I/O BRAMs. Finally, the driver enables the computation by writing the corresponding bit

in the control and status registers of the hardware components. A polling strategy [18] is adopted to constantly monitor the DyNN status register. As soon as the data path updates the status register, announcing computation completeness, the PS proceeds to collect results stored in the output BRAM.

2.5 Software-to-Hardware Orchestration

While general-purpose DNN accelerators resort on vast address spaces and flexible data paths, application-specific accelerators lack the same resources and as such, they inevitably lack the same flexibility. For this reason, `FlexRFML` includes an orchestration engine to close the gap between the software-based DNN and its hardware counterpart. Figure 5 overviews the proposed software-to-hardware orchestration. The framework comprises multiple steps involving hardware (depicted in blue), middleware (red) and software (green).

The framework enables the deployment of any user-defined DyNN by starting from a pre-trained static DNN. For the sake of generalization, we consider this DNN to be generated by a hardware-aware neural architecture search (NAS) algorithm taking into consideration the application (dataset) and any user defined hardware constraints (**step 1**). The computation complexity of the dynamic layers in the original DNN is adapted based on the preliminary classification of an Auxiliary DNN. The Auxiliary DNN is trained to infer user defined patterns in the input sample and thus select the appropriate sub-DNN for further input-specific processing. The patterns detected by the Auxiliary DNN can be for example semantic association [28], channel conditions [20], for example SNR or user mobility, or based on application-specific requirements, e.g., the complexity of the current operating environment [13]. The output from the Auxiliary DNN is either accepted or rejected based on a confidence-threshold tuned during model validation (**step 2**). Next, the sub-DNNs are extracted from the static DNN based on the recognized pattern from the Auxiliary DNN (**step 3**). This step involves: (i) identifying the dynamic layers, (ii) ranking the filters, (iii) extracting the sub-DNNs (iv) building the instruction ROM.

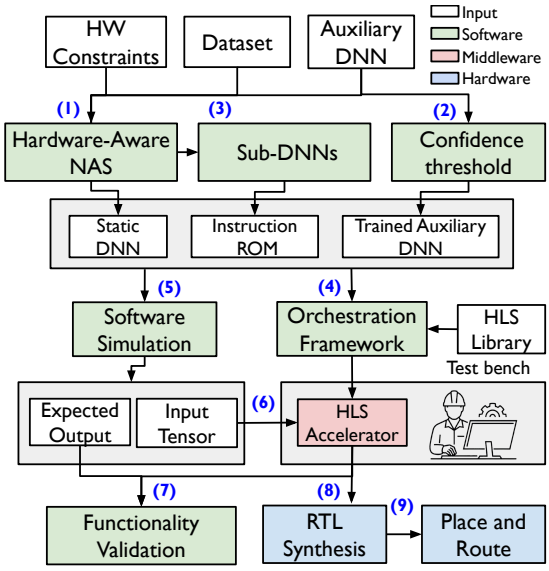

Figure 5: Software-to-Harware Orchestration in `FlexRFML`.

The number of static and dynamic layers in the DyNN depends on the computation complexity of the Auxiliary DNN. To achieve hardware synchronization, the waiting time of the instructions in the Program Buffer must be minimized. As such, the latency of the static layers computation should be equal or larger than the Auxiliary DNN. The filters in the dynamic layers are ranked by importance according to the Auxiliary DNN output. Next, the sub-DNNs are extracted leveraging the filter's ranking. The framework evaluates several sub-DNNs in terms of reduction of floating point operations (FLOPs) and accuracy degradation. Next, it builds the instruction ROM starting from the optimal sub-DNNs. At this stage, the software description of the DyNN and Auxiliary DNN are converted to HLS language (**step 4**) through a customized HLS library. The library comprises parametric and reconfigurable description of the `FlexRFML` sub-units designed to accelerate DNN's layers. The library retrieves the DNN and instantiates `FlexRFML` sub-units to map their parametric description to the layers hyper-parameters. Subsequently, the framework retrieves the content of the ROM to program the Auxiliary DNN. Then, a software simulation generates both input and expected output from the dynamic inference process (**step 5**). Next, the HLS accelerator undergoes HLS testing and synthesis flow that includes *(i)* behavioral simulation of the software description, *(ii)* synthesis to convert the HLS code to RTL, *(iii)* software-hardware co-simulation. The behavioral simulation ensures the correct functionality of the HLS-generated circuits. A test bench provides inputs to the top-level entity and captures the corresponding outputs (**step 6**). A functional validation module compares the HLS accelerator output against the expected values (**step 7**). Next, RTL is synthesized (**step 8**), with the post-synthesis report providing estimates of *(i)* the amount of resources consumed by the circuit (i.e, flip-flops, logic gates, BRAM blocks), *(ii)* the maximum clock frequency, *(iii)* power consumption. Then, the HLS tool reuses the test bench to ensure the functionality of the synthesized RTL description. Finally, the HDL code is integrated with the other PL components. After placing, routing and implementation (**step 9**), the resulting bitstream file is used to program the PL.

3 `FlexRFML` Prototype and Evaluation

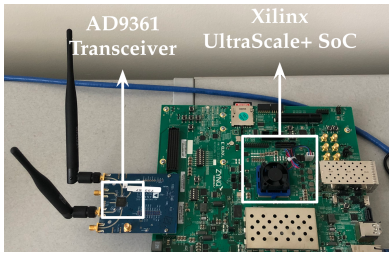

Figure 6: Experimental testbed used to evaluate `FlexRFML`.

To evaluate the performance of `FlexRFML`, we have designed and implemented a testbed shown in Figure 6. The testbed is composed of (i) a Zynq UltraScale+ XCZU9EG-2FFVB1156 multiprocessor, running on top of a Xilinx ZCU102 evaluation board; (ii) an Analog Devices (AD)-9361 RF transceiver [2] running on top of an AD-FMCOMMS2 evaluation board.

3.1 Proof-of-Concept Applications and Datasets

For our FlexRFML proof-of-concept, we have considered two radio spectrum perception applications: (i) automatic modulation recognition and (ii) radio fingerprinting. For each datasets, 80% of the samples were used for training and 20% for testing.

Modulation Classification. This problem entails classifying which modulation a transmitter is using at a given moment in time. We have used the *RadioML 2018.01A* open-source dataset [20], where up to 24 analog and digital modulations are collected and labeled in different propagation scenarios. The modulations are collected at different levels of SNR ranging between -20dB and 30dB.

Radio Fingerprinting. This problem deals with classifying which radio transmitter the received waveform belongs to. Specifically, radio fingerprinting leverages the inherent hardware imperfections present in every radio circuitry to form a unique and unforgeable "fingerprint" that can authenticate devices [23]. As far as the radio fingerprinting application is concerned, we have used the *AirID RF* dataset [19]. The I/Q samples are received and transmitted though *Ettus B200* mini software defined radios mounted on DJI Matrice M100 unmanned autonomous vehicles (UAVs). In [19], a fleet of 2 transmitting UAVs and 5 receiving UAVs is deployed outdoor. The data are transmitted with 5 different amplitude impairments to emulate 5 radio fingerprints.

3.2 Proof-of-Concept DyNN Design

We designed a dynamic inference procedure compatible with the hardware implementation and the orchestration engine proposed respectively in Sections 2.2 and 2.5.

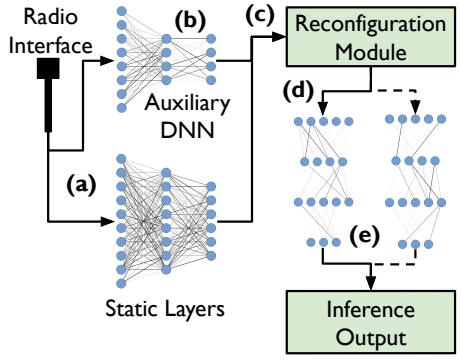

Figure 7: Overview of the DyNN inference process. The solid connector represents the chosen sub-DNN and dashed connectors are the excluded sub-DNNs.

Figure 7 overviews the proposed design. The dynamic inference comprises the following steps: **(a)** the input is fed into the static layers of the DyNN to extract common features **(b)** simultaneously, the same input is fed into an Auxiliary DNN. Next, **(c)** the prediction from the Auxiliary DNN and confidence level, along with the static extracted features, are fetched by a network reconfiguration module that **(d)** discriminated between full network inference (*static inference*) or select the input specific sub-DNN. Finally, **(e)** the inference output is computed by the dynamic part of the network. In this proof-of-concept, we train the Auxiliary DNN to infer the semantic clusters the input samples belong to. A semantic cluster is a group of predictable classes from the DyNN that share high-level features. Semantic association is often utilized in computer vision applications [28]. For example, images of cats and dogs can be semantically clustered as animals. The DyNN leverages the semantic cluster predicted by the Auxiliary DNN to activate the corresponding sub-DNN. To ease comprehension, we have depicted in Figure 7 with solid connections the activated sub-DNN and in dashed lines the discarded ones. In the hardware implementation, the sub-DNN selection and activation (steps **(c)-(d)**) is conducted by the confidence sub-unit in the Auxiliary DNN. The sub-DNNs execution (step **(e)**) is accelerated by the FlexRFML sub-units in the DyNN, described in details in Section 2.3. The static layers of the DyNN and Auxiliary DNN (steps **(a)-(b)**) are conducted by FlexRFML static sub-units, that do not support software-based reconfiguration.

3.3 Deep Learning Models Training

In this subsection, we shown the experimental results collected during steps (1), (2) and (3) of the orchestration framework.

3.3.1 Hardware-Aware Model Selection. Conversely from relying on computation-heavy hardware-aware NAS algorithms, we utilize model selection to determine the optimum architecture for the static DNN from which to build the DyNN. We start from the VGGNet architecture [29], which is composed of convolution blocks. The latter include two convolution layers, each with a Rectified Linear Unit (ReLU) activation and a max pooling (MaxPool) layer. The model selection algorithm incrementally increases the number of convolution blocks until the classification accuracy stops improving. Next, the algorithm selects the best architecture given the following constraints: (i) intermediate features' maximum size (in bytes) to avoid BRAM resources saturation for the local buffers; and (ii) number of sliding window units to avoid saturating logic components. The DyNN static structure search and training is conducted with four *NVIDIA A100* GPUs.

Modulation Classification. Table 1 shows the performance of the candidate architectures. While the accuracy is comparable with the ResNet accuracy presented in [20], the architectures require between 10% to 30% less parameters. The accuracy gain achieved in deeper VGGNets is minimal when compared to the growth of floating point operations (FLOPs) demanded for single input batch inference. For example, VGG16 has 3.58% gain in accuracy compared to VGG10 but requires 5.7x more operations and more than double parameters. The maximum local buffer size supported by the Xilinx ZCU102 is determined through iterative synthesis and set to 256KB. All the architectures, despite different depths, require the same amount of BRAM to load/store the intermediate feature maps. This is achieved through folding the local buffers as mentioned in Section 2.3. While the maximum number of sliding window units has not been set, we have chosen the lowest possible values. As such, we chose VGG8 and VGG10 as the most suitable candidates for PL deployment due to their optimal accuracy, complexity and number of sliding window units.

Radio Fingerprinting. Figure 8 illustrates the performances of the candidate VGGs for the radio fingerprinting problem as a function of the link distance (measured in feet). As in [19], the identification accuracy is strongly dependent on the communication channel. In

VGG	VGG8	VGG10	VGG12	VGG14	VGG16
ConvBlocks	3	4	5	6	7
Accuracy	82.94%	84.08%	85.60%	87.54%	87.66%
FLOPs	3.9M	4.73M	7.94M	14.3M	26.95 M
Parameters	430.83 k	447.14 k	484.2 k	632.04 k	1.22 M
Local Buffers	96KB	96KB	96KB	96KB	96KB
Sliding Units	6	8	10	12	14

Table 1: Performances of the candidate VGG architectures for the automatic modulation classification problem.

particular, when the drones distance is lower than 10 feet, all the considered classifiers achieve 99% accuracy. While our architectures have comparable accuracy with the VGG classifiers in [19], our architectures require between 7.52x and 18.4x less FLOPs. The search algorithm identified VGG4 and VGG6 as the most suitable candidates for FPGA deployment.

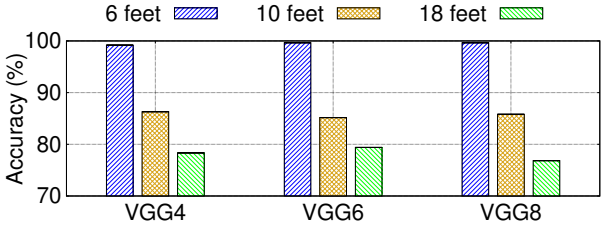

Figure 8: Performances of the candidate VGG architectures for the radio fingerprinting problem.

3.3.2 Auxiliary Networks Confidence Tuning. Next, we cluster the dataset and train the *Auxiliary* DNNs. To obtain the Auxiliary DNN architecture, we set a threshold based on the worst-case accuracy requirement and computational budget. Then we evaluate from a suit of auxiliary networks of different complexities and select the one with the least computational budget to increase the number of dynamic layers. The training of the auxiliary scout is conducted for 50 epochs with a learning rate of 0.001 and Adam optimizer on subsets of the same training/validation split employed for the DyNN.

Modulation Classification. We divide the 24 classes into 6 semantic clusters: *ASK, PSK, APSK, QAM, frequency modulations (fM)* and *analog modulations (aM)*, according to the modulation type. For example, 16APSK, 32APSK, 64APSK and 128APSK are clustered in APSK. The key intuition is that all the APSK modulations are based on amplitude and phase-shit keying and thus share similar features. The Auxiliary DNN architecture is illustrated in Table 2. Due to the limited number of layers, a single Auxiliary DNN is not capable to generalize its prediction across different levels of SNR without losing in accuracy. On the other hand, the accuracy recovers when training the Auxiliary DNN with I/Q samples collected at fixed SNR. In order to address this challenge we propose to conduct context switching whenever the channel noise fluctuates. This mechanism involves dynamically selecting the correct set of model's parameters according to the current operating environment (e.g, SNR). In hardware, this procedure is implemented by storing all the possible Auxiliary DNN configurations in the DDR memory. Then, the DDR-offset register in the Auxiliary DNN is dynamically modified by the PS to fetch the correct set of parameters according to the SNR. The proposed methodology achieves noise-driven adaptability with minimal delay (one Axi-Lite operation). Multiple auxiliary DNNs were trained, one for each SNR level spanning between 10 dB to 30 dB. We evaluate different confidence thresholds in terms of semantic cluster prediction accuracy and the probability of dynamically exclude filters in the VGG layers. Figure 9 shows the performance of the Auxiliary DNNs, averaged across different SNR levels, as a function of the confidence thresholds. The results indicate that higher confidence thresholds enhance the classification accuracy yet increase the probability of static inference, thus limiting the sub-DNN activation. These results are collected during the Auxiliary DNNs validation phase. The optimal threshold is determined while optimizing the weighted sum between accuracy and probability. The worst-case accuracy, associated to the lowest confidence threshold, oscillates between 50.4% and 63.7% according to the SNR. The worst-case accuracy is not monotonically dependent to SNR.

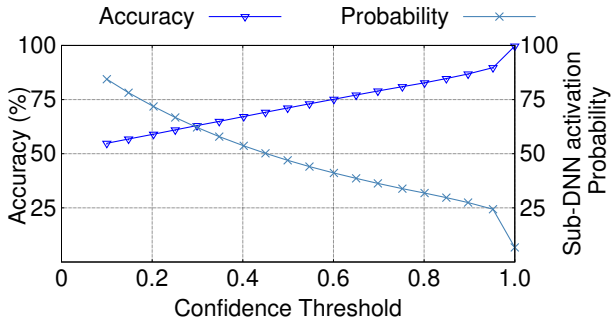

Figure 9: Performances of Auxiliary DNN as a function of different confidence thresholds in modulation classification.

Layer	Output dimensions
Input	2x1024
Conv1D	4x1024
MaxPool	4x512
Conv1D	6x512
MaxPool	6x256
FullyConnected	1x100
FullyConnected	1x6
SoftMax	1x6

Table 2: The Auxiliary DNN architecture for the automatic modulation classification problem

Radios	Radio1	Radio2	Radio3	Radio4	Radio5
$\mathbf{P}(Cluster0)$	99.7%	99.5%	95.9%	17.37%	1.21%
$\mathbf{P}(Cluster1)$	0.23%	0.5%	4.1%	82.6%	98.79%
Amplitude impairment	1dB	2dB	3dB	4dB	5dB

Table 3: Unsupervised clustering results

Radio Fingerprinting. Defining semantic clusters is challenging due to lack of clearly semantically similar classes. Nevertheless, semantic clusters can be learned in a unsupervised fashion leveraging

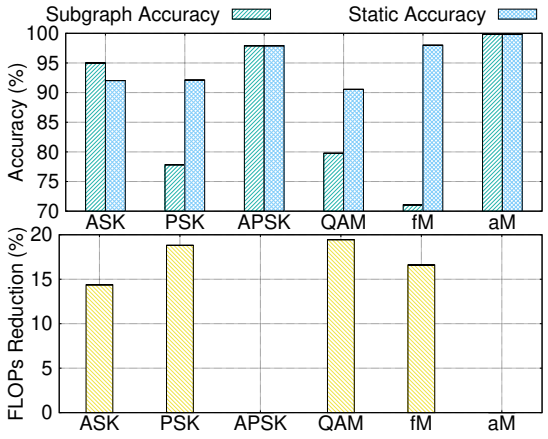

Figure 10: VGG10 sub-DNN performances in terms of classification accuracy and FLOPs reduction.

the transmissions between UAVs. As such, we train the DyNN using supervised contrastive learning [14]. Afterwards, we cluster the radios though K-means on the intermediate features of the DyNN. This kind of loss function independently learns to group closer the features belonging to similar classes. This in turn, enhances the semantic clusters identification. Table 3 shows the probabilities for each radio to belong to a cluster. The number of semantic clusters is defined in a supervised fashion and fed as input to the K-means algorithm. Interestingly, radios grouped in the same cluster share comparable amplitude impairments and subsequently similar injected IDs. The *Auxiliary* DNN in the UAV fingerprinting problem achieves high prediction accuracy (between 95-99%) resorting on a single fully connected layer and low confidence thresholds (about 10%). Such low confidence threshold allows sub-DNN activation for 97.9% of the inferences, resulting in latency reduction.

3.3.3 Sub-DNNs extraction. The orchestration framework extracts and evaluates sub-DNNs for VGG4, VGG6, VGG8 and VGG10 based on the Auxiliary DNN preliminary classification. To minimize the waiting time of the Program Buffer, the first convolution block (for VGG4 and VGG6 in radio fingerprinting) and up to the second convolution block (for VGG8 and VGG10 in modulation classification) are set to be static. The filters of the dynamic layers are ranked utilizing the DCS (Discriminative Capability Score) algorithm proposed in [28]. The Discriminative Capability Score algorithm scores the filters based on their capability to best discriminate among the classes of a given semantic cluster. The filters are ranked using these scores. Sub-DNNs are build, for each semantic cluster, by excluding the filters with the least scores in each dynamic layer. The exclusion rates are tuned evaluating the performance (in terms of FLOPs reduction and accuracy drop) of several candidate sub-DNNs.

Modulation Classification. Figure 10 shows the performance of the optimal sub-DNNs extracted from VGG10 when SNR is 26dB. The sub-DNN accuracy drop with respect to full static inference is not monotonically dependent with respect to the exclusion rates. In fact, while QAM's sub-DNN excludes 19.5% of the FLOPs resulting in 10.78% accuracy drop, for frequency modulations excluding 16% of FLOPs leads to 26.94% decrease in accuracy. Therefore, the accuracy drop strongly depends on the capability of filters to recognize semantic clusters. In fact, the analog modulation sub-DNN in VGG8, dynamically excludes 28% of FLOPs resulting in improvement up to 10% in classification accuracy. Similarly, ASK's sub-DNN extracted from VGG10 reduces the FLOPs by 14.38% and achieves 95% detection accuracy (3% more than the static scenario). The overall accuracy drop of the system (VGG+Aux) is 8% for VGG8 and 5% for VGG10. Although individual sub-DNN models may experience a substantial accuracy drop (up to 27%), the overall decline remains moderate, as many input samples are filtered out by the Auxiliary DNN based on the confidence threshold.

Radio Fingerprinting. Due to the limited number of semantic clusters and complexity of the accelerators adapted to the UAV fingerprinting problem, it has been possible to conduct an exhaustive search of the optimal sub-DNNs. This allowed to identify high-performing sub-DNNs in VGG6 able to dynamically reduce up to 69% of FLOPs without losing in accuracy (less than 3%).

3.4 Area, Power and Latency Evaluation

The pre-trained VGG models, the Auxiliary DNN and the sub-DNNs are then processed by the orchestration framework to generate, test and synthesize the hardware prototypes (steps 4-9). The confidence threshold is set to 80% in modulation classification and 10% for UAV identification. The synthesized hardware works on a fixed point number representation with 10 bits integer and 22 bits fractional parts. This design decision achieves comparable computation accuracy with respect to a floating point implementation yet consumes lower resources and optimizes computation delay. We exhaustively evaluate the synthesized cores in terms of resources utilization, power consumption and inference latency.

IP	LUTs	FF	CARRY8	CLB	BRAM	DSP
Aux(MC)	2.4%	2.91%	1.14%	8.52%	15.63%	1.98%
Aux(UAV)	1.1%	0.75%	0.69%	2.25%	0.33%	0.52%
VGG4	1.3%	0.92%	0.76%	2.81%	3.02%	0.95%
VGG6	2%	1.36%	1.21%	4.28%	5.37%	1.59%
VGG8	8.16%	5.63%	10.06%	19.70%	5.26%	4.52%
VGG10	15.08%	8.37%	15.08%	35.48%	10.31%	5.12%

Table 4: Area utilization.

3.4.1 Area. Table 4 reports the resource utilization of the hardware components. The overall additional resources allocated for the Auxiliary DNN are negligible when compared against the amount of hardware employed by the DyNNs. The main cost to deploy the Auxiliary DNN is in BRAM resources. In fact, up to 15.63% of the available BRAM blocks are consumed to instantiate the instruction ROM in the Auxiliary DNN architecture. Furthermore, the amount of instructions and in turn of BRAM resources are strongly application dependent. In fact, the size of the instruction ROM is heavily influenced by: *(i)* number of semantic clusters, *(ii)* number of dynamic layers *(iii)* number of filters in the dynamic layers. In resource-constrained platforms, the instructions must be stored in the DDR memory. Due to the parallel structure of the proposed inference process (see Sec. 3.2), the additional delay required to access the DDR can be compensated by incrementing the number of static layers in the DyNN. Although fewer dynamic layers reduce the speed-up for dynamic inferences, the static inference end-to-end latency is preserved.

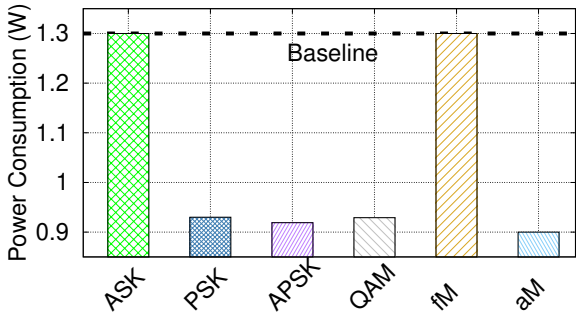

Figure 11: Power consumption for VGG8 as a function of the predicted semantic cluster. The black dotted baseline the power consumption for static inference.

3.4.2 Power consumption. We used HLS to obtain an estimation of the hardware power consumption. Table 5 reports the power utilization required for static DNN inference. As expected, the power consumption is heavily influenced by the accelerated DNNs complexity. Nevertheless, while the power consumption gap between VGG6 and VGG8 is substantial, the rise in FLOPs is only 50%. On the other hand, the number of burst accesses rises from 817 to 2,914. This indicates that for the proposed architecture, the power consumption is primarily due to memory accesses.

IP	Power consumption	Burst accesses
Aux	0.65W	120
Aux(UAV)	89mW	6
VGG4	162mW	283
VGG6	313mW	817
VGG8	1.3W	2,914
VGG10	2.45W	5,063

Table 5: Power consumption.

Overall, the additional power consumption required to deploy the the Auxiliary DNN is between 8.4-9.2% for the modulation classification and 2.01-2.09% for UAV identification. Nevertheless, dynamically pruning filters is expected to reduce the number of burst memory accesses and switching activity. As illustrated in Figure 11, the VGG8's power consumption highly fluctuates according to the predicted semantic cluster. We depicted in dotted black the static power consumption of the accelerator which we leverage as a benchmark. Up to 0.4 W can be saved by activating sub-DNNs during the inference process. Therefore, energy compensation is achieved whenever the Auxiliary DNN produces high confidence classifications. In fact, the overall additional power consumption required to deploy the Auxiliary network can drop from 9.2% to as little as 3.53% for VGG8.

3.4.3 Latency. An AXI-Timer is incorporated in the PL to measure the inference latency of the studied accelerators directly in hardware. The AXI-Timer is powered with the same clock frequency of the designed hardware accelerator. The HLS tools indicates 200MHz as the highest working frequency. The DyNN and Auxiliary DNN computation is enabled simultaneously. At the same time the AXI-Timer counting is triggered. As soon as the computation is completed, the AXI-Timer is disabled, the output BRAM content monitored and the inference latency measured.

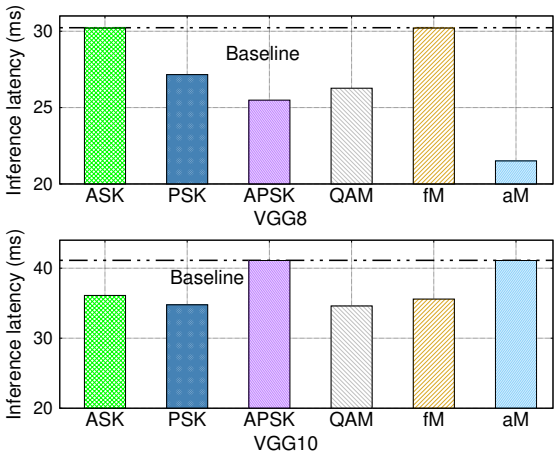

Figure 12: Inference latency for VGG8 and VGG10 accelerators according to the input cluster

Figure 12 depicts the inference latency for VGG8 and VGG10 accelerators as a function of the predicted cluster. The key take away are: *(i)* irrespective to the semantic class, the reconfiguration delay is negligible, *(ii)* the system achieves up to 28.80% lower latency with respect to deploying the VGG accelerators as a stand alone module (statically). In fact, the prediction latency of VGG10 when the Auxiliary DNN detects APSK (no FLOPs reduction) is only 0.02 ms higher than deploying the VGG accelerator statically (validating point (i)). This suggests that the latency required by the `FlexRFML` cores to fetch instruction from the Program Buffer and reconfigure the data flow is negligible. Regardless to the predicted cluster, the algorithm is computed in less than 30.2 ms for VGG8 and 41.1 ms for VGG10. Furthermore, according to the prediction of the Auxiliary classifier, the latency is optimized up to 28.80% for VGG8 and 15.8% for VGG10.

IP	Semantic Cluster	Latency	Acceleration
VGG4	*Cluster0*	4.94ms	10.1%
VGG4	*Cluster1*	5.05ms	9.2%
VGG6	*Cluster0*	9.44ms	0%
VGG6	*Cluster1*	6.08ms	35.6%

Table 6: Inference latencies as a function of the predicted semantic cluster in the UAV fingerprinting problem.

Table 6 shows the measured latencies and the resulting acceleration with respect to static inference for VGG4 and VGG6 according to the perceived radio signal. The static prediction delays are 5.55 ms for VGG4 and 9.44 ms for VGG6. Up to 35.6% acceleration is achieved for VGG6 when the auxiliary detects comunications from a radio belonging to *Cluster1*. The average speed up of the system strongly depends on the input signal sequence and the confidence threshold. For UAV identification, this metric can be easily measured due to the low confidence threshold. In particular, assuming

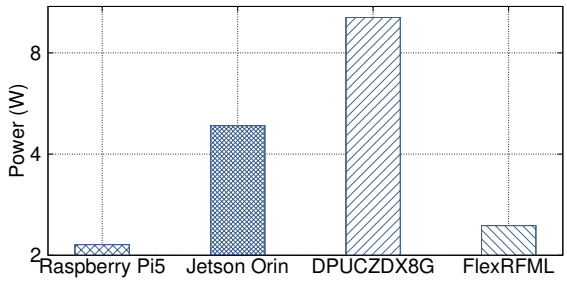

Figure 13: Power consumption of different platforms while computing VGG10

a similar amount of data transfer from each radio, the dynamic system achieves an average acceleration of 9.65% for VGG4 and 17.8% for VGG6.

3.5 FlexRFML vs GPU, CPU, and DPU

In this section, we compare the performances of FlexRFML with several benchmarks. Several testbeds have been designed to measure inference latency on a *Xilinx DPUCZDX8G*, a *Jetson Orin Nano*, a Rasberry Pi5, and a Cortex A53 64 bit quad core processor, which is the PS of the considered Xilinx board. To ensure a fair comparison across heterogeneous platforms, latencies are normalized based on the accelerator's operating clock frequencies. Table 7 reports the latencies, measured in clock cycles, for the acceleration of VGG10. The key limitations in accelerating DyNN on state of the art embedded accelerators are *(i)* the automatic sub-DNNs extraction and execution is not supported, *(ii)* the lack of a clear correlation between theoretical analysis (FLOPs reduction) and experimental performance (latency optimization).

Cluster	*RasberryPi5*	*CortexA53*	*JetsonOrinNano*	*DPUCZDX8G*	FlexRFML
ASK	54.1x10^6	106x10^6	2.7x10^6	2.72x10^6	7.2x10^6
PSK	43.3x10^6	103x10^6	2.63x10^6	2.76x10^6	6.94x10^6
APSK	30.5x10^6	124x10^6	2.58x10^6	2.73x10^6	8.2x10^6
QAM	37.8x10^6	101x10^6	2.67x10^6	2.74x10^6	6.92x10^6
fM	44.0x10^6	106x10^6	2.72x10^6	2.739x10^6	7.12x10^6
aM	42.8x10^6	124x10^6	2.57x10^6	2.74x10^6	8.2x10^6

Table 7: Software and hardware acceleration of VGG10 measured in clock cycles.

The former limitation has been addressed through a model selection based approach. In detail, the sub-DNNs are pre-compiled, stored in the accelerator's memory, and dynamically selected at inference time. Despite its lack in scalability, this procedure was essential in mitigating reconfiguration delays demanded by dynamic sub-DNN extraction, which takes 130–150 ms on GPUs, up to 500 ms on CPUs, and is not supported by DPUs. The limitation of CPU and DPU is underlined by the results in Table 7. Despite achieving lower latencies, a GPU or DPU based acceleration does not benefit from the reduced computation complexity of the pre-compiled sub-DNNs. This further supports our design choice to accelerate the dynamic convolutions using the sliding window core instead of parallel processing elements executing convolutions through matrix multiplications. Moreover, both DPU and GPU accelerators demand significant power consumption, with the DPU using 10.62W and the GPU drawing 4.86W (see Fig. 13). Although a CPU-based acceleration offers lower performances, it can still achieve latency optimization through dynamic inference. Indeed, the C-program executed by the Cortex® A53, inspired by the FlexRFML architecture, obtains 15% acceleration for APSK. On the other hand, state of the art libraries for CPU inference (i.e, PyTorch), despite optimized performances, achieve no correlation between theoretical and experimental analysis.

4 Related Work

Radio spectrum perception has garnered significant attention from the research community [3, 12]. Early work has focused on narrowband approaches such as energy detection and matched filtering, which require sensitive time to monitor the spectrum and estimate occupancy across a broader frequency range [10]. For this reason, prior work has adopted DNNs in a variety of wideband radio spectrum perception applications like radio fingerprinting [23], detection of unused and/or underutilized spectrum portions [30] and wireless protocol classification [31], among others[25]. The key issue with such prior work is that the DNN inference is performed offline with pre-collected wireless data.

Real-time Spectrum Perception. Although early work demonstrated the feasibility of real-time DNN-based radio spectrum perception on a SoC architecture [24, 26], such work assumes the architecture of the DNN is *static*. On the other hand, given their constrained and dynamic nature, radio devices may require a different trade-off between DNN accuracy (i.e., more depth, weights, computation) and energy consumption (i.e., less depth, weights, computation) during different periods of their lifetime. As such, DyNNs may bring unprecedented advantages both in terms of system performances, security against adversarial attacks [6] and accuracy through techniques such as lifelong learning [21].

Dynamic neural networks. For an excellent survey on DyNNs, we refer to [11]. Regarding hardware-based DyNNs, existing work has explored dynamic depth (i.e., *early exiting*) and dynamic width (i.e., *channel skipping*) DNNs. The work in [15] proposes an adaptive pruning and early exiting (AdaPEx) framework for FPGA, which resorts on a run-time manager to adaptively select from a library the most suitable DNN configuration in terms of early exits and pruning rates according to the workload. However, [15] considers the number of processing elements in determining the pruning rates, which implies pruning patterns constraints. Furthermore, dynamically changing the pruning rates requires reconfiguring the FPGA fabric which has been shown to take up to 42 ms in some cases [8] and is not sustained in ASIC. Currently, for the best of our knowledge, software-to-hardware automatic frameworks can only sustain dynamic-depth DNN deployment on FPGA. The newest work on the topic [4] proposes an orchestration tool to map the software description of any user defined dynamic-depth neural network to hardware prototypes. Unfortunately, such tool tailored to channel-skipping DyNNs has not been designed yet. Other work focus on computation precision dynamism. The latter consist in adapting the precision in real-time based on the workload to trade-off the DNN prediction accuracy with computation latency. [16] proposed an architecture including both low- and high-precision cores characterized by different quantization levels and parallelism.

Although the low-precision core computes the majority of the workload, the end-to-end latency increases when the high-precision core has to be executed. As such, we have designed an architecture that exploits parallel computation. To the best of our knowledge, prior work has not considered tailoring dynamic-width neural networks for radio spectrum perception.

5 Conclusion

In this work, we have proposed FlexRFML, the first hardware accelerator designed to sustain *channel skipping* DyNN computation in real-time through a software-based reconfiguration compatible with FPGA-to-ASIC conversion. As such, we provided a detailed walk through of the FlexRFML architecture operations and data flow. Building on a parametric description of the accelerator, we proposed a software-to-hardware orchestration engine that allows prototyping any user defined *channel skipping* DyNN. We extensively evaluated several FlexRFML prototypes tailoring the architectures to the modulation classification and UAV fingerprinting problems. To further validate the performances of the proposed architecture, we cross compared its performances in the area/power/latency/accuracy design space against high performances embedded devices. Experimental results revealed that FlexRFML improves inference acceleration up to 35.6%.

6 Acknowledgements

This work has been supported by the NSF under grants CCF-2218845, ECCS-2229472 and ECCS-2329013; by the Air Force Office of Scientific Research under grant FA9550-23-1-0261; and by the Office of Naval Research under grant N00014-23-1-2221.

References

[1] Reza Amini Gougeh and Zeljko Zilic. 2024. Systematic Review of IoT-Based Solutions for User Tracking: Towards Smarter Lifestyle, Wellness and Health Management. *Sensors* (2024). https://doi.org/10.3390/s24185939

[2] Analog Devices Incorporated. 2018. AD9361 RF Agile Transceiver Data Sheet. http://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf.

[3] Youness Arjoune and Naima Kaabouch. 2019. A Comprehensive Survey on Spectrum Sensing in Cognitive Radio Networks: Recent Advances, new Challenges, and Future Research Directions. *Sensors* 19, 1 (2019), 126.

[4] Benjamin Biggs, Christos-Savvas Bouganis, and George Constantinides. 2023. ATHEENA: A Toolflow for Hardware Early-Exit Network Automation. *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2023), 121–132. https://doi.org/10.1109/FCCM57271.2023.00022

[5] Xuyi Cai, Ying Wang, Xiaohan Ma, Yinhe Han, and Lei Zhang. 2022. DeepBurning-SEG: Generating DNN Accelerators of Segment-Grained Pipeline Architecture. *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1396–1413. https://doi.org/10.1109/MICRO56248.2022.00094

[6] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*. Ieee, 39–57.

[7] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 4 (2022), 1–42.

[8] Mohammad Farhadi, Mehdi Ghasemi, and Yezhou Yang. 2019. A Novel Design of Adaptive and Hierarchical Convolutional Neural Networks using Partial Reconfiguration on FPGA. (2019). arXiv:1909.05653 [cs.CV]

[9] Federal Communications Commission (FCC). [n. d.]. Spectrum Crunch. https://www.fcc.gov/general/spectrum-crunch.

[10] Jiabao Gao, Xuemei Yi, Caijun Zhong, Xiaoming Chen, and Zhaoyang Zhang. 2019. Deep Learning for Spectrum Sensing. *IEEE Wireless Communications Letters* 8, 6 (2019), 1727–1730.

[11] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 11 (2021), 7436–7456.

[12] Jithin Jagannath, Nicholas Polosky, Anu Jagannath, Francesco Restuccia, and Tommaso Melodia. 2019. Machine Learning for Wireless Communications in the Internet of Things: A Comprehensive Survey. *Ad Hoc Networks* 93 (2019), 101913.

[13] Timothy K Johnsen and Marco Levorato. 2024. NaviSlim: Adaptive Context-Aware Navigation and Sensing via Dynamic Slimmable Networks. *2024 IEEE/ACM Ninth International Conference on Internet-of-Things Design and Implementation (IoTDI)* (2024), 110–121. https://doi.org/10.1109/IoTDI61053.2024.00014

[14] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. *Advances in Neural Information Processing Systems* 33 (2020), 18661–18673. https://proceedings.neurips.cc/paper_files/paper/2020/file/d89a66c7c80a29b1bdbab0f2a1a94af8-Paper.pdf

[15] Guilherme Korol, Michael Guilherme Jordan, Mateus Beck Rutzig, Jeronimo Castrillon, and Antonio Carlos Schneider Beck. 2023. Pruning and Early-Exit Co-Optimization for CNN Acceleration on FPGAs. (2023), 1–6.

[16] Alexandros Kouris, Stylianos I Venieris, and Christos-Savvas Bouganis. 2018. $Cascade^{CNN}$: Pushing the performance limits of quantisation in convolutional neural networks. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), 155–1557.

[17] Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. 2021. Dynamic Slimmable Network. (2021), 8607–8617.

[18] Olivier Maquelin, Guang R Gao, Herbert HJ Hum, Kevin B Theobald, and Xin-Min Tian. 1996. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. *ACM SIGARCH Computer Architecture News* 24, 2 (1996), 179–188.

[19] Subhramoy Mohanti, Nasim Soltani, Kunal Sankhe, Dheryta Jaisinghani, Marco Di Felice, and Kaushik Chowdhury. 2020. AirID: Injecting a Custom RF Fingerprint for Enhanced UAV Identification using Deep Learning. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE, 1–6.

[20] Timothy James O'Shea, Tamoghna Roy, and T Charles Clancy. 2018. Over-the-air Deep Learning Based Radio Signal Classification. *IEEE Journal of Selected Topics in Signal Processing* 12, 1 (2018), 168–179.

[21] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. 2019. Continual Lifelong Learning with Neural Networks: A Review. *Neural networks* 113 (2019), 54–71.

[22] Nitin Pundir, Sohrab Aftabjahani, Rosario Cammarota, Mark Tehranipoor, and Farimah Farahmandi. 2022. Analyzing security vulnerabilities induced by high-level synthesis. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 18, 3 (2022), 1–22.

[23] Francesco Restuccia, Salvatore D'Oro, Amani Al-Shawabka, Mauro Belgiovine, Luca Angioloni, Stratis Ioannidis, Kaushik Chowdhury, and Tommaso Melodia. 2019. DeepRadioID: Real-Time Channel-Resilient Optimization of Deep Learning-based Radio Fingerprinting Algorithms. *Proc. of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)* (2019).

[24] Francesco Restuccia and Tommaso Melodia. 2019. Big Data Goes Small: Real-Time Spectrum-Driven Embedded Wireless Networking Through Deep Learning in the RF Loop. *Proc. of IEEE Conference on Computer Communications (INFOCOM)* (2019).

[25] Francesco Restuccia and Tommaso Melodia. 2020. Deep Learning at the Physical Layer: System Challenges and Applications to 5G and Beyond. *IEEE Communications Magazine* 58, 10 (2020), 58–64.

[26] Francesco Restuccia and Tommaso Melodia. 2020. DeepWiERL: Bringing Deep Reinforcement Learning to the Internet of Self-Adaptive Things. *Proceedings of IEEE Conference on Computer Communications (INFOCOM)* (2020).

[27] Otavio Salvador and Daiane Angolini. 2014. *Embedded Linux Development with Yocto Project.* Packt Publishing Ltd.

[28] Sazzad Sayyed, Jonathan Ashdown, and Francesco Restuccia. 2023. Faster and Accurate Neural Networks with Semantic Inference. *arXiv preprint arXiv:2310.01259* (2023).

[29] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[30] Daniel Uvaydov, Salvatore D'Oro, Francesco Restuccia, and Tommaso Melodia. 2021. DeepSense: Fast Wideband Spectrum Sensing Through Real-Time In-the-Loop Deep Learning. In *Proc. of IEEE Intl. Conf. on Computer Communications (INFOCOM)*. Vancouver, BC, Canada.

[31] Daniel Uvaydov, Milin Zhang, Clifton Paul Robinson, Salvatore D'Oro, Tommaso Melodia, and Francesco Restuccia. 2024. Stitching the Spectrum: Semantic Spectrum Segmentation with Wideband Signal Stitching. *Proceedings of IEEE Conference on Computer Communications (INFOCOM)* (2024).

[32] AXI Xilinx. 2011. Reference Guide, UG761 (v13. 1). *URL http://www. xilinx. com/support/documentation/ip documentation/ug761 axi reference guide. pdf* (2011).